OpenRSP Developer Manual
Version 1.0.0

Radovan Bast
Daniel H. Friese
Bin Gao
Dan J. Jonsson
Magnus Ringholm
Kenneth Ruud
Andreas Thorvaldsen

Centre for Theoretical and Computational Chemistry (CTCC)
Department of Chemistry
University of Tromsø—The Arctic University of Norway
N–9037, Tromsø, Norway

OpenRSP : open-ended library for response theory
© 2015
Radovan Bast
Daniel H. Friese
Bin Gao
Dan J. Jonsson
Magnus Ringholm
Kenneth Ruud
Andreas Thorvaldsen

OpenRSP is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

OpenRSP is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with OpenRSP . If not, see <http://www.gnu.org/licenses/>.

HOWEVER, YOU CANNOT USE IDEAS, DATA, OR WORDING OF THIS DOCUMENT WITHOUT AN APPROPRIATE CITATION.

Contents

Chapter 1

Preface

1.1 Version Numbering Scheme

The version numbering scheme used for OpenRSP is **Major.Minor.Patch**, where

Major generally represents new release containing important changes, and it may not be compatible with previous versions.

Minor is changed when a few new features introduced, and it should be compatible with those of the same **Major** number.

Patch usually includes bug fixes, and should be compatible with those of the same **Major.Minor** number.

Chapter 2

Analysis

This chapter presents the "Software Requirements Specification" (SRS) largely based on ISO/IEC/IEEE 29148:2011(E) "Systems and software engineering – Life cycle processes – Requirements engineering".

2.1 Introduction

This section gives a scope description and overview of everything needed to design and implement OpenRSP (library) by taking the requirements listed in this section into account.

2.1.1 Purpose

The SRS analyses the problem—molecular integral evaluation—that OpenRSP aims to solve, and lays out the requirements that OpenRSP should supply or should not supply.

The SRS is primarily intended to prepare a reference for developing OpenRSP (Version 1.0.0). It can also be proposed to OpenRSP users for their suggestions and comments.

2.1.2 Scope

2.1.3 Product Overview

Product Perspective

Product Functions

User Characteristics

Limitations

2.1.4 Definitions

2.2 References

2.3 Specific Requirements

2.3.1 External Interfaces

2.3.2 Functions

2.3.3 Usability Requirements

2.3.4 Performance Requirements

2.3.5 Logical Database Requirements

2.3.6 Design Constraints

2.3.7 Software System Attributes

2.3.8 Supporting Information

2.4 Verification

2.5 Appendices

2.5.1 Assumptions and Dependencies

2.5.2 Acronyms and Abbreviations

2.5.3 Theoretical Background

For the time being, OpenRSP has implemented the density matrix-based quasienergy formulation of the Kohn–Sham density functional response theory using perturbation- and time-dependent basis sets [1, 2].

The density matrix-based quasienergy formulation actually works for different levels of theory, i.e., one-, two- and four-component levels. A relativistic implementation can be found in Ref. [3].

OpenRSP uses the recursive programming techniques [4] to compute different molecular properties order by order. The recursive programming techniques can also be used for calculations of residues, the implementation of the first order residues can be found in Ref. [5].

2.5.4 Open-Ended Response Theory

The name OpenRSP stands for **open-ended response theory**, that is, the library is:

1. open-ended for different levels of theory, i.e., one-, two- and four-component levels;
2. open-ended for different wave functions, e.g., atomic-orbital (AO) based density matrix, molecular orbital (MO) cofficients and coupled cluster (CC);
3. open-ended for different kinds of perturbations; and
4. open-ended for different host programs.

As aformentioned, OpenRSP has for the time being implemented the AO based density matrix response theory (source codes in `src/ao_dens`)[1], and it works for one-, two- and four-component levels by simply setting the appropriate Hamiltonian. We are now planning to implement the MO and CC based response theories.

To make OpenRSP work for any perturbation, we will implement the so called **perturbation free scheme**, see Section 2.5.6.

In order to make it easy for implementing OpenRSP into different host programs (written in different programming languages), we agree to use the **callback function scheme** in OpenRSP in the 2015 Skibotn meeting. The callback functions are specified by host programs by calling the OpenRSP application program interface (APIs, both C and Fortran implemented) during run time, and will be used by OpenRSP during calculations, to get contributions from electronic and nuclear Hamiltonian, and to get response parameters from solving the linear response equation.

Another important issue affects the implementation of OpenRSP into different host programs is the matrix and its different operations that OpenRSP extensively depends on. Different host programs can have different types of matrices (dense and sparse, sequential and parallel) and written by different programming languages (e.g. C and Fortran).

To best utilize the host program's developed matrix routines (if there is), and also to remove this complexity of matrix problem from OpenRSP , we also agree to build OpenRSP on top of the **QcMatrix library** in the 2015 Skibotn meeting. This matrix library works as an adapter between OpenRSP and different matrix routines (implemented in different host programs) that can be written in C and Fortran[2].

2.5.5 OpenRSP Framework

Therefore, a full picture of OpenRSP used in a C host program can be (the description of OpenRSP Fortran APIs can be found in Section 3.12):

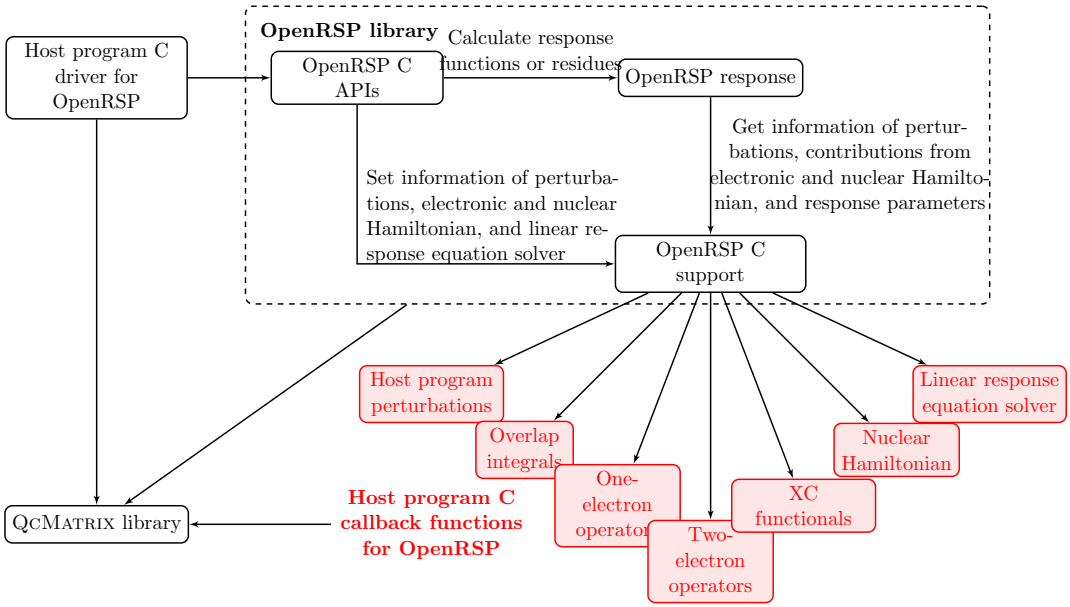

Figure 2.1: OpenRSP used in a C host program.

As shown in Figure 2.1, the OpenRSP library is divided into three parts:

[1] The codes in `src/ao_dens` are written in Fortran, but OpenRSP APIs are implemented using C language. Therefore, adapter codes between them are implemented in `src/ao_dens/adapter`, for OpenRSP APIs calling the codes of AO based density matrix response theory, also for the AO based density matrix response theory codes calling the callback functions (as function pointers saved by OpenRSP APIs).

[2] If there is no matrix routines implemented in a host program, it can fully use the QcMatrix library that will invoke BLAS and LAPACK libraries for matrix operations.

1. The "OpenRSP C APIs" work mostly between the host program driver routine and other parts of the OpenRSP library, that all the information saved in the "OpenRSP C support" will be set up by calling the corresponding OpenRSP C API;
2. The "OpenRSP response" is the core part in which the performance of response theory will be done;
3. The "OpenRSP C support" saves the information of perturbations, electronic and nuclear Hamiltonian and linear response equation solver, and will be used by the "OpenRSP response" part during calculating response functions and residues.

The "OpenRSP response" was already implemented using Fortran for the AO based density matrix response theory (source codes in `src/ao_dens`) that will not be covered here.

2.5.6 Perturbations

For perturbations in OpenRSP , we introduce the following notations and convention:

Perturbation is described by a label (a), a complex frequency (ω) and its order (n), and written as a^n_ω Any two perturbations are different if they have different labels, and/or frequencies, and/or orders.

Perturbation label is an integer distinguishing one perturbation from others; all *different* perturbation labels involved in the calculations should be given by calling the application programming interface (API) `OpenRSPSetPerturbations()`; OpenRSP will stop if there is any unspecified perturbation label given afterwards when calling the APIs `OpenRSPGetRSPFun()` or `OpenRSPGetResidue()`.

Perturbation order Each perturbation can acting on molecules once or many times, that is the order of the perturbation.

Perturbation components and their ranks Each perturbation may have different numbers of components for their different orders, the position of each component is called its rank.
For instance, there will usually be x, y, z components for the electric dipole perturbation, and their ranks are {`0,1,2`} in zero-based numbering, or {`1,2,3`} in one-based numbering.

Perturbation tuple An ordered list of perturbation labels, and in which we further require that *identical perturbation labels should be consecutive*. That means the tuple (a, b, b, c) is allowed, but (a, b, c, b) is illegal because the identical labels b are not consecutive.
As a tuple:

1. Multiple instances of the same labels are allowed so that $(a, b, b, c) \neq (a, b, c)$, and
2. The perturbation labels are ordered so that $(a, b, c) \neq (a, c, b)$ (because their corresponding response functions or residues are in different shapes).

We will sometimes use an abbreviated form of perturbation tuple as, for instance $abc \equiv (a, b, c)$.
Obviously, a perturbation tuple + its corresponding complex frequencies for each perturbation label can be viewed as a set of perturbations, in which the number of times a label (with the same frequency) appears is the order of the corresponding perturbation.
For example, a tuple (a, b, b, c) + its complex frequencies $(\omega_a, \omega_b, \omega_b, \omega_c)$ define perturbations $a^1_{\omega_a}$, $b^2_{\omega_b}$ and $c^1_{\omega_c}$; another tuple (a, b, b, c) + different complex frequencies for labels b—$(\omega_a, \omega_{b_1}, \omega_{b_2}, \omega_c)$ define different perturbations $a^1_{\omega_a}$, $b^1_{\omega_{b_1}}$, $b^1_{\omega_{b_2}}$ and $c^1_{\omega_c}$.

Canonical order

1. In OpenRSP , all perturbation tuples are canonically orderd according to the argument `pert_tuple` in the `OpenRSPGetRSPFun()` or `OpenRSPGetResidue()`. For instance, when a perturbation tuple (a, b, c) given as `pert_tuple` in the API `OpenRSPGetRSPFun()`, OpenRSP will use such order $(a > b > c)$ to arrange all perturbation tuples inside and sent to the callback functions.
2. Moreover, a collection of several perturbation tuples will also follow the canonical order. For instance, a collection of all possible perturbation tuples of labels a, b, c are $(0, a, b, c, ab, ac, bc, abc)$, where 0 means unperturbed quantities that is always the first one in the collection.

Perturbation a The first perturbation label in the tuple sent to OpenRSP APIs `OpenRSPGetRSPFun()` or `OpenRSPGetResidue()`, are the perturbation a [1].

Perturbation addressing

1. The addressing of perturbation labels in a tuple, as mentioned in the term **Canonical order**, is decided by
   (a) the argument `pert_tuple` sent to the API `OpenRSPGetRSPFun()` or `OpenRSPGetResidue()`, and
   (b) the canonical order that OpenRSP uses.
2. The addressing of components per perturbation (several consecutive identical labels with the same complex frequency) are decided by the host program, as will be discussed in the following **perturbation free scheme**.
3. The addressing of a collection of perturbation tuples follows the canonical order as aforementioned.

Therefore, the shape of response functions or residues is mostly decided by the host program. Take $\mathcal{E}^{abbc}$ for example, its shape is (N_a, N_{bb}, N_c), where N_a and N_c are respectively the numbers of components of the first order of the perturbations a and c, and N_{bb} is the number of components of the second order of the perturbation b, and

1. In OpenRSP , we will use notation `[a][bb][c]` for $\mathcal{E}^{abbc}$, where the leftmost index (`a`) runs slowest in memory and the rightmost index (`c`) runs fastest. However, one should be aware that the results are still in a one-dimensional array.
2. If there two different frequencies for the perturbation label b, OpenRSP will return `[a][b1][b2][c]`, where `b1` and `b2` stand for the components of the first order of the perturbation b.
3. The notation for a collection of perturbation tuples (still in a one-dimensional array) is `{1,[a],[b],[c],[a][b],[a][c],[b][c],[a][b][c]}` for $(0, a, b, c, ab, ac, bc, abc)$, where as aforementioned the first one is the unperturbed quantities.

Perturbation Free Scheme

Now, let us discuss our **perturbation free scheme**. As aforementioned, there could be **different numbers of components** for different perturbations. In different host programs, these components could **be arranged in different ways**.

For instance, there are 9 components for the second order magnetic derivatives in a redundant way $xx, xy, xz, yx, yy, yz, zx, zy, zz$, but 6 components in a non-redundant way xx, xy, xz, yy, yz, zz. There are at most four centers in different integrals, non-zero high order (≥ 5) geometric derivatives are only those with at most four differentiated centers.

To take all the above information into account in OpenRSP will make it so complicated and not necessary, because response theory actually does not care about the detailed knowledge of different perturbations. In particular, when all the (perturbed) integrals and expectation values are computed by the host program's callback functions, the detailed information of perturbations:

1. the number of components, and
2. how they are arranged in memory

can be hidden from OpenRSP .

The former can be easily solved by sending the numbers of components of different perturbation labels (up to their maximum orders) to the OpenRSP API `OpenRSPSetPerturbations()`.

The latter can be important for OpenRSP to construct higher-order derivatives from lower-order ones. We have two cases:

1. Higher-order derivatives are taken with respect to different perturbations, for instance, $\frac{\partial^3}{\partial a \partial b \partial c}$ are simply the direct product of components of lower-order derivatives with respect to each perturbation $\frac{\partial}{\partial a}$, $\frac{\partial}{\partial b}$ and $\frac{\partial}{\partial c}$.
2. Higher-order derivatives are taken with respect to **one perturbation**. Take the second order-derivatives (in the redundant format) for example, they can be constructed from the

first-order ones as,

$$
\begin{aligned}
x + x &\to xx, \;\; 0 + 0 \to 0, \\
x + y &\to xy, \;\; 0 + 1 \to 1, \\
x + z &\to xz, \;\; 0 + 2 \to 2, \\
y + x &\to yx, \;\; 1 + 0 \to 3, \\
y + y &\to yy, \;\; 1 + 1 \to 4, \\
y + z &\to yz, \;\; 1 + 2 \to 5, \\
z + x &\to zx, \;\; 2 + 0 \to 6, \\
z + y &\to zy, \;\; 2 + 1 \to 7, \\
z + z &\to zz, \;\; 2 + 2 \to 8,
\end{aligned}
$$

where we have ranked different components in zero-based numbering (numbers on the right). Because the ranks can be different in different host programs, also the above mapping relationship between lower- and higher-order derivatives (with respect to **one perturbation**) can be different in different host programs.

We therefore ask for a callback function `get_pert_concatenation()` from host programs. This callback function will, from given components of a **concatenated perturbation tuple** (i.e. higher-order derivatives with respect to one perturbation), get the ranks of components of the **sub-perturbation tuples with the same perturbation label** (i.e. lower-order derivatives with respect to one perturbation).

As such, the numbers of different components of perturbations and their ranks are totally decided by the host program—that is the **perturbation free scheme**.

Chapter 3

Implementation

3.1 Header File for Users

To use OpenRSP , C users need to include the following header file into their codes:

13 ⟨*OpenRSP.h* 13⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

  <header name='OpenRSP.h' author='Bin Gao' date='2014-01-27'>
    The header file of OpenRSP library for users
  </header>
*/

#if !defined(OPENRSP_H)
#define OPENRSP_H

/* host program perturbations */
#include "RSPPerturbation.h"
/* type of electronic wave function */
/*#include "RSPWaveFunction.h"*/
/* overlap integrals */
#include "RSPOverlap.h"
/* one-electron operators */
#include "RSPOneOper.h"
/* two-electron operators */
#include "RSPTwoOper.h"
/* exchange-correlation (XC) functionals */
#include "RSPXCFun.h"
/* nuclear Hamiltonian */
#include "RSPNucHamilton.h"
/* linear response equation solver */
#include "RSPSolver.h"

⟨OpenRSPStruct 14b⟩

⟨OpenRSPAPIs 14c⟩

#endif
```

Here, the directives `#if !defined(OPENRSP_H)` and `#define OPENRSP_H` (**include guard**) to-

gether prevent the header file from being compiled more than once.

We plan to release OpenRSP under the GNU Lesser General Public License:

14a ⟨*OpenRSPLicense* 14a⟩≡

```
OpenRSP: open-ended library for response theory
Copyright 2015 Radovan Bast,
               Daniel H. Friese,
               Bin Gao,
               Dan J. Jonsson,
               Magnus Ringholm,
               Kenneth Ruud,
               Andreas Thorvaldsen

OpenRSP is free software: you can redistribute it and/or modify
it under the terms of the GNU Lesser General Public License as
published by the Free Software Foundation, either version 3 of
the License, or (at your option) any later version.

OpenRSP is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with OpenRSP. If not, see <http://www.gnu.org/licenses/>.
```

In the following sections, we will describe how to implement each component of the "OpenRSP C support" (red blocks in Figure 2.1 under the "OpenRSP C support") and the corresponding "OpenRSP C API". Each component will have its header file, implemented C `struct` and corresponding functions that can be called inside OpenRSP.

The OpenRSP context `OpenRSPStruct` will encapsulate all the implemented C `struct`'s of the "OpenRSP C support" components into another C `struct`:

14b ⟨*OpenRSPStruct* 14b⟩≡

```
typedef struct {
    QBool assembled;               /* indicates if the OpenRSP context assembled */
    RSPPert *rsp_pert;             /* host program perturbations */
    /*ElecWav *elec_wav;*/           /* implementation-specific data of (electronic) wave f
    /*ElecWavType elec_wav_type;*/
    RSPOverlap *overlap;           /* overlap integrals */
    RSPOneOper *one_oper;          /* one-electron operators */
    RSPTwoOper *two_oper;          /* two-electron operators */
    RSPXCFun *xc_fun;              /* XC functionals */
    RSPNucHamilton *nuc_hamilton;  /* nuclear Hamiltonian */
    RSPSolver *rsp_solver;         /* linear response equation solver */
} OpenRSP;
```

where we have used types, macros and APIs implemented in the QcMatrix library and one should be familiar with them first.

Users should use the OpenRSP context and the following APIs to access the functionalities of OpenRSP:

14c ⟨*OpenRSPAPIs* 14c⟩≡

```
extern QErrorCode OpenRSPCreate(OpenRSP*);
extern QErrorCode OpenRSPSetPerturbations(OpenRSP*,
                                          const QInt,
```

```
                                          const QcPertInt*,
                                          const QInt*,
                                          const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                          void*,
#endif
                                          const GetPertCat);
/*extern QErrorCode OpenRSPSetWaveFunction(OpenRSP*,const ElecWavType);*/
extern QErrorCode OpenRSPSetOverlap(OpenRSP*,
                                    const QInt,
                                    const QcPertInt*,
                                    const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                    void*,
#endif
                                    const GetOverlapMat,
                                    const GetOverlapExp);
extern QErrorCode OpenRSPAddOneOper(OpenRSP*,
                                    const QInt,
                                    const QcPertInt*,
                                    const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                    void*,
#endif
                                    const GetOneOperMat,
                                    const GetOneOperExp);
extern QErrorCode OpenRSPAddTwoOper(OpenRSP*,
                                    const QInt,
                                    const QcPertInt*,
                                    const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                    void*,
#endif
                                    const GetTwoOperMat,
                                    const GetTwoOperExp);
extern QErrorCode OpenRSPAddXCFun(OpenRSP*,
                                  const QInt,
                                  const QcPertInt*,
                                  const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                  void*,
#endif
                                  const GetXCFunMat,
                                  const GetXCFunExp);
extern QErrorCode OpenRSPSetNucHamilton(OpenRSP*,
                                        const QInt,
                                        const QcPertInt*,
                                        const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                        void*,
#endif
                                        const GetNucContrib,
```

```
  /*FIXME: num_atoms to be removed after perturbation free scheme implemented*/
                                          const QInt);
  extern QErrorCode OpenRSPSetLinearRSPSolver(OpenRSP*,
  #if defined(OPENRSP_C_USER_CONTEXT)
                                              void*,
  #endif
                                              const GetLinearRSPSolution);
  extern QErrorCode OpenRSPAssemble(OpenRSP*);
  extern QErrorCode OpenRSPWrite(const OpenRSP*,const QChar*);
  extern QErrorCode OpenRSPGetRSPFun(OpenRSP*,
                                     const QcMat*,
                                     const QcMat*,
                                     const QcMat*,
                                     const QInt,
                                     const QInt*,
                                     const QcPertInt*,
                                     const QInt*,
                                     const QReal*,
                                     const QInt*,
                                     const QInt,
                                     QReal*);
  extern QErrorCode OpenRSPGetResidue(OpenRSP*,
                                      const QcMat*,
                                      const QcMat*,
                                      const QcMat*,
                                      const QInt,
                                      const QInt,
                                      const QReal*,
                                      QcMat*[],
                                      const QInt,
                                      const QInt*,
                                      const QcPertInt*,
                                      const QInt*,
                                      const QInt*,
                                      const QInt*,
                                      const QReal*,
                                      const QInt*,
                                      const QInt,
                                      QReal*);
  extern QErrorCode OpenRSPDestroy(OpenRSP*);
```

Here, we have also introduced the type of electronic wave function, but which has not been implemented.

Last but not least, the directive

```
#if defined(OPENRSP_C_USER_CONTEXT)
                                            void*,
#endif
```

in most OpenRSP APIs enables users to provide their necessary setting for the callback functions that OpenRSP will send it back when invoking the callback functions. For instance, users can provide the information of basis sets to OpenRSP and use it inside the callback functions for different integral calculations.

3.2 Four Basic APIs for the OpenRSP Context

In this section, we will implement four basic APIs `OpenRSPCreate()`, `OpenRSPAssemble()`, `OpenRSPWrite()` and `OpenRSPDestroy()`, while other APIs will be implemented in the following sections. These four APIs respectively create, assemble, write and destroy the OpenRSP context.

The API `OpenRSPCreate()` is very simple as it only initializes the pointers of the context:

17a ⟨*OpenRSP.c* 17a⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩
*/

#include "OpenRSP.h"

/* <function name='OpenRSPCreate' author='Bin Gao' date='2014-01-28'>
     Creates the OpenRSP context
     <param name='open_rsp' direction='inout'>The OpenRSP context</param>
     <return>Error information</return>
   </function> */
QErrorCode OpenRSPCreate(OpenRSP *open_rsp)
{
    open_rsp->assembled = QFALSE;
    open_rsp->rsp_pert = NULL;
    /*open_rsp->elec_wav = NULL;*/
    /*open_rsp->elec_wav_type = ELEC_AO_D_MATRIX;*/
    open_rsp->overlap = NULL;
    open_rsp->one_oper = NULL;
    open_rsp->two_oper = NULL;
    open_rsp->xc_fun = NULL;
    open_rsp->nuc_hamilton = NULL;
    open_rsp->rsp_solver = NULL;
    return QSUCCESS;
}
```

The other three APIs are also easy to implement, as they only invoke functions of the "OpenRSP C support" part to respectively assemble, write and destroy the corresponding C `struct`'s:

17b ⟨*OpenRSP.c* 17a⟩+≡

```
/* <function name='OpenRSPAssemble' author='Bin Gao' date='2014-07-30'>
     Assembles the OpenRSP context
     <param name='open_rsp' direction='inout'>The OpenRSP context</param>
     <return>Error information</return>
   </function> */
QErrorCode OpenRSPAssemble(OpenRSP *open_rsp)
{
    QErrorCode ierr;  /* error information */
    open_rsp->assembled = QFALSE;
    /* assembles host program perturbations */
    if (open_rsp->rsp_pert!=NULL) {
        ierr = RSPPertAssemble(open_rsp->rsp_pert);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertAssemble()");
    }
    else {
        QErrorExit(FILE_AND_LINE, "perturbations not set by OpenRSPSetPerturbations()");
```

```
    }
/*FIXME: to implement ierr = xxAssemble(open_rsp->elec_eom); */
    /* assembles overlap integrals */
    if (open_rsp->overlap!=NULL) {
        ierr = RSPOverlapAssemble(open_rsp->overlap,
                                  open_rsp->rsp_pert);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPOverlapAssemble()");
    }
    /* assembles one-electron operators */
    if (open_rsp->one_oper!=NULL) {
        ierr = RSPOneOperAssemble(open_rsp->one_oper,
                                  open_rsp->rsp_pert);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPOneOperAssemble()");
    }
    /* assembles two-electron operators */
    if (open_rsp->two_oper!=NULL) {
        ierr = RSPTwoOperAssemble(open_rsp->two_oper,
                                  open_rsp->rsp_pert);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPTwoOperAssemble()");
    }
    /* assembles XC functionals */
    if (open_rsp->xc_fun!=NULL) {
        ierr = RSPXCFunAssemble(open_rsp->xc_fun,
                                open_rsp->rsp_pert);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPXCFunAssemble()");
    }
    /* assembles nuclear Hamiltonian */
    if (open_rsp->nuc_hamilton!=NULL) {
        ierr = RSPNucHamiltonAssemble(open_rsp->nuc_hamilton,
                                      open_rsp->rsp_pert);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPNucHamiltonAssemble()");
    }
    /* assembles linear response equation solver */
    if (open_rsp->rsp_solver!=NULL) {
        ierr = RSPSolverAssemble(open_rsp->rsp_solver);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPSolverAssemble()");
    }
    else {
        QErrorExit(FILE_AND_LINE, "solver not set by OpenRSPSetSolver()");
    }
    open_rsp->assembled = QTRUE;
    return QSUCCESS;
}

/* <function name='OpenRSPWrite' author='Bin Gao' date='2014-07-30'>
     Writes the OpenRSP context
     <param name='open_rsp' direction='in'>The OpenRSP context</param>
     <param name='file_name' direction='in'>File to write the context</param>
     <return>Error information</return>
   </function> */
QErrorCode OpenRSPWrite(const OpenRSP *open_rsp, const QChar *file_name)
{
```

```
    FILE *fp_rsp;     /* file pointer */
    QErrorCode ierr;  /* error information */
    /* opens the file */
    fp_rsp = fopen(file_name, "a");
    if (fp_rsp==NULL) {
        printf("OpenRSPWrite>> file: %s\n", file_name);
        QErrorExit(FILE_AND_LINE, "failed to open the file in appending mode");
    }
    fprintf(fp_rsp, "\nOpenRSP library compiled at %s, %s\n", __TIME__, __DATE__);
    /* context of the (electronic) wave function */
    /*FIXME: ierr = xxWrite(open_rsp->elec_eom); */
    if (open_rsp->rsp_pert!=NULL) {
        ierr = RSPPertWrite(open_rsp->rsp_pert, fp_rsp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertWrite()");
    }
    if (open_rsp->overlap!=NULL) {
        fprintf(fp_rsp, "OpenRSPWrite>> overlap integrals\n");
        ierr = RSPOverlapWrite(open_rsp->overlap, fp_rsp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPOverlapWrite()");
    }
    if (open_rsp->one_oper!=NULL) {
        fprintf(fp_rsp, "OpenRSPWrite>> linked list of one-electron operators\n");
        ierr = RSPOneOperWrite(open_rsp->one_oper, fp_rsp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPOneOperWrite()");
    }
    if (open_rsp->two_oper!=NULL) {
        fprintf(fp_rsp, "OpenRSPWrite>> linked list of two-electron operators\n");
        ierr = RSPTwoOperWrite(open_rsp->two_oper, fp_rsp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPTwoOperWrite()");
    }
    if (open_rsp->xc_fun!=NULL) {
        fprintf(fp_rsp, "OpenRSPWrite>> linked list of XC functionals\n");
        ierr = RSPXCFunWrite(open_rsp->xc_fun, fp_rsp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPXCFunWrite()");
    }
    if (open_rsp->nuc_hamilton!=NULL) {
        fprintf(fp_rsp, "OpenRSPWrite>> nuclear Hamiltonian\n");
        ierr = RSPNucHamiltonWrite(open_rsp->nuc_hamilton, fp_rsp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPNucHamiltonWrite()");
    }
    if (open_rsp->rsp_solver!=NULL) {
        ierr = RSPSolverWrite(open_rsp->rsp_solver, fp_rsp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPSolverWrite()");
    }
    /* closes the file */
    fclose(fp_rsp);
    return QSUCCESS;
}

/* <function name='OpenRSPDestroy' author='Bin Gao' date='2014-01-28'>
     Destroys the OpenRSP context
     <param name='open_rsp' direction='inout'>The OpenRSP context</param>
```

```
     <return>Error information</return>
   </function> */
QErrorCode OpenRSPDestroy(OpenRSP *open_rsp)
{
    QErrorCode ierr;  /* error information */
    open_rsp->assembled = QFALSE;
//    if (open_rsp->elec_eom!=NULL) {
///*FIXME: to implement ierr = xxDestroy(open_rsp->elec_eom); */
//        free(open_rsp->elec_eom);
//        open_rsp->elec_eom = NULL;
//    }
    /* destroys the context of all perturbations involved in calculations */
    if (open_rsp->rsp_pert!=NULL) {
        ierr = RSPPertDestroy(open_rsp->rsp_pert);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertDestroy()");
        free(open_rsp->rsp_pert);
        open_rsp->rsp_pert = NULL;
    }
    /* destroys the context of overlap integrals */
    if (open_rsp->overlap!=NULL) {
        ierr = RSPOverlapDestroy(open_rsp->overlap);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPOverlapDestroy()");
        free(open_rsp->overlap);
        open_rsp->overlap = NULL;
    }
    /* destroys the linked list of one-electron operators */
    if (open_rsp->one_oper!=NULL) {
        ierr = RSPOneOperDestroy(&open_rsp->one_oper);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPOneOperDestroy()");
    }
    /* destroys the linked list of two-electron operators */
    if (open_rsp->two_oper!=NULL) {
        ierr = RSPTwoOperDestroy(&open_rsp->two_oper);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPTwoOperDestroy()");
    }
    /* destroys the linked list of exchange-correlation functionals */
    if (open_rsp->xc_fun!=NULL) {
        ierr = RSPXCFunDestroy(&open_rsp->xc_fun);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPXCFunDestroy()");
    }
    /* destroys the context of nuclear Hamiltonian */
    if (open_rsp->nuc_hamilton!=NULL) {
        ierr = RSPNucHamiltonDestroy(open_rsp->nuc_hamilton);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPNucHamiltonDestroy()");
        free(open_rsp->nuc_hamilton);
        open_rsp->nuc_hamilton = NULL;
    }
    /* destroys the context of linear response equation sovler */
    if (open_rsp->rsp_solver!=NULL) {
        ierr = RSPSolverDestroy(open_rsp->rsp_solver);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPSolverDestroy()");
        free(open_rsp->rsp_solver);
```

```
        open_rsp->rsp_solver = NULL;
    }
    return QSUCCESS;
}
```

3.3 Perturbations

The header file of perturbations is organized as:

22a ⟨*RSPPerturbation.h* 22a⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

  <header name='RSPPerturbation.h' author='Bin Gao' date='2015-06-23'>
    The header file of perturbations used inside OpenRSP
  </header>
*/

#if !defined(RSP_PERTURBATION_H)
#define RSP_PERTURBATION_H

/* QcMatrix library */
#include "qcmatrix.h"

⟨RSPPertBasicTypes 22b⟩

⟨RSPPertCallback 24a⟩

⟨RSPPertStruct 24b⟩

⟨RSPertAPIs 26a⟩

#endif
```

3.3.1 Internal Perturbation Labels

Perturbation labels are represented by integers. Each different label is marked by an internal identifier for calculating reponse functions and residues, so OpenRSP can sort these labels in ascending order by their identifiers. For instance, identifiers for a perturbation tuple *acbadbc* are 0120321.

To avoid using extra arrays for identifiers, we combine each host programs' label with its identifier into a `QcPertInt` type integer:

$$\texttt{QcPertInt} = \text{identifier}\texttt{<<OPENRSP\_PERT\_LABEL\_BIT} + \text{label},$$

where `OPENRSP_PERT_LABEL_BIT` is the number of bits in an object of `QcPertInt` for representing the host programs' perturbation labels:

22b ⟨*RSPPertBasicTypes* 22b⟩≡

```
/* <macrodef name='OPENRSP_PERT_LABEL_BIT'>
     Set <OPENRSP_PERT_LABEL_BIT>
   </macrodef>
   <constant name='OPENRSP_PERT_LABEL_BIT'>
     Number of bits in an object of <QcPertInt> type for a perturbation label
   </constant> */
#if !defined(OPENRSP_PERT_LABEL_BIT)
#define OPENRSP_PERT_LABEL_BIT 10
#endif
```

The type `QcPertInt` is defined as:

23a ⟨*RSPPertBasicTypes* 22b⟩+≡

```
/* <datatype name='QcPertInt'>
     Data type of integers to represent perturbation labels
   </datatype>
   <constant name='QCPERTINT_MAX'>
     Maximal value of an object of the <QcPertInt> type
   </constant>
   <constant name='QCPERTINT_FMT'>
     Format string of <QcPertInt> type
   </constant> */
//typedef unsigned long QcPertInt;
//#define QCPERTINT_MAX ULONG_MAX
//#define QCPERTINT_FMT "lu"
typedef unsigned int QcPertInt;
#define QCPERTINT_MAX UINT_MAX
#define QCPERTINT_FMT "u"
```

Here we also define a constant `QCPERTINT_MAX` for the maximal value of an object of the `QcPertInt` type, and a format string (`QCPERTINT_FMT`) of the `QcPertInt` type.

From `OPENRSP_PERT_LABEL_BIT` and `QCPERTINT_MAX` we can compute `OPENRSP_PERT_LABEL_MAX` and `OPENRSP_PERT_ID_MAX`, which are the maximal values of host programs' perturbation labels and internal perturbation identifiers:

23b ⟨*RSPPertBasicTypes* 22b⟩+≡

```
/* <constant name='OPENRSP_PERT_LABEL_MAX'>
     Maximal value for perturbation labels
   </constant>
   <constant name='OPENRSP_PERT_ID_MAX'>
     Maximal value for internal perturbation identifier
   </constant> */
extern const QcPertInt OPENRSP_PERT_LABEL_MAX;
extern const QcPertInt OPENRSP_PERT_ID_MAX;
```

Here, to avoid multiple inclusions of the header file that will lead to multiple definitions, we have the following implementation file for the `OPENRSP_PERT_LABEL_MAX` and `OPENRSP_PERT_ID_MAX`:

23c ⟨*RSPPerturbation.c* 23c⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩
*/

#include "RSPPerturbation.h"

const QcPertInt OPENRSP_PERT_LABEL_MAX = (1<<OPENRSP_PERT_LABEL_BIT)-1;
const QcPertInt OPENRSP_PERT_ID_MAX =
                (QCPERTINT_MAX-OPENRSP_PERT_LABEL_MAX)>>OPENRSP_PERT_LABEL_BIT;

/* see https://scaryreasoner.wordpress.com/2009/02/28/checking-sizeof-at-compile-time
   accessing date Oct. 6, 2015 */
#define QC_BUILD_BUG_ON(condition) ((void)sizeof(char[1 - 2*!!(condition)]))
void RSPPertCheckLabelBit()
{
    QC_BUILD_BUG_ON(sizeof(QcPertInt)*CHAR_BIT<=OPENRSP_PERT_LABEL_BIT);
    QC_BUILD_BUG_ON(QINT_MAX<OPENRSP_PERT_LABEL_MAX);
}
```

The function `RSPPertCheckLabelBit()` ensures that (i) `OPENRSP_PERT_LABEL_BIT` is not too large and there are still bits left for the identifiers, and (ii) the number of different perturbation labels is not greater than the maximal value of an object of the `QInt` type so that we can still use `QInt` type integers for the number of (different) perturbation labels and the length of perturbation tuples. Note that the number of internal identifiers is less than or equal to the number of different perturbation labels involved in calculations, so it automatically satisfies the condition (ii).

One will have building error when compiling the function `RSPPertCheckLabelBit()` if the constant `OPENRSP_PERT_LABEL_BIT` is too large. However, the function `RSPPertCheckLabelBit()` can not guarantee the above setting (`QcPertInt` type and `OPENRSP_PERT_LABEL_BIT`) is enough for holding the host programs' perturbation labels and the internal identifiers. This will be checked against `OPENRSP_PERT_LABEL_MAX` and `OPENRSP_PERT_ID_MAX` by OpenRSP when (i) setting the host programs' perturbations, and (ii) calculating response functions or residues.

3.3.2 Callback Functions for Perturbations

As discussed in Section 2.5.6, we will need a callback function `get_pert_concatenation()` to get the ranks of components of sub-perturbation tuples (with the same perturbation label) for given components of the corresponding concatenated perturbation tuple. The type of this callback function is defined as follows:

24a ⟨*RSPPertCallback* 24a⟩≡

```
typedef void (*GetPertCat)(const QInt,
                           const QcPertInt,
                           const QInt,
                           const QInt,
                           const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                           void*,
#endif
                           QInt*);
```

We will discuss how to use this callback function in Section 3.3.4.

3.3.3 Perturbation Context and Its Basic APIs

Also as mentioned in Section 2.5.6, OpenRSP needs to know the numbers of components of host programs' perturbations. Because the number of components of a higher-order perturbation is simply the product of numbers of components of lower-order perturbations with different labels. For instance, the number of components of a perturbation $a^{n_a}_{\omega_a} b^{n_b}_{\omega_b}$ is simply the product of numbers of components of perturbations $a^{n_a}_{\omega_a}$ and $b^{n_b}_{\omega_b}$.

Therefore, if there are in total p different perturbation labels $a_1, a_2, \cdots, a_p$ involved in calculations, OpenRSP needs to know

1. allowed maximal order of a perturbation described by exactly **one** of these different labels $a_1, a_2, \cdots, a_p$; let us mark these allowed maximal orders as $n_1, n_2, \cdots, n_p$, which means we will have $k_j \leq n_j$ $(1 \leq j \leq p)$ for any perturbation $a^{k_1}_{1,\omega_1} a^{k_2}_{2,\omega_2} \cdots a^{k_p}_{p,\omega_p}$;
2. numbers of components of perturbations $a^{k_j}_{j,\omega_j}$, where $1 \leq k_j \leq n_j$ and $1 \leq j \leq p$; let us mark these numbers of components as $[N^{k_j}_j]$.

The above information is saved into the following `struct`:

24b ⟨*RSPPertStruct* 24b⟩≡

```
typedef struct {
    QInt num_pert_lab;                  /* number of different perturbation
                                           labels $p$ */
    QInt *pert_max_orders;              /* $n_{1},n_{2},\cdots,n_{p}$ */
    QInt *ptr_ncomp;                    /* pointers to $[N_{j}^{k_{j}}]$
```

```
                                        for each $a_{j}$ */
    QInt *pert_num_comps;               /* $[N_{j}^{k_{j}}]$, where
                                        $1\le k_{j}\le n_{j}$ and $1\le j\le p$ */
    QcPertInt *pert_labels;             /* $a_{1},a_{2},\cdots,a_{p}$ */
#if defined(OPENRSP_C_USER_CONTEXT)
    void *user_ctx;                     /* user-defined callback function context */
#endif
    GetPertCat get_pert_concatenation;  /* user-specified function for getting
                                        the ranks of components of sub-perturbation
                                        tuples (with the same perturbation label)
                                        for given components of the corresponding
                                        concatenated perturbation tuple */
} RSPPert;
```

and users can set the above information by the following API:

25 ⟨*OpenRSP.c* 17a⟩+≡

```
/* <function name='OpenRSPSetPerturbations' author='Bin Gao' date='2015-06-29'>
     Sets all perturbations involved in response theory calculations
     <param name='open_rsp' direction='inout'>The OpenRSP context</param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels involved in calculations
     </param>
     <param name='pert_labels' direction='in'>
       All the different perturbation labels involved
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different labels
     </param>
     <param name='pert_num_comps' direction='in'>
       Number of components of a perturbation described by exactly one of
       the above different labels, up to the allowed maximal order, size
       is therefore the sum of <pert_max_orders>
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback function context
     </param>
     <param name='get_pert_concatenation' direction='in'>
       User specified function for getting the ranks of components of
       sub-perturbation tuples (with the same perturbation label) for given
       components of the corresponding concatenated perturbation tuple
     </param>
     <return>Error information</return>
   </function> */
QErrorCode OpenRSPSetPerturbations(OpenRSP *open_rsp,
                                   const QInt num_pert_lab,
                                   const QcPertInt *pert_labels,
                                   const QInt *pert_max_orders,
                                   const QInt *pert_num_comps,
#if defined(OPENRSP_C_USER_CONTEXT)
                                   void *user_ctx,
#endif
                                   const GetPertCat get_pert_concatenation)
```

```
{
    QErrorCode ierr;  /* error information */
    /* creates the context of all perturbations involved in calculations */
    if (open_rsp->rsp_pert!=NULL) {
        ierr = RSPPertDestroy(open_rsp->rsp_pert);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertDestroy()");
    }
    else {
        open_rsp->rsp_pert = (RSPPert *)malloc(sizeof(RSPPert));
        if (open_rsp->rsp_pert==NULL) {
            QErrorExit(FILE_AND_LINE, "allocates memory for perturbations");
        }
    }
    ierr = RSPPertCreate(open_rsp->rsp_pert,
                         num_pert_lab,
                         pert_labels,
                         pert_max_orders,
                         pert_num_comps,
#if defined(OPENRSP_C_USER_CONTEXT)
                         user_ctx,
#endif
                         get_pert_concatenation);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertCreate()");
    return QSUCCESS;
}
```

The APIs `RSPPertCreate()` and `RSPPertDestroy()` will respectively create and destroy the content of the `struct RSPPert`. We also need APIs to assemble and to write the `struct RSPPert`:

26a ⟨*RSPertAPIs* 26a⟩≡

```
extern QErrorCode RSPPertCreate(RSPPert*,
                                const QInt,
                                const QcPertInt*,
                                const QInt*,
                                const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                void*,
#endif
                                const GetPertCat);
extern QErrorCode RSPPertAssemble(RSPPert*);
extern QErrorCode RSPPertWrite(const RSPPert*,FILE*);
extern QErrorCode RSPPertDestroy(RSPPert*);
```

which are respectively implemented as follows:

26b ⟨*RSPPerturbation.c* 23c⟩+≡

```
/* <function name='RSPPertCreate'
             attr='private'
             author='Bin Gao'
             date='2015-06-28'>
     Sets all perturbations involved in response theory calculations, should be
     called at first
     <param name='rsp_pert' direction='inout'>
       The context of perturbations
     </param>
```

```
      <param name='num_pert_lab' direction='in'>
        Number of all different perturbation labels involved in calculations
      </param>
      <param name='pert_labels' direction='in'>
        All the different perturbation labels involved
      </param>
      <param name='pert_max_orders' direction='in'>
        Allowed maximal order of a perturbation described by exactly one of
        the above different labels
      </param>
      <param name='pert_num_comps' direction='in'>
        Number of components of a perturbation described by exactly one of
        the above different labels, up to the allowed maximal order, size
        is therefore the sum of <pert_max_orders>
      </param>
      <param name='user_ctx' direction='in'>
        User-defined callback function context
      </param>
      <param name='get_pert_concatenation' direction='in'>
        User-specified function for getting the ranks of components of
        sub-perturbation tuples (with the same perturbation label) for given
        components of the corresponding concatenated perturbation tuple
      </param>
      <return>Error information</return>
    </function> */
QErrorCode RSPPertCreate(RSPPert *rsp_pert,
                         const QInt num_pert_lab,
                         const QcPertInt *pert_labels,
                         const QInt *pert_max_orders,
                         const QInt *pert_num_comps,
#if defined(OPENRSP_C_USER_CONTEXT)
                         void *user_ctx,
#endif
                         const GetPertCat get_pert_concatenation)
{
    QInt ilab;    /* incremental recorders over perturbation labels */
    QInt jlab;
    QInt iorder;  /* incremental recorder over orders */
    QInt icomp;   /* incremental recorder over components */
    if (num_pert_lab<1) {
        printf("RSPPertCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        QErrorExit(FILE_AND_LINE, "invalid number of perturbation labels");
    }
    else if (num_pert_lab>OPENRSP_PERT_LABEL_MAX) {
        printf("RSPPertCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        printf("RSPPertCreate>> maximal value for pert. labels %"QCPERTINT_FMT"\n",
               OPENRSP_PERT_LABEL_MAX);
        QErrorExit(FILE_AND_LINE, "too many perturbation labels");
    }
    rsp_pert->num_pert_lab = num_pert_lab;
```

```
rsp_pert->pert_labels = (QcPertInt *)malloc(num_pert_lab*sizeof(QcPertInt));
if (rsp_pert->pert_labels==NULL) {
    printf("RSPPertCreate>> number of perturbation labels %"QINT_FMT"\n",
           num_pert_lab);
    QErrorExit(FILE_AND_LINE, "allocates memory for perturbation labels");
}
rsp_pert->pert_max_orders = (QInt *)malloc(num_pert_lab*sizeof(QInt));
if (rsp_pert->pert_max_orders==NULL) {
    printf("RSPPertCreate>> number of perturbation labels %"QINT_FMT"\n",
           num_pert_lab);
    QErrorExit(FILE_AND_LINE, "allocates memory for allowed maximal orders");
}
rsp_pert->ptr_ncomp = (QInt *)malloc((num_pert_lab+1)*sizeof(QInt));
if (rsp_pert->ptr_ncomp==NULL) {
    printf("RSPPertCreate>> number of perturbation labels %"QINT_FMT"\n",
           num_pert_lab);
    QErrorExit(FILE_AND_LINE, "allocates memory for pointers to components");
}
rsp_pert->ptr_ncomp[0] = 0;
for (ilab=0; ilab<num_pert_lab; ilab++) {
    if (pert_labels[ilab]>OPENRSP_PERT_LABEL_MAX) {
        printf("RSPPertCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
               ilab,
               pert_labels[ilab]);
        printf("RSPPertCreate>> maximal value for pert. labels %"QCPERTINT_FMT"\n",
               OPENRSP_PERT_LABEL_MAX);
        QErrorExit(FILE_AND_LINE, "invalid perturbation label");
    }
    /* each element of <pert_labels> should be unique */
    for (jlab=0; jlab<ilab; jlab++) {
        if (pert_labels[jlab]==pert_labels[ilab]) {
            printf("RSPPertCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
                   jlab,
                   pert_labels[jlab]);
            printf("RSPPertCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
                   ilab,
                   pert_labels[ilab]);
            QErrorExit(FILE_AND_LINE, "repeated perturbation labels not allowed");
        }
    }
    rsp_pert->pert_labels[ilab] = pert_labels[ilab];
    if (pert_max_orders[ilab]<1) {
        printf("RSPPertCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
               ilab,
               pert_labels[ilab]);
        printf("RSPPertCreate>> allowed maximal order is %"QINT_FMT"\n",
               pert_max_orders[ilab]);
        QErrorExit(FILE_AND_LINE, "only positive order allowed");
    }
    rsp_pert->pert_max_orders[ilab] = pert_max_orders[ilab];
    /* <c>rsp_pert->ptr_ncomp[ilab]</c> points to the number of components
       of <c>rsp_pert->pert_labels[ilab]</c> */
```

```
        rsp_pert->ptr_ncomp[ilab+1] = rsp_pert->ptr_ncomp[ilab]+pert_max_orders[ilab];
    }
    /* <c>rsp_pert->ptr_ncomp[num_pert_lab]</c> equals to the size of
       <c>rsp_pert->pert_num_comps</c> */
    rsp_pert->pert_num_comps = (QInt *)malloc(rsp_pert->ptr_ncomp[num_pert_lab]
                                              *sizeof(QInt));
    if (rsp_pert->pert_num_comps==NULL) {
        printf("RSPPertCreate>> size of numbers of components %"QINT_FMT"\n",
               rsp_pert->ptr_ncomp[num_pert_lab]);
        QErrorExit(FILE_AND_LINE, "allocates memory for numbers of components");
    }
    for (ilab=0,icomp=0; ilab<num_pert_lab; ilab++) {
        for (iorder=1; iorder<=rsp_pert->pert_max_orders[ilab]; iorder++,icomp++) {
            if (pert_num_comps[icomp]<1) {
                printf("RSPPertCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
                       ilab,
                       pert_labels[ilab]);
                printf("RSPPertCreate>> allowed maximal order is %"QINT_FMT"\n",
                       pert_max_orders[ilab]);
                printf("RSPPertCreate>> %"QINT_FMT"-th No. of comps. is %"QINT_FMT"\n",
                       iorder,
                       pert_num_comps[icomp]);
                QErrorExit(FILE_AND_LINE, "incorrect number of components");
            }
            rsp_pert->pert_num_comps[icomp] = pert_num_comps[icomp];
        }
    }
#if defined(OPENRSP_C_USER_CONTEXT)
    rsp_pert->user_ctx = user_ctx;
#endif
    rsp_pert->get_pert_concatenation = get_pert_concatenation;
    return QSUCCESS;
}
```

Here we check the number of perturbation labels and each label against `OPENRSP_PERT_LABEL_MAX`.

29 ⟨*RSPPerturbation.c* 23c⟩+≡

```
/* <function name='RSPPertAssemble'
             attr='private'
             author='Bin Gao'
             date='2015-06-28'>
     Assembles the context of perturbations involved in calculations
     <param name='rsp_pert' direction='inout'>
       The context of perturbations
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPPertAssemble(RSPPert *rsp_pert)
{
    if (rsp_pert->pert_labels==NULL ||
        rsp_pert->pert_max_orders==NULL ||
        rsp_pert->ptr_ncomp==NULL ||
        rsp_pert->pert_num_comps==NULL ||
```

```
        rsp_pert->get_pert_concatenation==NULL) {
        QErrorExit(FILE_AND_LINE, "perturbations are not correctly set");
    }
    return QSUCCESS;
}

/* <function name='RSPPertWrite'
             attr='private'
             author='Bin Gao'
             date='2015-06-28'>
     Writes the context of perturbations involved in calculations
     <param name='rsp_pert' direction='in'>
       The context of perturbations
     </param>
     <param name='fp_pert' direction='inout'>File pointer</param>
     <return>Error information</return>
   </function> */
QErrorCode RSPPertWrite(const RSPPert *rsp_pert, FILE *fp_pert)
{
    QInt ilab;   /* incremental recorder over perturbation labels */
    QInt icomp;  /* incremental recorder over components */
    fprintf(fp_pert,
            "RSPPertWrite>> number of all perturbation labels %"QINT_FMT"\n",
            rsp_pert->num_pert_lab);
    fprintf(fp_pert,
            "RSPPertWrite>> label           maximum-order    numbers-of-components\n");
    for (ilab=0; ilab<rsp_pert->num_pert_lab; ilab++) {
        fprintf(fp_pert,
                "RSPPertWrite>>  %"QCPERTINT_FMT"               %"QINT_FMT"               "
                rsp_pert->pert_labels[ilab],
                rsp_pert->pert_max_orders[ilab]);
        for (icomp=rsp_pert->ptr_ncomp[ilab]; icomp<rsp_pert->ptr_ncomp[ilab+1]; icomp++) {
            fprintf(fp_pert, " %"QINT_FMT"", rsp_pert->pert_num_comps[icomp]);
        }
        fprintf(fp_pert, "\n");
    }
#if defined(OPENRSP_C_USER_CONTEXT)
    if (rsp_pert->user_ctx!=NULL) {
        fprintf(fp_pert, "RSPPertWrite>> user-defined function context given\n");
    }
#endif
    return QSUCCESS;
}

/* <function name='RSPPertDestroy'
             attr='private'
             author='Bin Gao'
             date='2015-06-28'>
     Destroys the context of perturbations involved in calculations, should be
     called at the end
     <param name='rsp_pert' direction='inout'>
       The context of perturbations
```

```
      </param>
      <return>Error information</return>
    </function> */
QErrorCode RSPPertDestroy(RSPPert *rsp_pert)
{
    rsp_pert->num_pert_lab = 0;
    free(rsp_pert->pert_labels);
    rsp_pert->pert_labels = NULL;
    free(rsp_pert->pert_max_orders);
    rsp_pert->pert_max_orders = NULL;
    free(rsp_pert->ptr_ncomp);
    rsp_pert->ptr_ncomp = NULL;
    free(rsp_pert->pert_num_comps);
    rsp_pert->pert_num_comps = NULL;
#if defined(OPENRSP_C_USER_CONTEXT)
    rsp_pert->user_ctx = NULL;
#endif
    rsp_pert->get_pert_concatenation = NULL;
    return QSUCCESS;
}
```

When users set different contributions to the Hamiltonian, there are also perturbations acting on the corresponding contributions. We need to check if these perturbations are valid:

31a ⟨*RSPertAPIs* 26a⟩+≡

```
extern QErrorCode RSPPertValidateLabelOrder(const RSPPert*,
                                            const QInt,
                                            const QcPertInt*,
                                            const QInt*);
```

31b ⟨*RSPPerturbation.c* 23c⟩+≡

```
/* <function name='RSPPertValidateLabelOrder'
             attr='private'
             author='Bin Gao'
             date='2015-10-15'>
     Check the validity of given perturbation labels and corresponding allowed
     maximal orders
     <param name='rsp_pert' direction='in'>
       The context of perturbations
     </param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels
     </param>
     <param name='pert_labels' direction='in'>
       Different perturbation labels that will be checked
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different labels
     </param>
     <return>
       <QSUCCESS> if perturbation labels and orders are valid,
       <QFAILURE> otherwise
     </return>
```

```
    </function> */
QErrorCode RSPPertValidateLabelOrder(const RSPPert *rsp_pert,
                                     const QInt num_pert_lab,
                                     const QcPertInt *pert_labels,
                                     const QInt *pert_max_orders)
{
    QInt ilab;         /* incremental recorders over perturbation labels */
    QInt jlab;
    QBool pert_valid;  /* validity of the perturbations */
    for (ilab=0; ilab<num_pert_lab; ilab++) {
        pert_valid = QFALSE;
        for (jlab=0; jlab<rsp_pert->num_pert_lab; jlab++) {
            /* valid perturbation label, checks its allowed maximal order */
            if (pert_labels[ilab]==rsp_pert->pert_labels[jlab]) {
                if (pert_max_orders[ilab]<=rsp_pert->pert_max_orders[jlab]) {
                    pert_valid = QTRUE;
                }
                break;
            }
        }
        if (pert_valid==QFALSE) return QFAILURE;
    }
    return QSUCCESS;
}
```

One should note that we here assume all the elements of `pert_labels` are different.

3.3.4 Conversion of Perturbation Tuples and Labels

As discussed in Section 3.3.1, we will use internal perturbation labels (identifier and host program's perturbation label) inside OpenRSP . As such, we need to implement functions for the conversion of internal perturbation labels and host programs' ones.

First, when calculating response functions and residues, we need to convert the users' given perturbation tuple $abc\cdots$ into our internal one. We also need to sort the perturbation tuple $abc\cdots$ and the corresponding complex frequencies $[\omega_a\omega_b\omega_c\cdots,\cdots]$. The sorting is carried out in two steps:

1. We take the first label of the perturbation tuple as the perturbation a (see definition in Section 2.5.6). We fix the first label of the tuple, and rearrange the tuple by collecting identical labels together after their first entity.
   For example, the sorted perturbation tuple of *acbadbc* becomes *aaccbbd*.
2. We next sort frequencies of the same perturbation labels in ascending order, and the frequency of the first label of the tuple is the negative sum of frequencies of other labels.
   For example, if users give two frequency configurations for a perturbation tuple *aaccbbd*, the sorted frequencies will satisfy:

$$-\sum_i \omega_i \le \omega_1 \le \omega_2 \le \omega_3 \le \omega_4 \le \omega_5 \le \omega_6,$$
$$-\sum_i \omega'_i \le \omega'_1 \le \omega'_2 \le \omega'_3 \le \omega'_4 \le \omega'_5 \le \omega'_6.$$

The following function will perform the above converting and sorting procedure, and also check if the perturbation labels have been given by the API `OpenRSPSetPerturbations()`:

32 ⟨*RSPertAPIs* 26a⟩+≡

```
extern QErrorCode RSPPertHostToInternal(const RSPPert*,
                                        const QInt,
                                        QcPertInt*,
                                        const QInt,
                                        QReal*);
```

33 ⟨*RSPPerturbation.c* 23c⟩+≡

```
/* <function name='RSPPertHostToInternal'
             attr='private'
             author='Bin Gao'
             date='2015-10-08'>
     Check, convert and sort a host program's perturbation tuple and
     corresponding frequencies
     <param name='rsp_pert' direction='in'>
       The context of perturbations
     </param>
     <param name='len_tuple' direction='in'>
       Length of the host program's and the internal perturbation tuples
     </param>
     <param name='pert_tuple' direction='in'>
       The host program's perturbation tuple, in which the first label
       is the perturbation $a$
     </param>
     <param name='num_freq_configs' direction='in'>
       Number of different frequency configurations
     </param>
     <param name='pert_freqs' direction='in'>
       Complex frequencies of each perturbation label (except for the
       perturbation $a$) over all frequency configurations, size is therefore
       $2\times[(<len_tuple>-1)\times<num_freq_configs>]$, and arranged as
       <c>[num_freq_configs][len_tuple-1][2]</c> in memory (that is, the real
       and imaginary parts of each frequency are consecutive in memory)
     </param>
     <param name='intern_pert_tuple' direction='out'>
       The internal perturbation tuple, in which identical perturbation labels
       are consecutive, and the first one is the perturbation $a$
     </param>
     <param name='intern_pert_freqs' direction='out'>
       Internal complex frequencies (in ascending order among identical
       perturbation labels) of each perturbation label (including the
       perturbation $a$) over all frequency configurations, size is therefore
       $2\times<len_tuple>\times<num_freq_configs>$, and arranged as
       <c>[num_freq_configs][len_tuple][2]</c> in memory (that is, the real and
       imaginary parts of each frequency are consecutive in memory)
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPPertHostToInternal(const RSPPert *rsp_pert,
                                 const QInt len_tuple,
                                 QcPertInt *pert_tuple,
                                 const QInt num_freq_configs,
                                 QReal *pert_freqs)
{
```

```
//    QcPertInt id_pert;  /* identifiers of different perturbation labels */
//    QBool lab_valid;    /* validity of the perturbation labels */
//
//    QInt ipert,jpert;  /* incremental recorders */
//    QInt first_id;     /* first identical pertubation label in the tuple */
//    QInt last_id;      /* last identical pertubation label in the tuple */
//    QBool non_id;      /* indicates if non-identical label found */
//
//    id_pert = 0;
//
//    for (ipert=0,jpert=0; ipert<len_tuple; ) {
//
//      /* checks the current perturbation label against all known
//           perturbation labels */
//        lab_valid = QFALSE;
//        for (ilab=0; ilab<rsp_pert->num_pert_lab; ilab++) {
//            if (pert_tuple[ipert]==rsp_pert->pert_labels[ilab]) {
//                lab_valid = QTRUE;
//                break;
//            }
//        }
//        if (lab_valid==QTRUE) {
//            /* converts the current perturbation label to internal one */
//            intern_pert_tuple[jpert] = (id_pert<<OPENRSP_PERT_LABEL_BIT)
//                                       + pert_tuple[ipert];
//            /* finds the other same perturbation labels */
//
//
//            /* updates the identifier */
//            id_pert++;
//        else {
//            printf("RSPPertCreate>> %"QINT_FMT"-th perturbation %"QCPERTINT_FMT"\n",
//                   ipert,
//                   pert_tuple[ipert]);
//            QErrorExit(FILE_AND_LINE, "invalid perturbation label");
//        }
//
//
//    }
//
//    /* we first try to find consecutive identical pertubation labels */
//    first_id = 0;
//    non_id = QFALSE;
//    for (ipert=first_id; ipert<len_tuple-1; ipert++) {
//        if (pert_tuple[ipert]!=pert_tuple[ipert+1]) {
//            last_id = ipert;
//            non_id = QTRUE;
//            break;
//        }
//    }
//    if (non_id=QTRUE) {
//    }
```

```
//    else {
//    }

    return QSUCCESS;
}
```

where we have also checked the number of different identifiers against `OPENRSP_PERT_ID_MAX`.

Inside OpenRSP, we mostly use the internal perturbation labels/tuples. But when we use callback functions to get integrals and/or expectation values of overlap, one- and two-electron operators as well as contributions from nuclear Hamiltonian, only the host programs' perturbation labels and orders are meaningful for these callback functions; when we want the integrals and/or expectation values of XC functionals[1], we need to convert our internal perturbation tuples to the host programs' ones.

We first consider the former taks—a function that converts an internal pertubration tuple into host program's perturbation labels and orders:

35a ⟨*RSPertAPIs* 26a⟩+≡

```
extern QErrorCode RSPPertInternTupleToHostLabelOrder(const QInt,
                                                     const QcPertInt*,
                                                     const QInt,
                                                     const QcPertInt*,
                                                     const QInt*,
                                                     QInt*,
                                                     QcPertInt*,
                                                     QInt*);
```

This function also checks if the resulted perturbation labels and orders are allowed (or non-zero perturbed quantities) by chechecking against allowed host program's perturbation labels and orders:

35b ⟨*RSPPerturbation.c* 23c⟩+≡

```
/* <function name='RSPPertInternTupleToHostLabelOrder'
              attr='private'
              author='Bin Gao'
              date='2015-10-08'>
     Convert an internal perturbation tuple to host program's pertubation
     labels and the corresponding orders
     <param name='len_intern_tuple' direction='in'>
       Length of the internal perturbation tuple
     </param>
     <param name='intern_pert_tuple' direction='in'>
       The internal perturbation tuple
     </param>
     <param name='num_allowed_labels' direction='in'>
       Number of allowed different host program's perturbation labels
     </param>
     <param name='allowed_pert_labels' direction='in'>
       All the allowed different host program's perturbation labels
     </param>
     <param name='allowed_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different host program's labels
     </param>
```

[1] XC functionals need all density matrices and are calculated differently from other operators, we therefore choose to send the perturbation tuple to XC-functional callback functions

```
      <param name='num_pert' direction='out'>
        Number of different perturbations from the internal perturbation tuple,
        $-1$ indicates there are perturbation labels/orders not allowed
      </param>
      <param name='pert_labels' direction='out'>
        Host program's pertubation labels of the resulted perturbations
      </param>
      <param name='pert_orders' direction='out'>
        Orders of the resulted perturbations
      </param>
      <return>Error information</return>
    </function> */
QErrorCode RSPPertInternTupleToHostLabelOrder(const QInt len_intern_tuple,
                                              const QcPertInt *intern_pert_tuple,
                                              const QInt num_allowed_labels,
                                              const QcPertInt *allowed_pert_labels,
                                              const QInt *allowed_max_orders,
                                              QInt *num_pert,
                                              QcPertInt *pert_labels,
                                              QInt *pert_orders)
{
    QcPertInt host_pert_label;  /* host program's perturbation label */
    QBool pert_allowed;         /* resulted perturbations allowed or not */
    QInt ipert;                 /* incremental recorder for different perturbations */
    QInt ilab;                  /* incremental recorder over perturbation labels */
    QInt idx_allowed;           /* index of the allowed perturbation label */
    ipert = 0;
    for (ilab=0; ilab<len_intern_tuple; ) {
        /* converts to the host program's label */
        host_pert_label = intern_pert_tuple[ilab] & OPENRSP_PERT_LABEL_MAX;
        /* checks if the label is allowed */
        pert_allowed = QFALSE;
        for (idx_allowed=0; idx_allowed<num_allowed_labels; idx_allowed++) {
            if (host_pert_label==allowed_pert_labels[idx_allowed]) {
                pert_allowed = QTRUE;
                break;
            }
        }
        /* returns a negative number if the label is not allowed */
        if (pert_allowed==QFALSE) {
            *num_pert = -1;
            return QSUCCESS;
        }
        /* finds consecutive identical internal perturbation labels */
        pert_labels[ipert] = intern_pert_tuple[ilab];
        pert_orders[ipert] = 1;
        ilab++;
        for (; ilab<len_intern_tuple; ) {
            if (pert_labels[ipert]==intern_pert_tuple[ilab]) {
                pert_orders[ipert]++;
            }
            else {
```

```
                break;
            }
            ilab++;
        }
        /* checks if the order is allowed */
        if (pert_orders[ipert]>allowed_max_orders[idx_allowed]) {
            *num_pert = -1;
            return QSUCCESS;
        }
        /* saves the host program's label */
        pert_labels[ipert] = host_pert_label;
        ipert++;
    }
    *num_pert = ipert;
    return QSUCCESS;
}
```

where the conversion of the internal label to the host program's label is simply done by the bitwise AND operation (`&=`) with `OPENRSP_PERT_LABEL_MAX`.

As such, the latter task—converting our internal perturbation tuples to the host programs' ones is also quite easy:

37a ⟨*RSPertAPIs* 26a⟩+≡

```
extern QErrorCode RSPPertInternTupleToHostTuple(const QInt,
                                                 const QcPertInt*,
                                                 const QInt,
                                                 const QcPertInt*,
                                                 const QInt*,
                                                 QInt*,
                                                 QcPertInt*);
```

37b ⟨*RSPPerturbation.c* 23c⟩+≡

```
/* <function name='RSPPertInternTupleToHostTuple'
             attr='private'
             author='Bin Gao'
             date='2015-10-12'>
     Convert an internal perturbation tuple to the corresponding host
     program's one
     <param name='len_intern_tuple' direction='in'>
       Length of the internal perturbation tuple
     </param>
     <param name='intern_pert_tuple' direction='in'>
       The internal perturbation tuple
     </param>
     <param name='num_allowed_labels' direction='in'>
       Number of allowed different host program's perturbation labels
     </param>
     <param name='allowed_pert_labels' direction='in'>
       All the allowed different host program's perturbation labels
     </param>
     <param name='allowed_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different host program's labels
     </param>
```

```
      <param name='len_tuple' direction='out'>
        Length of the resulted host program's perturbation tuple,
        $-1$ indicates there are perturbation labels/orders not allowed
      </param>
      <param name='pert_tuple' direction='out'>
        The resulted host program's perturbation tuple
      </param>
      <return>Error information</return>
    </function> */
QErrorCode RSPPertInternTupleToHostTuple(const QInt len_intern_tuple,
                                         const QcPertInt *intern_pert_tuple,
                                         const QInt num_allowed_labels,
                                         const QcPertInt *allowed_pert_labels,
                                         const QInt *allowed_max_orders,
                                         QInt *len_tuple,
                                         QcPertInt *pert_tuple)
{
    QcPertInt host_pert_label;  /* host program's perturbation label */
    QInt host_pert_order;       /* order of host program's perturbation */
    QBool pert_allowed;         /* resulted perturbations allowed or not */
    QInt ipert;                 /* incremental recorder for different perturbations */
    QInt ilab;                  /* incremental recorder over perturbation labels */
    QInt idx_allowed;           /* index of the allowed perturbation label */
    ipert = 0;
    for (ilab=0; ilab<len_intern_tuple; ) {
        /* converts to the host program's label */
        host_pert_label = intern_pert_tuple[ilab] & OPENRSP_PERT_LABEL_MAX;
        /* checks if the label is allowed */
        pert_allowed = QFALSE;
        for (idx_allowed=0; idx_allowed<num_allowed_labels; idx_allowed++) {
            if (host_pert_label==allowed_pert_labels[idx_allowed]) {
                pert_allowed = QTRUE;
                break;
            }
        }
        /* returns a negative number if the label is not allowed */
        if (pert_allowed==QFALSE) {
            *len_tuple = -1;
            return QSUCCESS;
        }
        /* finds consecutive identical internal perturbation labels */
        pert_tuple[ipert] = intern_pert_tuple[ilab];
        host_pert_order = 1;
        ilab++;
        for (; ilab<len_intern_tuple; ) {
            if (pert_tuple[ipert]==intern_pert_tuple[ilab]) {
                host_pert_order++;
            }
            else {
                break;
            }
            ilab++;
```

```
        }
        /* checks if the order is allowed */
        if (host_pert_order>allowed_max_orders[idx_allowed]) {
            *len_tuple = -1;
            return QSUCCESS;
        }
        /* saves the host program's labels */
        for (; host_pert_order>0; host_pert_order--) {
            pert_tuple[ipert] = host_pert_label;
            ipert++;
        }
    }
    *len_tuple = ipert;
    return QSUCCESS;
}
```

We will also use the callback function declared in Section 3.3.2 for OpenRSP to construct higher-order derivatives from lower-order ones:

39a ⟨*RSPertAPIs* 26a⟩+≡

```
extern QErrorCode RSPPertGetConcatenation(const RSPPert*,
                                          const QcPertInt,
                                          const QInt,
                                          const QInt,
                                          const QInt,
                                          const QInt*,
                                          QInt*);
```

This function actually uses the callback function `get_pert_concatenation()` to get ranks of components of sub-perturbation tuples (with the same perturbation label) for given components of the corresponding concatenated perturbation tuple:

39b ⟨*RSPPerturbation.c* 23c⟩+≡

```
/* <function name='RSPPertGetConcatenation'
             attr='private'
             author='Bin Gao'
             date='2015-06-28'>
     Gets the ranks of components of sub-perturbation tuples
     <param name='rsp_pert' direction='inout'>
       The context of perturbations
     </param>
     <param name='intern_pert_label' direction='in'>
       The internal perturbation label
     </param>
     <param name='first_cat_comp' direction='in'>
       Rank of the first component of the concatenated perturbation tuple
     </param>
     <param name='num_cat_comps' direction='in'>
       Number of components of the concatenated perturbation tuple
     </param>
     <param name='num_sub_tuples' direction='in'>
       Number of sub-perturbation tuples to construct the concatenated
       perturbation tuple
     </param>
     <param name='len_sub_tuples' direction='in'>
```

```
      Length of each sub-perturbation tuple, size is <num_sub_tuples> so
      that the length of the concatenated perturbation tuple is the sum
      of <len_sub_tuples>
     </param>
     <param name='rank_sub_comps' direction='out'>
      Ranks of components of sub-perturbation tuples for the corresponding
      component of the concatenated perturbation tuple, i.e. <num_cat_comps>
      components starting from the one with the rank <first_cat_comp>; size
      is therefore the product of <num_sub_tuples> and <num_cat_comps>, and
      is arranged as <c>[num_cat_comps][num_sub_tuples]</c> in memory
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPPertGetConcatenation(const RSPPert *rsp_pert,
                                   const QcPertInt intern_pert_label,
                                   const QInt first_cat_comp,
                                   const QInt num_cat_comps,
                                   const QInt num_sub_tuples,
                                   const QInt *len_sub_tuples,
                                   QInt *rank_sub_comps)
{
    QcPertInt pert_label;
    /* converts to host program's perturbation label */
    pert_label = intern_pert_label & OPENRSP_PERT_LABEL_MAX;
#if defined(OPENRSP_ZERO_BASED)
    rsp_pert->get_pert_concatenation(pert_label,
                                     first_cat_comp,
                                     num_cat_comps,
                                     num_sub_tuples,
                                     len_sub_tuples,
#if defined(OPENRSP_C_USER_CONTEXT)
                                     rsp_pert->user_ctx,
#endif
                                     rank_sub_comps);
#else
    QInt icomp;  /* incremental recorder over ranks of components */
    rsp_pert->get_pert_concatenation(pert_label,
                                     first_cat_comp+1,
                                     num_cat_comps,
                                     num_sub_tuples,
                                     len_sub_tuples,
#if defined(OPENRSP_C_USER_CONTEXT)
                                     rsp_pert->user_ctx,
#endif
                                     rank_sub_comps);
    for (icomp=0; icomp<num_cat_comps*num_sub_tuples; icomp++) {
        rank_sub_comps[icomp]--;
    }
#endif
    return QSUCCESS;
}
```

Here except for the conversion to host program's perturbation label, we also need to convert ranks to zero-based numbering if users have chosen the one-based numbering.

3.4 Overlap Operator

OpenRSP needs to invoke host program's callback functions to calculate the integral matrices or expectation values of overlap operator as well as its derivatives with respect to different perturbations. Users can use the following API to tell OpenRSP the information of the overlap operator:

42 ⟨*OpenRSP.c* 17a⟩+≡

```
/* <function name='OpenRSPSetOverlap' author='Bin Gao' date='2014-07-30'>
     Set the overlap operator
     <param name='open_rsp' direction='inout'>
       The context of response theory calculations
     </param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels that can act on the
       overlap operator
     </param>
     <param name='pert_labels' direction='in'>
       All the different perturbation labels involved
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different labels
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback function context
     </param>
     <param name='get_overlap_mat' direction='in'>
       User-specified callback function to calculate integral matrices of
       overlap operator as well as its derivatives with respect to different
       perturbations
     </param>
     <param name='get_overlap_exp' direction='in'>
       User-specified callback function to calculate expectation values of
       overlap operator as well as its derivatives with respect to different
       perturbations
     </param>
     <return>Error information</return>
   </function> */
QErrorCode OpenRSPSetOverlap(OpenRSP *open_rsp,
                             const QInt num_pert_lab,
                             const QcPertInt *pert_labels,
                             const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                             void *user_ctx,
#endif
                             const GetOverlapMat get_overlap_mat,
                             const GetOverlapExp get_overlap_exp)
{
    QErrorCode ierr;  /* error information */
    /* creates the context of overlap operator */
    if (open_rsp->overlap!=NULL) {
        ierr = RSPOverlapDestroy(open_rsp->overlap);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPOverlapDestroy()");
```

```
    }
    else {
        open_rsp->overlap = (RSPOverlap *)malloc(sizeof(RSPOverlap));
        if (open_rsp->overlap==NULL) {
            QErrorExit(FILE_AND_LINE, "allocates memory for overlap");
        }
    }
    ierr = RSPOverlapCreate(open_rsp->overlap,
                            num_pert_lab,
                            pert_labels,
                            pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                            user_ctx,
#endif
                            get_overlap_mat,
                            get_overlap_exp);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPOverlapCreate()");
    return QSUCCESS;
}
```

The following header file defines all quantities we need for the overlap operator. Types `GetOverlapMat` and `GetOverlapExp` define the requirements of two callback functions from the host program to calculate respectively the integral matrices and expectation values of overlap operator and its derivatives.

43 ⟨*RSPOverlap.h* 43⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

  <header name='RSPOneOper.h' author='Bin Gao' date='2014-08-05'>
    The header file of overlap operator used inside OpenRSP
  </header>
*/

#if !defined(RSP_OVERLAP_H)
#define RSP_OVERLAP_H

#include "qcmatrix.h"
#include "RSPPerturbation.h"

typedef void (*GetOverlapMat)(const QInt,
                              const QcPertInt*,
                              const QInt*,
                              const QInt,
                              const QcPertInt*,
                              const QInt*,
                              const QInt,
                              const QcPertInt*,
                              const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                              void*,
#endif
                              const QInt,
```

```
                                QcMat*[]);
typedef void (*GetOverlapExp)(const QInt,
                                const QcPertInt*,
                                const QInt*,
                                const QInt,
                                const QcPertInt*,
                                const QInt*,
                                const QInt,
                                const QcPertInt*,
                                const QInt*,
                                const QInt,
                                QcMat*[],
#if defined(OPENRSP_C_USER_CONTEXT)
                                void*,
#endif
                                const QInt,
                                QReal*);
```

⟨*RSPOverlapStruct* 44⟩

⟨*RSPOverlapAPIs* 45a⟩

```
#endif
```

The context of overlap operator is:

44 ⟨*RSPOverlapStruct* 44⟩≡

```
typedef struct {
    QInt num_pert_lab;              /* number of different perturbation labels
                                       that can act as perturbations on the
                                       overlap operator */
    QInt bra_num_pert;              /* number of perturbations on the bra center,
                                       only used for callback functions */
    QInt ket_num_pert;              /* number of perturbations on the ket center,
                                       only used for callback functions */
    QInt oper_num_pert;             /* number of perturbations on the overlap operator,
                                       only used for callback functions */
    QInt *pert_max_orders;          /* allowed maximal order of a perturbation
                                       described by exactly one of these
                                       different labels */
    QInt *bra_pert_orders;          /* orders of perturbations on the bra center,
                                       only used for callback functions */
    QInt *ket_pert_orders;          /* orders of perturbations on the ket center,
                                       only used for callback functions */
    QInt *oper_pert_orders;         /* orders of perturbations on the overlap operator,
                                       only used for callback functions */
    QcPertInt *pert_labels;         /* all the different perturbation labels */
    QcPertInt *bra_pert_labels;     /* labels of perturbations on the bra center,
                                       only used for callback functions */
    QcPertInt *ket_pert_labels;     /* labels of perturbations on the ket center,
                                       only used for callback functions */
    QcPertInt *oper_pert_labels;    /* labels of perturbations on the overlap operator,
                                       only used for callback functions */
#if defined(OPENRSP_C_USER_CONTEXT)
```

```
    void *user_ctx;                 /* user-defined callback-function context */
#endif
    GetOverlapMat get_overlap_mat;  /* user-specified function for calculating
                                       integral matrices */
    GetOverlapExp get_overlap_exp;  /* user-specified function for calculating
                                       expectation values */
} RSPOverlap;
```

and the functions related to the overlap operator:

45a ⟨*RSPOverlapAPIs* 45a⟩≡

```
extern QErrorCode RSPOverlapCreate(RSPOverlap*,
                                   const QInt,
                                   const QcPertInt*,
                                   const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                   void*,
#endif
                                   const GetOverlapMat,
                                   const GetOverlapExp);
extern QErrorCode RSPOverlapAssemble(RSPOverlap*,const RSPPert*);
extern QErrorCode RSPOverlapWrite(const RSPOverlap*,FILE*);
extern QErrorCode RSPOverlapGetMat(RSPOverlap*,
                                   const QInt,
                                   const QcPertInt*,
                                   const QInt,
                                   const QcPertInt*,
                                   const QInt,
                                   const QcPertInt*,
                                   const QInt,
                                   QcMat*[]);
extern QErrorCode RSPOverlapGetExp(RSPOverlap*,
                                   const QInt,
                                   const QcPertInt*,
                                   const QInt,
                                   const QcPertInt*,
                                   const QInt,
                                   const QcPertInt*,
                                   const QInt,
                                   QcMat*[],
                                   const QInt,
                                   QReal*);
extern QErrorCode RSPOverlapDestroy(RSPOverlap*);
```

Let us now implement all the functions declared:

45b ⟨*RSPOverlap.c* 45b⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩
*/

#include "RSPOverlap.h"

/* <function name='RSPOverlapCreate'
             attr='private'
             author='Bin Gao'
```

```
            date='2014-08-05'>
     Create the context of overlap operator, should be called at first
     <param name='overlap' direction='inout'>
       The context of overlap operator
     </param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels that can act as
       perturbations on the overlap operator
     </param>
     <param name='pert_labels' direction='in'>
       All the different perturbation labels
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different labels
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback-function context
     </param>
     <param name='get_overlap_mat' direction='in'>
       User-specified function for calculating integral matrices of the
       overlap operator and its derivatives
     </param>
     <param name='get_overlap_exp' direction='in'>
       User-specified function for calculating expectation values of the
       overlap operator and its derivatives
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPOverlapCreate(RSPOverlap *overlap,
                            const QInt num_pert_lab,
                            const QcPertInt *pert_labels,
                            const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                            void *user_ctx,
#endif
                            const GetOverlapMat get_overlap_mat,
                            const GetOverlapExp get_overlap_exp)
{
    QInt ilab;  /* incremental recorders over perturbation labels */
    QInt jlab;
    if (num_pert_lab<0) {
        printf("RSPOverlapCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        QErrorExit(FILE_AND_LINE, "invalid number of perturbation labels");
    }
    else if (num_pert_lab>OPENRSP_PERT_LABEL_MAX) {
        printf("RSPOverlapCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        printf("RSPOverlapCreate>> maximal value for pert. labels %"QCPERTINT_FMT"\n",
               OPENRSP_PERT_LABEL_MAX);
        QErrorExit(FILE_AND_LINE, "too many perturbation labels");
```

```
    }
    overlap->num_pert_lab = num_pert_lab;
    if (overlap->num_pert_lab>0) {
        overlap->pert_max_orders = (QInt *)malloc(num_pert_lab*sizeof(QInt));
        if (overlap->pert_max_orders==NULL) {
            printf("RSPOverlapCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for allowed maximal orders");
        }
        overlap->bra_pert_orders = (QInt *)malloc(num_pert_lab*sizeof(QInt));
        if (overlap->bra_pert_orders==NULL) {
            printf("RSPOverlapCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for pert. orders on bra center");
        }
        overlap->ket_pert_orders = (QInt *)malloc(num_pert_lab*sizeof(QInt));
        if (overlap->ket_pert_orders==NULL) {
            printf("RSPOverlapCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for pert. orders on ket center");
        }
        overlap->oper_pert_orders = (QInt *)malloc(num_pert_lab*sizeof(QInt));
        if (overlap->oper_pert_orders==NULL) {
            printf("RSPOverlapCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for pert. orders on overlap operato
        }
        overlap->pert_labels = (QcPertInt *)malloc(num_pert_lab*sizeof(QcPertInt));
        if (overlap->pert_labels==NULL) {
            printf("RSPOverlapCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for perturbation labels");
        }
        overlap->bra_pert_labels = (QcPertInt *)malloc(num_pert_lab*sizeof(QcPertInt));
        if (overlap->bra_pert_labels==NULL) {
            printf("RSPOverlapCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for pert. labels on bra center");
        }
        overlap->ket_pert_labels = (QcPertInt *)malloc(num_pert_lab*sizeof(QcPertInt));
        if (overlap->ket_pert_labels==NULL) {
            printf("RSPOverlapCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for pert. labels on ket center");
        }
        overlap->oper_pert_labels = (QcPertInt *)malloc(num_pert_lab*sizeof(QcPertInt));
        if (overlap->oper_pert_labels==NULL) {
            printf("RSPOverlapCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for pert. labels on overlap operato
        }
        for (ilab=0; ilab<num_pert_lab; ilab++) {
```

```
            if (pert_labels[ilab]>OPENRSP_PERT_LABEL_MAX) {
                printf("RSPOverlapCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
                       ilab,
                       pert_labels[ilab]);
                printf("RSPOverlapCreate>> maximal value for pert. labels %"QCPERTINT_FMT"\
                       OPENRSP_PERT_LABEL_MAX);
                QErrorExit(FILE_AND_LINE, "invalid perturbation label");
            }
            /* each element of <pert_labels> should be unique */
            for (jlab=0; jlab<ilab; jlab++) {
                if (pert_labels[jlab]==pert_labels[ilab]) {
                    printf("RSPOverlapCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\
                           jlab,
                           pert_labels[jlab]);
                    printf("RSPOverlapCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\
                           ilab,
                           pert_labels[ilab]);
                    QErrorExit(FILE_AND_LINE, "repeated perturbation labels not allowed");
                }
            }
            overlap->pert_labels[ilab] = pert_labels[ilab];
            if (pert_max_orders[ilab]<1) {
                printf("RSPOverlapCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
                       ilab,
                       pert_labels[ilab]);
                printf("RSPOverlapCreate>> allowed maximal order is %"QINT_FMT"\n",
                       pert_max_orders[ilab]);
                QErrorExit(FILE_AND_LINE, "only positive order allowed");
            }
            overlap->pert_max_orders[ilab] = pert_max_orders[ilab];
        }
    }
    else {
        overlap->pert_max_orders = NULL;
        overlap->bra_pert_orders = NULL;
        overlap->ket_pert_orders = NULL;
        overlap->oper_pert_orders = NULL;
        overlap->pert_labels = NULL;
        overlap->bra_pert_labels = NULL;
        overlap->ket_pert_labels = NULL;
        overlap->oper_pert_labels = NULL;
    }
#if defined(OPENRSP_C_USER_CONTEXT)
    overlap->user_ctx = user_ctx;
#endif
    overlap->get_overlap_mat = get_overlap_mat;
    overlap->get_overlap_exp = get_overlap_exp;
    return QSUCCESS;
}
```

As shown here, we allow for an overlap operator that does not depend on any peraturbation—`num_pert_lab==0`, i.e. any perturbed integral matrix and expectation value of this overlap operator

is zero.

49 ⟨*RSPOverlap.c* 45b⟩+≡

```
/* <function name='RSPOverlapAssemble'
              attr='private'
              author='Bin Gao'
              date='2014-08-05'>
     Assembles the context of overlap operator
     <param name='overlap' direction='inout'>
       The context of overlap operator
     </param>
     <param name='rsp_pert' direction='in'>
       The context of perturbations
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPOverlapAssemble(RSPOverlap *overlap, const RSPPert *rsp_pert)
{
    QErrorCode ierr;  /* error information */
    if (overlap->num_pert_lab>0 &&
        (overlap->pert_labels==NULL || overlap->pert_max_orders==NULL)) {
        QErrorExit(FILE_AND_LINE, "perturbations of overlap operator not set");
    }
    if (overlap->get_overlap_mat==NULL || overlap->get_overlap_exp==NULL) {
        QErrorExit(FILE_AND_LINE, "callback functions of overlap operator not set");
    }
    /* checks perturbation labels and allowed maximal orders against
       all known perturbations */
    ierr = RSPPertValidateLabelOrder(rsp_pert,
                                     overlap->num_pert_lab,
                                     overlap->pert_labels,
                                     overlap->pert_max_orders);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertValidateLabelOrder()");
    return QSUCCESS;
}

/* <function name='RSPOverlapWrite'
              attr='private'
              author='Bin Gao'
              date='2014-08-05'>
     Writes the context of overlap operator
     <param name='overlap' direction='in'>
       The context of overlap operator
     </param>
     <param name='fp_overlap' direction='inout'>File pointer</param>
     <return>Error information</return>
   </function> */
QErrorCode RSPOverlapWrite(const RSPOverlap *overlap, FILE *fp_overlap)
{
    QInt ilab;  /* incremental recorder over perturbation labels */
    fprintf(fp_overlap,
            "RSPOverlapWrite>> number of pert. labels that overlap operator depends on %"QI
            overlap->num_pert_lab);
```

```
    fprintf(fp_overlap, "RSPOverlapWrite>> label           maximum-order\n");
    for (ilab=0; ilab<overlap->num_pert_lab; ilab++) {
        fprintf(fp_overlap,
                "RSPOverlapWrite>>       %"QCPERTINT_FMT"                  %"QINT_FMT"\n",
                overlap->pert_labels[ilab],
                overlap->pert_max_orders[ilab]);
    }
#if defined(OPENRSP_C_USER_CONTEXT)
    if (overlap->user_ctx!=NULL) {
        fprintf(fp_overlap, "RSPOverlapWrite>> user-defined function context given\n");
    }
#endif
    return QSUCCESS;
}

/* <function name='RSPOverlapGetMat'
             attr='private'
             author='Bin Gao'
             date='2015-10-15'>
     Calculates integral matrices of the overlap operator
     <param name='overlap' direction='inout'>
       The context of overlap operator
     </param>
     <param name='bra_len_tuple' direction='in'>
       Length of the perturbation tuple on the bra center
     </param>
     <param name='bra_pert_tuple' direction='in'>
       Perturbation tuple on the bra center
     </param>
     <param name='ket_len_tuple' direction='in'>
       Length of the perturbation tuple on the ket center
     </param>
     <param name='ket_pert_tuple' direction='in'>
       Perturbation tuple on the ket center
     </param>
     <param name='oper_len_tuple' direction='in'>
       Length of the perturbation tuple on the overlap operator
     </param>
     <param name='oper_pert_tuple' direction='in'>
       Perturbation tuple on the overlap operator
     </param>
     <param name='num_int' direction='in'>
       Number of the integral matrices
     </param>
     <param name='val_int' direction='inout'>
       The integral matrices
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPOverlapGetMat(RSPOverlap *overlap,
                            const QInt bra_len_tuple,
                            const QcPertInt *bra_pert_tuple,
```

```
                            const QInt ket_len_tuple,
                            const QcPertInt *ket_pert_tuple,
                            const QInt oper_len_tuple,
                            const QcPertInt *oper_pert_tuple,
                            const QInt num_int,
                            QcMat *val_int[])
{
    QErrorCode ierr;  /* error information */
    /* gets perturbation labels and corresponding orders out of the internal
       perturbation tuple on the bra center */
    ierr = RSPPertInternTupleToHostLabelOrder(bra_len_tuple,
                                              bra_pert_tuple,
                                              overlap->num_pert_lab,
                                              overlap->pert_labels,
                                              overlap->pert_max_orders,
                                              &overlap->bra_num_pert,
                                              overlap->bra_pert_labels,
                                              overlap->bra_pert_orders);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostLabelOrder()");
    /* checks if the perturbations on the bra center result in zero values */
    if (overlap->bra_num_pert<0) return QSUCCESS;
    /* performs the same procedure for the perturbations on the ket center */
    ierr = RSPPertInternTupleToHostLabelOrder(ket_len_tuple,
                                              ket_pert_tuple,
                                              overlap->num_pert_lab,
                                              overlap->pert_labels,
                                              overlap->pert_max_orders,
                                              &overlap->ket_num_pert,
                                              overlap->ket_pert_labels,
                                              overlap->ket_pert_orders);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostLabelOrder()");
    if (overlap->ket_num_pert<0) return QSUCCESS;
    /* performs the same procedure for the perturbations on the overlap operator */
    ierr = RSPPertInternTupleToHostLabelOrder(oper_len_tuple,
                                              oper_pert_tuple,
                                              overlap->num_pert_lab,
                                              overlap->pert_labels,
                                              overlap->pert_max_orders,
                                              &overlap->oper_num_pert,
                                              overlap->oper_pert_labels,
                                              overlap->oper_pert_orders);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostLabelOrder()");
    if (overlap->oper_num_pert<0) return QSUCCESS;
    /* calculates integral matrices using the callback function */
    overlap->get_overlap_mat(overlap->bra_num_pert,
                             overlap->bra_pert_labels,
                             overlap->bra_pert_orders,
                             overlap->ket_num_pert,
                             overlap->ket_pert_labels,
                             overlap->ket_pert_orders,
                             overlap->oper_num_pert,
                             overlap->oper_pert_labels,
```

```
                              overlap->oper_pert_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                              overlap->user_ctx,
#endif
                              num_int,
                              val_int);
    return QSUCCESS;
}

/* <function name='RSPOverlapGetExp'
             attr='private'
             author='Bin Gao'
             date='2015-10-15'>
     Calculates expectation values of the overlap operator
     <param name='overlap' direction='inout'>
       The context of overlap operator
     </param>
     <param name='bra_len_tuple' direction='in'>
       Length of the perturbation tuple on the bra center
     </param>
     <param name='bra_pert_tuple' direction='in'>
       Perturbation tuple on the bra center
     </param>
     <param name='ket_len_tuple' direction='in'>
       Length of the perturbation tuple on the ket center
     </param>
     <param name='ket_pert_tuple' direction='in'>
       Perturbation tuple on the ket center
     </param>
     <param name='oper_len_tuple' direction='in'>
       Length of the perturbation tuple on the overlap operator
     </param>
     <param name='oper_pert_tuple' direction='in'>
       Perturbation tuple on the overlap operator
     </param>
     <param name='num_dmat' direction='in'>
       Number of atomic orbital (AO) based density matrices
     </param>
     <param name='dens_mat' direction='in'>
       The AO based density matrices
     </param>
     <param name='num_exp' direction='in'>
       Number of the expectation values
     </param>
     <param name='val_exp' direction='inout'>
       The expectation values
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPOverlapGetExp(RSPOverlap *overlap,
                            const QInt bra_len_tuple,
                            const QcPertInt *bra_pert_tuple,
```

```
                            const QInt ket_len_tuple,
                            const QcPertInt *ket_pert_tuple,
                            const QInt oper_len_tuple,
                            const QcPertInt *oper_pert_tuple,
                            const QInt num_dmat,
                            QcMat *dens_mat[],
                            const QInt num_exp,
                            QReal *val_exp)
{
    QErrorCode ierr;  /* error information */
    /* gets perturbation labels and corresponding orders out of the internal
       perturbation tuple on the bra center */
    ierr = RSPPertInternTupleToHostLabelOrder(bra_len_tuple,
                                              bra_pert_tuple,
                                              overlap->num_pert_lab,
                                              overlap->pert_labels,
                                              overlap->pert_max_orders,
                                              &overlap->bra_num_pert,
                                              overlap->bra_pert_labels,
                                              overlap->bra_pert_orders);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostLabelOrder()");
    /* checks if the perturbations on the bra center result in zero values */
    if (overlap->bra_num_pert<0) return QSUCCESS;
    /* performs the same procedure for the perturbations on the ket center */
    ierr = RSPPertInternTupleToHostLabelOrder(ket_len_tuple,
                                              ket_pert_tuple,
                                              overlap->num_pert_lab,
                                              overlap->pert_labels,
                                              overlap->pert_max_orders,
                                              &overlap->ket_num_pert,
                                              overlap->ket_pert_labels,
                                              overlap->ket_pert_orders);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostLabelOrder()");
    if (overlap->ket_num_pert<0) return QSUCCESS;
    /* performs the same procedure for the perturbations on the overlap operator */
    ierr = RSPPertInternTupleToHostLabelOrder(oper_len_tuple,
                                              oper_pert_tuple,
                                              overlap->num_pert_lab,
                                              overlap->pert_labels,
                                              overlap->pert_max_orders,
                                              &overlap->oper_num_pert,
                                              overlap->oper_pert_labels,
                                              overlap->oper_pert_orders);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostLabelOrder()");
    if (overlap->oper_num_pert<0) return QSUCCESS;
    /* calculates expectation values using the callback function */
    overlap->get_overlap_exp(overlap->bra_num_pert,
                             overlap->bra_pert_labels,
                             overlap->bra_pert_orders,
                             overlap->ket_num_pert,
                             overlap->ket_pert_labels,
                             overlap->ket_pert_orders,
```

```
                              overlap->oper_num_pert,
                              overlap->oper_pert_labels,
                              overlap->oper_pert_orders,
                              num_dmat,
                              dens_mat,
#if defined(OPENRSP_C_USER_CONTEXT)
                              overlap->user_ctx,
#endif
                              num_exp,
                              val_exp);
    return QSUCCESS;
}

/* <function name='RSPOverlapDestroy'
             attr='private'
             author='Bin Gao'
             date='2014-08-05'>
     Destroys the context of overlap operator, should be called at the end
     <param name='overlap' direction='inout'>
       The context of overlap operator
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPOverlapDestroy(RSPOverlap *overlap)
{
    if (overlap->pert_max_orders!=NULL) {
        free(overlap->pert_max_orders);
        overlap->pert_max_orders = NULL;
    }
    if (overlap->bra_pert_orders!=NULL) {
        free(overlap->bra_pert_orders);
        overlap->bra_pert_orders = NULL;
    }
    if (overlap->ket_pert_orders!=NULL) {
        free(overlap->ket_pert_orders);
        overlap->ket_pert_orders = NULL;
    }
    if (overlap->oper_pert_orders!=NULL) {
        free(overlap->oper_pert_orders);
        overlap->oper_pert_orders = NULL;
    }
    if (overlap->pert_labels!=NULL) {
        free(overlap->pert_labels);
        overlap->pert_labels = NULL;
    }
    if (overlap->bra_pert_labels!=NULL) {
        free(overlap->bra_pert_labels);
        overlap->bra_pert_labels = NULL;
    }
    if (overlap->ket_pert_labels!=NULL) {
        free(overlap->ket_pert_labels);
        overlap->ket_pert_labels = NULL;
```

```
    }
    if (overlap->oper_pert_labels!=NULL) {
        free(overlap->oper_pert_labels);
        overlap->oper_pert_labels = NULL;
    }
#if defined(OPENRSP_C_USER_CONTEXT)
    overlap->user_ctx = NULL;
#endif
    overlap->get_overlap_mat = NULL;
    overlap->get_overlap_exp = NULL;
    return QSUCCESS;
}
```

3.5 One-Electron Operators

Users can use the following API to add different one-electron operators:

56 ⟨*OpenRSP.c* 17a⟩+≡

```
/* <function name='OpenRSPAddOneOper' author='Bin Gao' date='2014-07-30'>
     Add a one-electron operator to the Hamiltonian
     <param name='open_rsp' direction='inout'>
       The context of response theory calculations
     </param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels that can act on the
       one-electron operator
     </param>
     <param name='pert_labels' direction='in'>
       All the different perturbation labels involved
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different labels
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback function context
     </param>
     <param name='get_one_oper_mat' direction='in'>
       User-specified callback function to calculate integral matrices of
       one-electron operator as well as its derivatives with respect to
       different perturbations
     </param>
     <param name='get_one_oper_exp' direction='in'>
       User-specified callback function to calculate expectation values of
       one-electron operator as well as its derivatives with respect to
       different perturbations
     </param>
     <return>Error information</return>
   </function> */
QErrorCode OpenRSPAddOneOper(OpenRSP *open_rsp,
                             const QInt num_pert_lab,
                             const QcPertInt *pert_labels,
                             const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                             void *user_ctx,
#endif
                             const GetOneOperMat get_one_oper_mat,
                             const GetOneOperExp get_one_oper_exp)
{
    QErrorCode ierr;  /* error information */
    /* creates the linked list of one-electron operators */
    if (open_rsp->one_oper==NULL) {
        ierr = RSPOneOperCreate(&open_rsp->one_oper,
                                num_pert_lab,
                                pert_labels,
                                pert_max_orders,
```

```
#if defined(OPENRSP_C_USER_CONTEXT)
                                 user_ctx,
#endif
                                 get_one_oper_mat,
                                 get_one_oper_exp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPOneOperCreate()");
    }
    /* adds the one-electron operator to the linked list */
    else {
        ierr = RSPOneOperAdd(open_rsp->one_oper,
                             num_pert_lab,
                             pert_labels,
                             pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                             user_ctx,
#endif
                             get_one_oper_mat,
                             get_one_oper_exp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPOneOperAdd()");
    }
    return QSUCCESS;
}
```

The following header file defines all quantities we need for one-electron operators. Types `GetOneOperMat` and `GetOneOperpExp` define the requirements of two callback functions from the host program to calculate respectively the integral matrices and expectation values of a one-electron operator and its derivatives.

57 ⟨*RSPOneOper.h* 57⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

  <header name='RSPOneOper.h' author='Bin Gao' date='2014-07-30'>
    The header file of one-electron operators used inside OpenRSP
  </header>
*/

#if !defined(RSP_ONEOPER_H)
#define RSP_ONEOPER_H

#include "qcmatrix.h"
#include "RSPPerturbation.h"

typedef void (*GetOneOperMat)(const QInt,
                              const QcPertInt*,
                              const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                              void*,
#endif
                              const QInt,
                              QcMat*[]);
typedef void (*GetOneOperExp)(const QInt,
                              const QcPertInt*,
```

```
                                const QInt*,
                                const QInt,
                                QcMat*[],
#if defined(OPENRSP_C_USER_CONTEXT)
                                void*,
#endif
                                const QInt,
                                QReal*);
```

⟨*RSPOneOperStruct* 58a⟩

⟨*RSPOneOperAPIs* 58b⟩

```
#endif
```

Here we use a linked list for the context of one-electron operators:

58a ⟨*RSPOneOperStruct* 58a⟩≡

```
typedef struct RSPOneOper RSPOneOper;
struct RSPOneOper {
    QInt num_pert_lab;              /* number of different perturbation labels
                                       that can act as perturbations on the
                                       one-electron operator */
    QInt oper_num_pert;             /* number of perturbations on the
                                       one-electron operator, only used for
                                       callback functions */
    QInt *pert_max_orders;          /* allowed maximal order of a perturbation
                                       described by exactly one of these
                                       different labels */
    QInt *oper_pert_orders;         /* orders of perturbations on the
                                       one-electron operator, only used for
                                       callback functions */
    QcPertInt *pert_labels;         /* all the different perturbation labels */
    QcPertInt *oper_pert_labels;    /* labels of perturbations on the
                                       one-electron operator, only used for
                                       callback functions */
#if defined(OPENRSP_C_USER_CONTEXT)
    void *user_ctx;                 /* user-defined callback-function context */
#endif
    GetOneOperMat get_one_oper_mat; /* user-specified function for calculating
                                       integral matrices */
    GetOneOperExp get_one_oper_exp; /* user-specified function for calculating
                                       expectation values */
    RSPOneOper *next_oper;          /* pointer to the next one-electron operator */
};
```

and the functions related to the one-electron operators:

58b ⟨*RSPOneOperAPIs* 58b⟩≡

```
extern QErrorCode RSPOneOperCreate(RSPOneOper**,
                                   const QInt,
                                   const QcPertInt*,
                                   const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                   void*,
#endif
```

```
                                    const GetOneOperMat,
                                    const GetOneOperExp);
extern QErrorCode RSPOneOperAdd(RSPOneOper*,
                                const QInt,
                                const QcPertInt*,
                                const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                void*,
#endif
                                const GetOneOperMat,
                                const GetOneOperExp);
extern QErrorCode RSPOneOperAssemble(RSPOneOper*,const RSPPert*);
extern QErrorCode RSPOneOperWrite(RSPOneOper*,FILE*);
extern QErrorCode RSPOneOperGetMat(RSPOneOper*,
                                    const QInt,
                                    const QcPertInt*,
                                    const QInt,
                                    QcMat*[]);
extern QErrorCode RSPOneOperGetExp(RSPOneOper*,
                                    const QInt,
                                    const QcPertInt*,
                                    const QInt,
                                    QcMat*[],
                                    const QInt,
                                    QReal*);
extern QErrorCode RSPOneOperDestroy(RSPOneOper**);
```

The functions are implemented as follows:

59 ⟨*RSPOneOper.c* 59⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩
*/

#include "RSPOneOper.h"

/* <function name='RSPOneOperCreate'
             attr='private'
             author='Bin Gao'
             date='2014-07-30'>
     Create a node of a linked list for a given one-electron operator, should
     be called at first
     <param name='one_oper' direction='inout'>
       The linked list of one-electron operators
     </param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels that can act as
       perturbations on the one-electron operator
     </param>
     <param name='pert_labels' direction='in'>
       All the different perturbation labels
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
```

```
        the above different labels
       </param>
       <param name='user_ctx' direction='in'>
        User-defined callback-function context
       </param>
       <param name='get_one_oper_mat' direction='in'>
        User-specified function for calculating integral matrices of the
        one-electron operator and its derivatives
       </param>
       <param name='get_one_oper_exp' direction='in'>
        User-specified function for calculating expectation values of the
        one-electron operator and its derivatives
       </param>
       <return>Error information</return>
     </function> */
QErrorCode RSPOneOperCreate(RSPOneOper **one_oper,
                            const QInt num_pert_lab,
                            const QcPertInt *pert_labels,
                            const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                            void *user_ctx,
#endif
                            const GetOneOperMat get_one_oper_mat,
                            const GetOneOperExp get_one_oper_exp)
{
    RSPOneOper *new_oper;  /* new operator */
    QInt ilab;             /* incremental recorders over perturbation labels */
    QInt jlab;
    new_oper = (RSPOneOper *)malloc(sizeof(RSPOneOper));
    if (new_oper==NULL) {
        QErrorExit(FILE_AND_LINE, "allocates memory for one-electron operator");
    }
    if (num_pert_lab<0) {
        printf("RSPOneOperCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        QErrorExit(FILE_AND_LINE, "invalid number of perturbation labels");
    }
    else if (num_pert_lab>OPENRSP_PERT_LABEL_MAX) {
        printf("RSPOneOperCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        printf("RSPOneOperCreate>> maximal value for pert. labels %"QCPERTINT_FMT"\n",
               OPENRSP_PERT_LABEL_MAX);
        QErrorExit(FILE_AND_LINE, "too many perturbation labels");
    }
    new_oper->num_pert_lab = num_pert_lab;
    if (new_oper->num_pert_lab>0) {
        new_oper->pert_max_orders = (QInt *)malloc(num_pert_lab*sizeof(QInt));
        if (new_oper->pert_max_orders==NULL) {
            printf("RSPOneOperCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for allowed maximal orders");
        }
```

```
        new_oper->oper_pert_orders = (QInt *)malloc(num_pert_lab*sizeof(QInt));
        if (new_oper->oper_pert_orders==NULL) {
            printf("RSPOneOperCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for pert. orders on 1el operator");
        }
        new_oper->pert_labels = (QcPertInt *)malloc(num_pert_lab*sizeof(QcPertInt));
        if (new_oper->pert_labels==NULL) {
            printf("RSPOneOperCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for perturbation labels");
        }
        new_oper->oper_pert_labels = (QcPertInt *)malloc(num_pert_lab*sizeof(QcPertInt));
        if (new_oper->oper_pert_labels==NULL) {
            printf("RSPOneOperCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for pert. labels on 1el operator");
        }
        for (ilab=0; ilab<num_pert_lab; ilab++) {
            if (pert_labels[ilab]>OPENRSP_PERT_LABEL_MAX) {
                printf("RSPOneOperCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
                       ilab,
                       pert_labels[ilab]);
                printf("RSPOneOperCreate>> maximal value for pert. labels %"QCPERTINT_FMT"\
                       OPENRSP_PERT_LABEL_MAX);
                QErrorExit(FILE_AND_LINE, "invalid perturbation label");
            }
            /* each element of <pert_labels> should be unique */
            for (jlab=0; jlab<ilab; jlab++) {
                if (pert_labels[jlab]==pert_labels[ilab]) {
                    printf("RSPOneOperCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\
                           jlab,
                           pert_labels[jlab]);
                    printf("RSPOneOperCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\
                           ilab,
                           pert_labels[ilab]);
                    QErrorExit(FILE_AND_LINE, "repeated perturbation labels not allowed");
                }
            }
            new_oper->pert_labels[ilab] = pert_labels[ilab];
            if (pert_max_orders[ilab]<1) {
                printf("RSPOneOperCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
                       ilab,
                       pert_labels[ilab]);
                printf("RSPOneOperCreate>> allowed maximal order is %"QINT_FMT"\n",
                       pert_max_orders[ilab]);
                QErrorExit(FILE_AND_LINE, "only positive order allowed");
            }
            new_oper->pert_max_orders[ilab] = pert_max_orders[ilab];
        }
    }
    else {
```

```
        new_oper->pert_max_orders = NULL;
        new_oper->oper_pert_orders = NULL;
        new_oper->pert_labels = NULL;
        new_oper->oper_pert_labels = NULL;
    }
#if defined(OPENRSP_C_USER_CONTEXT)
    new_oper->user_ctx = user_ctx;
#endif
    new_oper->get_one_oper_mat = get_one_oper_mat;
    new_oper->get_one_oper_exp = get_one_oper_exp;
    new_oper->next_oper = NULL;
    *one_oper = new_oper;
    return QSUCCESS;
}
```

As shown here, we allow for a one-electron operator that does not depend on any peraturbation—`num_pert_lab==0`, i.e. any perturbed integral matrix and expectation value of this one-electron operator is zero.

62 ⟨*RSPOneOper.c* 59⟩+≡

```
/* <function name='RSPOneOperAdd'
             attr='private'
             author='Bin Gao'
             date='2014-07-30'>
     Add a given one-electron operator to the linked list
     <param name='one_oper' direction='inout'>
       The linked list of one-electron operators
     </param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels that can act as
       perturbations on the one-electron operator
     </param>
     <param name='pert_labels' direction='in'>
       All the different perturbation labels
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different labels
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback-function context
     </param>
     <param name='get_one_oper_mat' direction='in'>
       User-specified function for calculating integral matrices of the
       one-electron operator and its derivatives
     </param>
     <param name='get_one_oper_exp' direction='in'>
       User-specified function for calculating expectation values of the
       one-electron operator and its derivatives
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPOneOperAdd(RSPOneOper *one_oper,
```

```
                          const QInt num_pert_lab,
                          const QcPertInt *pert_labels,
                          const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                          void *user_ctx,
#endif
                          const GetOneOperMat get_one_oper_mat,
                          const GetOneOperExp get_one_oper_exp)
{
    RSPOneOper *new_oper;  /* new operator */
    RSPOneOper *cur_oper;  /* current operator */
    QErrorCode ierr;       /* error information */
    /* creates the new operator */
    ierr = RSPOneOperCreate(&new_oper,
                            num_pert_lab,
                            pert_labels,
                            pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                            user_ctx,
#endif
                            get_one_oper_mat,
                            get_one_oper_exp);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPOneOperCreate()");
    /* walks to the last operator */
    cur_oper = one_oper;
    while (cur_oper->next_oper!=NULL) {
        cur_oper = cur_oper->next_oper;
    }
    /* inserts the new operator to the tail of the linked list */
    cur_oper->next_oper = new_oper;
    return QSUCCESS;
}

/* <function name='RSPOneOperAssemble'
             attr='private'
             author='Bin Gao'
             date='2014-07-30'>
     Assembles the linked list of one-electron operators
     <param name='one_oper' direction='inout'>
       The linked list of one-electron operators
     </param>
     <param name='rsp_pert' direction='in'>
       The context of perturbations
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPOneOperAssemble(RSPOneOper *one_oper, const RSPPert *rsp_pert)
{
    QInt ioper;            /* incremental recorder over operators */
    RSPOneOper *cur_oper;  /* current operator */
    QErrorCode ierr;       /* error information */
    ioper = 0;
```

```
    cur_oper = one_oper;
    do {
        if (cur_oper->num_pert_lab>0 &&
            (cur_oper->pert_labels==NULL || cur_oper->pert_max_orders==NULL)) {
            printf("RSPOneOperAssemble>> %"QINT_FMT"-th one-electron operator\n",
                   ioper);
            QErrorExit(FILE_AND_LINE, "perturbations of one-electron operator not set");
        }
        if (cur_oper->get_one_oper_mat==NULL || cur_oper->get_one_oper_exp==NULL) {
            printf("RSPOneOperAssemble>> %"QINT_FMT"-th one-electron operator\n",
                   ioper);
            QErrorExit(FILE_AND_LINE, "callback functions of one-electron operator not set"
        }
        /* checks perturbation labels and allowed maximal orders against
           all known perturbations */
        ierr = RSPPertValidateLabelOrder(rsp_pert,
                                         cur_oper->num_pert_lab,
                                         cur_oper->pert_labels,
                                         cur_oper->pert_max_orders);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertValidateLabelOrder()");
        /* moves to the next operator */
        ioper++;
        cur_oper = cur_oper->next_oper;
    } while (cur_oper!=NULL);
    return QSUCCESS;
}

/* <function name='RSPOneOperWrite'
             attr='private'
             author='Bin Gao'
             date='2014-07-30'>
     Writes the linked list of one-electron operators
     <param name='one_oper' direction='in'>
       The linked list of one-electron operators
     </param>
     <param name='fp_oper' direction='inout'>File pointer</param>
     <return>Error information</return>
   </function> */
QErrorCode RSPOneOperWrite(RSPOneOper *one_oper, FILE *fp_oper)
{
    QInt ioper;            /* incremental recorder over operators */
    RSPOneOper *cur_oper;  /* current operator */
    QInt ilab;             /* incremental recorder over perturbation labels */
    ioper = 0;
    cur_oper = one_oper;
    do {
        fprintf(fp_oper, "RSPOneOperWrite>> operator %"QINT_FMT"\n", ioper);
        fprintf(fp_oper,
                "RSPOneOperWrite>> number of pert. labels that one-electron operator depend
                cur_oper->num_pert_lab);
        fprintf(fp_oper, "RSPOneOperWrite>> label           maximum-order\n");
        for (ilab=0; ilab<cur_oper->num_pert_lab; ilab++) {
```

```
            fprintf(fp_oper,
                    "RSPOneOperWrite>>       %"QCPERTINT_FMT"                  %"QINT_FMT"\
                    cur_oper->pert_labels[ilab],
                    cur_oper->pert_max_orders[ilab]);
        }
#if defined(OPENRSP_C_USER_CONTEXT)
        if (cur_oper->user_ctx!=NULL) {
            fprintf(fp_oper, "RSPOneOperWrite>> user-defined function context given\n");
        }
#endif
        /* moves to the next operator */
        ioper++;
        cur_oper = cur_oper->next_oper;
    } while (cur_oper!=NULL);
    return QSUCCESS;
}

/* <function name='RSPOneOperGetMat'
             attr='private'
             author='Bin Gao'
             date='2015-10-15'>
     Calculates integral matrices of the linked list of one-electron operators
     <param name='one_oper' direction='inout'>
       The linked list of one-electron operators
     </param>
     <param name='oper_len_tuple' direction='in'>
       Length of the perturbation tuple on the linked list of one-electron
       operators
     </param>
     <param name='oper_pert_tuple' direction='in'>
       Perturbation tuple on the linked list of one-electron operators
     </param>
     <param name='num_int' direction='in'>
       Number of the integral matrices
     </param>
     <param name='val_int' direction='inout'>
       The integral matrices
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPOneOperGetMat(RSPOneOper *one_oper,
                            const QInt oper_len_tuple,
                            const QcPertInt *oper_pert_tuple,
                            const QInt num_int,
                            QcMat *val_int[])
{
    RSPOneOper *cur_oper;  /* current operator */
    QErrorCode ierr;       /* error information */
    cur_oper = one_oper;
    do {
        /* gets perturbation labels and corresponding orders out of the internal
           perturbation tuple on the one-electron operator */
```

```
        ierr = RSPPertInternTupleToHostLabelOrder(oper_len_tuple,
                                                  oper_pert_tuple,
                                                  cur_oper->num_pert_lab,
                                                  cur_oper->pert_labels,
                                                  cur_oper->pert_max_orders,
                                                  &cur_oper->oper_num_pert,
                                                  cur_oper->oper_pert_labels,
                                                  cur_oper->oper_pert_orders);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostLabelOrder()"
        /* checks if the perturbations on the one-electron operator
           result in zero values */
        if (cur_oper->oper_num_pert<0) continue;
        /* calculates integral matrices using the callback function */
        cur_oper->get_one_oper_mat(cur_oper->oper_num_pert,
                                   cur_oper->oper_pert_labels,
                                   cur_oper->oper_pert_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                                   cur_oper->user_ctx,
#endif
                                   num_int,
                                   val_int);
        /* moves to the next operator */
        cur_oper = cur_oper->next_oper;
    } while (cur_oper!=NULL);
    return QSUCCESS;
}

/* <function name='RSPOneOperGetExp'
             attr='private'
             author='Bin Gao'
             date='2015-10-15'>
     Calculates expectation values of the linked list of one-electron operators
     <param name='one_oper' direction='inout'>
       The linked list of one-electron operators
     </param>
     <param name='oper_len_tuple' direction='in'>
       Length of the perturbation tuple on the linked list of one-electron
       operators
     </param>
     <param name='oper_pert_tuple' direction='in'>
       Perturbation tuple on the linked list of one-electron operators
     </param>
     <param name='num_dmat' direction='in'>
       Number of atomic orbital (AO) based density matrices
     </param>
     <param name='dens_mat' direction='in'>
       The AO based density matrices
     </param>
     <param name='num_exp' direction='in'>
       Number of the expectation values
     </param>
     <param name='val_exp' direction='inout'>
```

```
       The expectation values
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPOneOperGetExp(RSPOneOper *one_oper,
                            const QInt oper_len_tuple,
                            const QcPertInt *oper_pert_tuple,
                            const QInt num_dmat,
                            QcMat *dens_mat[],
                            const QInt num_exp,
                            QReal *val_exp)
{
    RSPOneOper *cur_oper;  /* current operator */
    QErrorCode ierr;       /* error information */
    cur_oper = one_oper;
    do {
        /* gets perturbation labels and corresponding orders out of the internal
           perturbation tuple on the one-electron operator */
        ierr = RSPPertInternTupleToHostLabelOrder(oper_len_tuple,
                                                  oper_pert_tuple,
                                                  cur_oper->num_pert_lab,
                                                  cur_oper->pert_labels,
                                                  cur_oper->pert_max_orders,
                                                  &cur_oper->oper_num_pert,
                                                  cur_oper->oper_pert_labels,
                                                  cur_oper->oper_pert_orders);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostLabelOrder()"
        /* checks if the perturbations on the one-electron operator
           result in zero values */
        if (cur_oper->oper_num_pert<0) continue;
        /* calculates expectation values using the callback function */
        cur_oper->get_one_oper_exp(cur_oper->oper_num_pert,
                                   cur_oper->oper_pert_labels,
                                   cur_oper->oper_pert_orders,
                                   num_dmat,
                                   dens_mat,
#if defined(OPENRSP_C_USER_CONTEXT)
                                   cur_oper->user_ctx,
#endif
                                   num_exp,
                                   val_exp);
        /* moves to the next operator */
        cur_oper = cur_oper->next_oper;
    } while (cur_oper!=NULL);
    return QSUCCESS;
}

/* <function name='RSPOneOperDestroy'
             attr='private'
             author='Bin Gao'
             date='2014-07-30'>
     Destroys the linked list of one-electron operators, should be called
```

```
     at the end
     <param name='one_oper' direction='inout'>
       The linked list of one-electron operators
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPOneOperDestroy(RSPOneOper **one_oper)
{
    RSPOneOper *cur_oper;   /* current operator */
    RSPOneOper *next_oper;  /* next operator */
    /* walks to the last operator */
    cur_oper = *one_oper;
    while (cur_oper!=NULL) {
        if (cur_oper->pert_max_orders!=NULL) {
            free(cur_oper->pert_max_orders);
            cur_oper->pert_max_orders = NULL;
        }
        if (cur_oper->oper_pert_orders!=NULL) {
            free(cur_oper->oper_pert_orders);
            cur_oper->oper_pert_orders = NULL;
        }
        if (cur_oper->pert_labels!=NULL) {
            free(cur_oper->pert_labels);
            cur_oper->pert_labels = NULL;
        }
        if (cur_oper->oper_pert_labels!=NULL) {
            free(cur_oper->oper_pert_labels);
            cur_oper->oper_pert_labels = NULL;
        }
#if defined(OPENRSP_C_USER_CONTEXT)
        cur_oper->user_ctx = NULL;
#endif
        cur_oper->get_one_oper_mat = NULL;
        cur_oper->get_one_oper_exp = NULL;
        next_oper = cur_oper->next_oper;
        free(cur_oper);
        cur_oper = NULL;
        cur_oper = next_oper;
    }
    return QSUCCESS;
}
```

3.6 Two-Electron Operators

Users can use the following API to add different two-electron operators:

69 ⟨*OpenRSP.c* 17a⟩+≡

```
/* <function name='OpenRSPAddTwoOper' author='Bin Gao' date='2014-08-05'>
     Add a two-electron operator to the Hamiltonian
     <param name='open_rsp' direction='inout'>
       The context of response theory calculations
     </param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels that can act on the
       two-electron operator
     </param>
     <param name='pert_labels' direction='in'>
       All the different perturbation labels involved
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different labels
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback function context
     </param>
     <param name='get_two_oper_mat' direction='in'>
       User-specified callback function to calculate integral matrices of
       two-electron operator as well as its derivatives with respect to
       different perturbations
     </param>
     <param name='get_two_oper_exp' direction='in'>
       User-specified callback function to calculate expectation values of
       two-electron operator as well as its derivatives with respect to
       different perturbations
     </param>
     <return>Error information</return>
   </function> */
QErrorCode OpenRSPAddTwoOper(OpenRSP *open_rsp,
                             const QInt num_pert_lab,
                             const QcPertInt *pert_labels,
                             const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                             void *user_ctx,
#endif
                             const GetTwoOperMat get_two_oper_mat,
                             const GetTwoOperExp get_two_oper_exp)
{
    QErrorCode ierr;  /* error information */
    /* creates the linked list of two-electron operators */
    if (open_rsp->two_oper==NULL) {
        ierr = RSPTwoOperCreate(&open_rsp->two_oper,
                                num_pert_lab,
                                pert_labels,
                                pert_max_orders,
```

```
#if defined(OPENRSP_C_USER_CONTEXT)
                                 user_ctx,
#endif
                                 get_two_oper_mat,
                                 get_two_oper_exp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPTwoOperCreate()");
    }
    /* adds the two-electron operator to the linked list */
    else {
        ierr = RSPTwoOperAdd(open_rsp->two_oper,
                             num_pert_lab,
                             pert_labels,
                             pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                             user_ctx,
#endif
                             get_two_oper_mat,
                             get_two_oper_exp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPTwoOperAdd()");
    }
    return QSUCCESS;
}
```

The following header file defines all quantities we need for two-electron operators. Types `GetTwoOperMat` and `GetTwoOperpExp` define the requirements of two callback functions from the host program to calculate respectively the integral matrices and expectation values of a two-electron operator and its derivatives.

70 ⟨*RSPTwoOper.h* 70⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

  <header name='RSPTwoOper.h' author='Bin Gao' date='2014-08-05'>
    The header file of two-electron operators used inside OpenRSP
  </header>
*/

#if !defined(RSP_TWOOPER_H)
#define RSP_TWOOPER_H

#include "qcmatrix.h"
#include "RSPPerturbation.h"

typedef void (*GetTwoOperMat)(const QInt,
                              const QcPertInt*,
                              const QInt*,
                              const QInt,
                              QcMat*[],
#if defined(OPENRSP_C_USER_CONTEXT)
                              void*,
#endif
                              const QInt,
                              QcMat*[]);
```

```
typedef void (*GetTwoOperExp)(const QInt,
                              const QcPertInt*,
                              const QInt*,
                              const QInt,
                              const QInt*,
                              QcMat*[],
                              const QInt*,
                              QcMat*[],
#if defined(OPENRSP_C_USER_CONTEXT)
                              void*,
#endif
                              const QInt,
                              QReal*);
```

⟨*RSPTwoOperStruct* 71a⟩

⟨*RSPTwoOperAPIs* 71b⟩

```
#endif
```

Here we use a linked list for the context of two-electron operators:

71a ⟨*RSPTwoOperStruct* 71a⟩≡

```
typedef struct RSPTwoOper RSPTwoOper;
struct RSPTwoOper {
    QInt num_pert_lab;               /* number of different perturbation labels
                                        that can act as perturbations on the
                                        two-electron operator */
    QInt oper_num_pert;              /* number of perturbations on the
                                        two-electron operator, only used for
                                        callback functions */
    QInt *pert_max_orders;           /* allowed maximal order of a perturbation
                                        described by exactly one of these
                                        different labels */
    QInt *oper_pert_orders;          /* orders of perturbations on the
                                        two-electron operator, only used for
                                        callback functions */
    QcPertInt *pert_labels;          /* all the different perturbation labels */
    QcPertInt *oper_pert_labels;     /* labels of perturbations on the
                                        two-electron operator, only used for
                                        callback functions */
#if defined(OPENRSP_C_USER_CONTEXT)
    void *user_ctx;                  /* user-defined callback-function context */
#endif
    GetTwoOperMat get_two_oper_mat;  /* user-specified function for calculating
                                        integral matrices */
    GetTwoOperExp get_two_oper_exp;  /* user-specified function for calculating
                                        expectation values */
    RSPTwoOper *next_oper;           /* pointer to the next two-electron operator */
};
```

and the functions related to the two-electron operators:

71b ⟨*RSPTwoOperAPIs* 71b⟩≡

```
extern QErrorCode RSPTwoOperCreate(RSPTwoOper**,
                                   const QInt,
```

```
                                    const QcPertInt*,
                                    const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                    void*,
#endif
                                    const GetTwoOperMat,
                                    const GetTwoOperExp);
extern QErrorCode RSPTwoOperAdd(RSPTwoOper*,
                                const QInt,
                                const QcPertInt*,
                                const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                void*,
#endif
                                const GetTwoOperMat,
                                const GetTwoOperExp);
extern QErrorCode RSPTwoOperAssemble(RSPTwoOper*,const RSPPert*);
extern QErrorCode RSPTwoOperWrite(RSPTwoOper*,FILE*);
extern QErrorCode RSPTwoOperGetMat(RSPTwoOper*,
                                    const QInt,
                                    const QcPertInt*,
                                    const QInt,
                                    QcMat*[],
                                    const QInt,
                                    QcMat*[]);
extern QErrorCode RSPTwoOperGetExp(RSPTwoOper*,
                                    const QInt,
                                    const QcPertInt*,
                                    const QInt,
                                    const QInt*,
                                    QcMat*[],
                                    const QInt*,
                                    QcMat*[],
                                    const QInt,
                                    QReal*);
extern QErrorCode RSPTwoOperDestroy(RSPTwoOper**);
```

The functions are implemented as follows:

72 ⟨*RSPTwoOper.c* 72⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩
*/

#include "RSPTwoOper.h"

/* <function name='RSPTwoOperCreate'
             attr='private'
             author='Bin Gao'
             date='2014-08-06'>
     Create a node of a linked list for a given two-electron operator, should
     be called at first
     <param name='two_oper' direction='inout'>
       The linked list of two-electron operators
```

```
      </param>
      <param name='num_pert_lab' direction='in'>
        Number of all different perturbation labels that can act as
        perturbations on the two-electron operator
      </param>
      <param name='pert_labels' direction='in'>
        All the different perturbation labels
      </param>
      <param name='pert_max_orders' direction='in'>
        Allowed maximal order of a perturbation described by exactly one of
        the above different labels
      </param>
      <param name='user_ctx' direction='in'>
        User-defined callback-function context
      </param>
      <param name='get_two_oper_mat' direction='in'>
        User-specified function for calculating integral matrices of the
        two-electron operator and its derivatives
      </param>
      <param name='get_two_oper_exp' direction='in'>
        User-specified function for calculating expectation values of the
        two-electron operator and its derivatives
      </param>
      <return>Error information</return>
   </function> */
QErrorCode RSPTwoOperCreate(RSPTwoOper **two_oper,
                            const QInt num_pert_lab,
                            const QcPertInt *pert_labels,
                            const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                            void *user_ctx,
#endif
                            const GetTwoOperMat get_two_oper_mat,
                            const GetTwoOperExp get_two_oper_exp)
{
    RSPTwoOper *new_oper;  /* new operator */
    QInt ilab;             /* incremental recorders over perturbation labels */
    QInt jlab;
    new_oper = (RSPTwoOper *)malloc(sizeof(RSPTwoOper));
    if (new_oper==NULL) {
        QErrorExit(FILE_AND_LINE, "allocates memory for two-electron operator");
    }
    if (num_pert_lab<0) {
        printf("RSPTwoOperCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        QErrorExit(FILE_AND_LINE, "invalid number of perturbation labels");
    }
    else if (num_pert_lab>OPENRSP_PERT_LABEL_MAX) {
        printf("RSPTwoOperCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        printf("RSPTwoOperCreate>> maximal value for pert. labels %"QCPERTINT_FMT"\n",
               OPENRSP_PERT_LABEL_MAX);
```

```
        QErrorExit(FILE_AND_LINE, "too many perturbation labels");
    }
    new_oper->num_pert_lab = num_pert_lab;
    if (new_oper->num_pert_lab>0) {
        new_oper->pert_max_orders = (QInt *)malloc(num_pert_lab*sizeof(QInt));
        if (new_oper->pert_max_orders==NULL) {
            printf("RSPTwoOperCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for allowed maximal orders");
        }
        new_oper->oper_pert_orders = (QInt *)malloc(num_pert_lab*sizeof(QInt));
        if (new_oper->oper_pert_orders==NULL) {
            printf("RSPTwoOperCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for pert. orders on 2el operator");
        }
        new_oper->pert_labels = (QcPertInt *)malloc(num_pert_lab*sizeof(QcPertInt));
        if (new_oper->pert_labels==NULL) {
            printf("RSPTwoOperCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for perturbation labels");
        }
        new_oper->oper_pert_labels = (QcPertInt *)malloc(num_pert_lab*sizeof(QcPertInt));
        if (new_oper->oper_pert_labels==NULL) {
            printf("RSPTwoOperCreate>> number of perturbation labels %"QINT_FMT"\n",
                   num_pert_lab);
            QErrorExit(FILE_AND_LINE, "allocates memory for pert. labels on 2el operator");
        }
        for (ilab=0; ilab<num_pert_lab; ilab++) {
            if (pert_labels[ilab]>OPENRSP_PERT_LABEL_MAX) {
                printf("RSPTwoOperCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
                       ilab,
                       pert_labels[ilab]);
                printf("RSPTwoOperCreate>> maximal value for pert. labels %"QCPERTINT_FMT"\
                       OPENRSP_PERT_LABEL_MAX);
                QErrorExit(FILE_AND_LINE, "invalid perturbation label");
            }
            /* each element of <pert_labels> should be unique */
            for (jlab=0; jlab<ilab; jlab++) {
                if (pert_labels[jlab]==pert_labels[ilab]) {
                    printf("RSPTwoOperCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\
                           jlab,
                           pert_labels[jlab]);
                    printf("RSPTwoOperCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\
                           ilab,
                           pert_labels[ilab]);
                    QErrorExit(FILE_AND_LINE, "repeated perturbation labels not allowed");
                }
            }
            new_oper->pert_labels[ilab] = pert_labels[ilab];
            if (pert_max_orders[ilab]<1) {
                printf("RSPTwoOperCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
```

```
                        ilab,
                        pert_labels[ilab]);
                printf("RSPTwoOperCreate>> allowed maximal order is %"QINT_FMT"\n",
                        pert_max_orders[ilab]);
                QErrorExit(FILE_AND_LINE, "only positive order allowed");
            }
            new_oper->pert_max_orders[ilab] = pert_max_orders[ilab];
        }
    }
    else {
        new_oper->pert_max_orders = NULL;
        new_oper->oper_pert_orders = NULL;
        new_oper->pert_labels = NULL;
        new_oper->oper_pert_labels = NULL;
    }
#if defined(OPENRSP_C_USER_CONTEXT)
    new_oper->user_ctx = user_ctx;
#endif
    new_oper->get_two_oper_mat = get_two_oper_mat;
    new_oper->get_two_oper_exp = get_two_oper_exp;
    new_oper->next_oper = NULL;
    *two_oper = new_oper;
    return QSUCCESS;
}
```

As shown here, we allow for a two-electron operator that does not depend on any peraturbation—`num_pert_lab==0`, i.e. any perturbed integral matrix and expectation value of this two-electron operator is zero.

75 ⟨*RSPTwoOper.c* 72⟩+≡

```
/* <function name='RSPTwoOperAdd'
             attr='private'
             author='Bin Gao'
             date='2014-08-06'>
     Add a given two-electron operator to the linked list
     <param name='two_oper' direction='inout'>
       The linked list of two-electron operators
     </param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels that can act as
       perturbations on the two-electron operator
     </param>
     <param name='pert_labels' direction='in'>
       All the different perturbation labels
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different labels
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback-function context
     </param>
     <param name='get_two_oper_mat' direction='in'>
```

```
       User-specified function for calculating integral matrices of the
       two-electron operator and its derivatives
     </param>
     <param name='get_two_oper_exp' direction='in'>
       User-specified function for calculating expectation values of the
       two-electron operator and its derivatives
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPTwoOperAdd(RSPTwoOper *two_oper,
                         const QInt num_pert_lab,
                         const QcPertInt *pert_labels,
                         const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                         void *user_ctx,
#endif
                         const GetTwoOperMat get_two_oper_mat,
                         const GetTwoOperExp get_two_oper_exp)
{
    RSPTwoOper *new_oper;  /* new operator */
    RSPTwoOper *cur_oper;  /* current operator */
    QErrorCode ierr;       /* error information */
    /* creates the new operator */
    ierr = RSPTwoOperCreate(&new_oper,
                            num_pert_lab,
                            pert_labels,
                            pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                            user_ctx,
#endif
                            get_two_oper_mat,
                            get_two_oper_exp);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPTwoOperCreate()");
    /* walks to the last operator */
    cur_oper = two_oper;
    while (cur_oper->next_oper!=NULL) {
        cur_oper = cur_oper->next_oper;
    }
    /* inserts the new operator to the tail of the linked list */
    cur_oper->next_oper = new_oper;
    return QSUCCESS;
}

/* <function name='RSPTwoOperAssemble'
             attr='private'
             author='Bin Gao'
             date='2014-08-06'>
     Assembles the linked list of two-electron operators
     <param name='two_oper' direction='inout'>
       The linked list of two-electron operators
     </param>
     <param name='rsp_pert' direction='in'>
```

```
       The context of perturbations
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPTwoOperAssemble(RSPTwoOper *two_oper, const RSPPert *rsp_pert)
{
    QInt ioper;            /* incremental recorder over operators */
    RSPTwoOper *cur_oper;  /* current operator */
    QErrorCode ierr;       /* error information */
    ioper = 0;
    cur_oper = two_oper;
    do {
        if (cur_oper->num_pert_lab>0 &&
            (cur_oper->pert_labels==NULL || cur_oper->pert_max_orders==NULL)) {
            printf("RSPTwoOperAssemble>> %"QINT_FMT"-th two-electron operator\n",
                   ioper);
            QErrorExit(FILE_AND_LINE, "perturbations of two-electron operator not set");
        }
        if (cur_oper->get_two_oper_mat==NULL || cur_oper->get_two_oper_exp==NULL) {
            printf("RSPTwoOperAssemble>> %"QINT_FMT"-th two-electron operator\n",
                   ioper);
            QErrorExit(FILE_AND_LINE, "callback functions of two-electron operator not set"
        }
        /* checks perturbation labels and allowed maximal orders against
           all known perturbations */
        ierr = RSPPertValidateLabelOrder(rsp_pert,
                                         cur_oper->num_pert_lab,
                                         cur_oper->pert_labels,
                                         cur_oper->pert_max_orders);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertValidateLabelOrder()");
        /* moves to the next operator */
        ioper++;
        cur_oper = cur_oper->next_oper;
    } while (cur_oper!=NULL);
    return QSUCCESS;
}

/* <function name='RSPTwoOperWrite'
             attr='private'
             author='Bin Gao'
             date='2014-08-06'>
     Writes the linked list of two-electron operators
     <param name='two_oper' direction='in'>
       The linked list of two-electron operators
     </param>
     <param name='fp_oper' direction='inout'>File pointer</param>
     <return>Error information</return>
   </function> */
QErrorCode RSPTwoOperWrite(RSPTwoOper *two_oper, FILE *fp_oper)
{
    QInt ioper;            /* incremental recorder over operators */
    RSPTwoOper *cur_oper;  /* current operator */
```

```
    QInt ilab;             /* incremental recorder over perturbation labels */
    ioper = 0;
    cur_oper = two_oper;
    do {
        fprintf(fp_oper, "RSPTwoOperWrite>> operator %"QINT_FMT"\n", ioper);
        fprintf(fp_oper,
                "RSPTwoOperWrite>> number of pert. labels that two-electron operator depend
                cur_oper->num_pert_lab);
        fprintf(fp_oper, "RSPTwoOperWrite>> label           maximum-order\n");
        for (ilab=0; ilab<cur_oper->num_pert_lab; ilab++) {
            fprintf(fp_oper,
                    "RSPTwoOperWrite>>       %"QCPERTINT_FMT"                  %"QINT_FMT"\
                    cur_oper->pert_labels[ilab],
                    cur_oper->pert_max_orders[ilab]);
        }
#if defined(OPENRSP_C_USER_CONTEXT)
        if (cur_oper->user_ctx!=NULL) {
            fprintf(fp_oper, "RSPTwoOperWrite>> user-defined function context given\n");
        }
#endif
        /* moves to the next operator */
        ioper++;
        cur_oper = cur_oper->next_oper;
    } while (cur_oper!=NULL);
    return QSUCCESS;
}

/* <function name='RSPTwoOperGetMat'
             attr='private'
             author='Bin Gao'
             date='2015-10-15'>
     Calculates integral matrices of the linked list of two-electron operators
     <param name='two_oper' direction='inout'>
       The linked list of two-electron operators
     </param>
     <param name='oper_len_tuple' direction='in'>
       Length of the perturbation tuple on the linked list of two-electron
       operators
     </param>
     <param name='oper_pert_tuple' direction='in'>
       Perturbation tuple on the linked list of two-electron operators
     </param>
     <param name='num_int' direction='in'>
       Number of the integral matrices
     </param>
     <param name='val_int' direction='inout'>
       The integral matrices
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPTwoOperGetMat(RSPTwoOper *two_oper,
                            const QInt oper_len_tuple,
```

```
                            const QcPertInt *oper_pert_tuple,
                            const QInt num_dmat,
                            QcMat *dens_mat[],
                            const QInt num_int,
                            QcMat *val_int[])
{
    RSPTwoOper *cur_oper;  /* current operator */
    QErrorCode ierr;       /* error information */
    cur_oper = two_oper;
    do {
        /* gets perturbation labels and corresponding orders out of the internal
           perturbation tuple on the two-electron operator */
        ierr = RSPPertInternTupleToHostLabelOrder(oper_len_tuple,
                                                  oper_pert_tuple,
                                                  cur_oper->num_pert_lab,
                                                  cur_oper->pert_labels,
                                                  cur_oper->pert_max_orders,
                                                  &cur_oper->oper_num_pert,
                                                  cur_oper->oper_pert_labels,
                                                  cur_oper->oper_pert_orders);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostLabelOrder()"
        /* checks if the perturbations on the two-electron operator
           result in zero values */
        if (cur_oper->oper_num_pert<0) continue;
        /* calculates integral matrices using the callback function */
        cur_oper->get_two_oper_mat(cur_oper->oper_num_pert,
                                   cur_oper->oper_pert_labels,
                                   cur_oper->oper_pert_orders,
                                   num_dmat,
                                   dens_mat,
#if defined(OPENRSP_C_USER_CONTEXT)
                                   cur_oper->user_ctx,
#endif
                                   num_int,
                                   val_int);
        /* moves to the next operator */
        cur_oper = cur_oper->next_oper;
    } while (cur_oper!=NULL);
    return QSUCCESS;
}

/* <function name='RSPTwoOperGetExp'
             attr='private'
             author='Bin Gao'
             date='2015-10-15'>
     Calculates expectation values of the linked list of two-electron operators
     <param name='two_oper' direction='inout'>
       The linked list of two-electron operators
     </param>
     <param name='oper_len_tuple' direction='in'>
       Length of the perturbation tuple on the linked list of two-electron
       operators
```

```
      </param>
      <param name='oper_pert_tuple' direction='in'>
        Perturbation tuple on the linked list of two-electron operators
      </param>
      <param name='dmat_len_tuple' direction='in'>
        Length of different perturbation tuples of the left-hand-side (LHS) and
        right-hand-side (RHS) atomic orbital (AO) based density matrices passed
      </param>
      <param name='num_LHS_dmat' direction='in'>
        Number of LHS AO based density matrices passed for each LHS density
        matrix perturbation tuple
      </param>
      <param name='LHS_dens_mat' direction='in'>
        The LHS AO based density matrices
      </param>
      <param name='num_RHS_dmat' direction='in'>
        Number of RHS AO based density matrices passed for each RHS density
        matrix perturbation tuple
      </param>
      <param name='RHS_dens_mat' direction='in'>
        The RHS AO based density matrices
      </param>
      <param name='num_exp' direction='in'>
        Number of the expectation values
      </param>
      <param name='val_exp' direction='inout'>
        The expectation values
      </param>
      <return>Error information</return>
    </function> */
QErrorCode RSPTwoOperGetExp(RSPTwoOper *two_oper,
                            const QInt oper_len_tuple,
                            const QcPertInt *oper_pert_tuple,
                            const QInt dmat_len_tuple,
                            const QInt *num_LHS_dmat,
                            QcMat *LHS_dens_mat[],
                            const QInt *num_RHS_dmat,
                            QcMat *RHS_dens_mat[],
                            const QInt num_exp,
                            QReal *val_exp)
{
    RSPTwoOper *cur_oper;  /* current operator */
    QErrorCode ierr;       /* error information */
    cur_oper = two_oper;
    do {
        /* gets perturbation labels and corresponding orders out of the internal
           perturbation tuple on the two-electron operator */
        ierr = RSPPertInternTupleToHostLabelOrder(oper_len_tuple,
                                                  oper_pert_tuple,
                                                  cur_oper->num_pert_lab,
                                                  cur_oper->pert_labels,
                                                  cur_oper->pert_max_orders,
```

```
                                                &cur_oper->oper_num_pert,
                                                cur_oper->oper_pert_labels,
                                                cur_oper->oper_pert_orders);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostLabelOrder()"
        /* checks if the perturbations on the two-electron operator
           result in zero values */
        if (cur_oper->oper_num_pert<0) continue;
        /* calculates expectation values using the callback function */
        cur_oper->get_two_oper_exp(cur_oper->oper_num_pert,
                                   cur_oper->oper_pert_labels,
                                   cur_oper->oper_pert_orders,
                                   dmat_len_tuple,
                                   num_LHS_dmat,
                                   LHS_dens_mat,
                                   num_RHS_dmat,
                                   RHS_dens_mat,
#if defined(OPENRSP_C_USER_CONTEXT)
                                   cur_oper->user_ctx,
#endif
                                   num_exp,
                                   val_exp);
        /* moves to the next operator */
        cur_oper = cur_oper->next_oper;
    } while (cur_oper!=NULL);
    return QSUCCESS;
}

/* <function name='RSPTwoOperDestroy'
             attr='private'
             author='Bin Gao'
             date='2014-08-06'>
     Destroys the linked list of two-electron operators, should be called
     at the end
     <param name='two_oper' direction='inout'>
       The linked list of two-electron operators
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPTwoOperDestroy(RSPTwoOper **two_oper)
{
    RSPTwoOper *cur_oper;   /* current operator */
    RSPTwoOper *next_oper;  /* next operator */
    /* walks to the last operator */
    cur_oper = *two_oper;
    while (cur_oper!=NULL) {
        if (cur_oper->pert_max_orders!=NULL) {
            free(cur_oper->pert_max_orders);
            cur_oper->pert_max_orders = NULL;
        }
        if (cur_oper->oper_pert_orders!=NULL) {
            free(cur_oper->oper_pert_orders);
            cur_oper->oper_pert_orders = NULL;
```

```
        }
        if (cur_oper->pert_labels!=NULL) {
            free(cur_oper->pert_labels);
            cur_oper->pert_labels = NULL;
        }
        if (cur_oper->oper_pert_labels!=NULL) {
            free(cur_oper->oper_pert_labels);
            cur_oper->oper_pert_labels = NULL;
        }
#if defined(OPENRSP_C_USER_CONTEXT)
        cur_oper->user_ctx = NULL;
#endif
        cur_oper->get_two_oper_mat = NULL;
        cur_oper->get_two_oper_exp = NULL;
        next_oper = cur_oper->next_oper;
        free(cur_oper);
        cur_oper = NULL;
        cur_oper = next_oper;
    }
    return QSUCCESS;
}
```

3.7 XC Functionals

Users can use the following API to add different XC functionals:

83 ⟨*OpenRSP.c* 17a⟩+≡

```
/* <function name='OpenRSPAddXCFun' author='Bin Gao' date='2015-06-23'>
     Add an XC functional to the Hamiltonian
     <param name='open_rsp' direction='inout'>
       The context of response theory calculations
     </param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels that can act on the
       XC functional
     </param>
     <param name='pert_labels' direction='in'>
       All the different perturbation labels involved
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different labels
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback function context
     </param>
     <param name='get_xc_fun_mat' direction='in'>
       User-specified callback function to calculate integral matrices of
       XC functional as well as its derivatives with respect to
       different perturbations
     </param>
     <param name='get_xc_fun_exp' direction='in'>
       User-specified callback function to calculate expectation values of
       XC functional as well as its derivatives with respect to
       different perturbations
     </param>
     <return>Error information</return>
   </function> */
QErrorCode OpenRSPAddXCFun(OpenRSP *open_rsp,
                           const QInt num_pert_lab,
                           const QcPertInt *pert_labels,
                           const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                           void *user_ctx,
#endif
                           const GetXCFunMat get_xc_fun_mat,
                           const GetXCFunExp get_xc_fun_exp)
{
    QErrorCode ierr;  /* error information */
    /* creates the linked list of XC functionals */
    if (open_rsp->xc_fun==NULL) {
        ierr = RSPXCFunCreate(&open_rsp->xc_fun,
                              num_pert_lab,
                              pert_labels,
                              pert_max_orders,
```

```
#if defined(OPENRSP_C_USER_CONTEXT)
                              user_ctx,
#endif
                              get_xc_fun_mat,
                              get_xc_fun_exp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPXCFunCreate()");
    }
    /* adds the XC functional to the linked list */
    else {
        ierr = RSPXCFunAdd(open_rsp->xc_fun,
                           num_pert_lab,
                           pert_labels,
                           pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                           user_ctx,
#endif
                           get_xc_fun_mat,
                           get_xc_fun_exp);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPXCFunAdd()");
    }
    return QSUCCESS;
}
```

The following header file defines all quantities we need for XC functionals. Types `GetXCFunMat` and `GetXCFunpExp` define the requirements of two callback functions from the host program to calculate respectively the integral matrices and expectation values of an XC functional and its derivatives.

84 ⟨*RSPXCFun.h* 84⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

  <header name='RSPXCFun.h' author='Bin Gao' date='2014-08-06'>
    The header file of XC functionals used inside OpenRSP
  </header>
*/

#if !defined(RSP_XCFUN_H)
#define RSP_XCFUN_H

#include "qcmatrix.h"
#include "RSPPerturbation.h"

typedef void (*GetXCFunMat)(const QInt,
                            const QcPertInt*,
                            const QInt,
                            const QInt,
                            const QInt*,
                            const QInt,
                            QcMat*[],
#if defined(OPENRSP_C_USER_CONTEXT)
                            void*,
#endif
                            const QInt,
```

```
                                QcMat*[]);
  typedef void (*GetXCFunExp)(const QInt,
                                const QcPertInt*,
                                const QInt,
                                const QInt,
                                const QInt*,
                                const QInt,
                                QcMat*[],
  #if defined(OPENRSP_C_USER_CONTEXT)
                                void*,
  #endif
                                const QInt,
                                QReal*);
```

⟨*RSPXCFunStruct* 85a⟩

⟨*RSPXCFunAPIs* 85b⟩

```
  #endif
```

Here we use a linked list for the context of XC functionals:

85a ⟨*RSPXCFunStruct* 85a⟩≡

```
  typedef struct RSPXCFun RSPXCFun;
  struct RSPXCFun {
      QInt num_pert_lab;           /* number of different perturbation labels
                                      that can act as perturbations on the
                                      XC functional */
      QInt xc_len_tuple;           /* length of perturbation tuple on the
                                      XC functional, only used for
                                      callback functions */
      QInt *pert_max_orders;       /* allowed maximal order of a perturbation
                                      described by exactly one of these
                                      different labels */
      QcPertInt *pert_labels;      /* all the different perturbation labels */
      QcPertInt *xc_pert_tuple;    /* perturbation tuple on the XC functional,
                                      only used for callback functions */
  #if defined(OPENRSP_C_USER_CONTEXT)
      void *user_ctx;              /* user-defined callbac-kfunction context */
  #endif
      GetXCFunMat get_xc_fun_mat;  /* user-specified function for calculating
                                      integral matrices */
      GetXCFunExp get_xc_fun_exp;  /* user-specified function for calculating
                                      expectation values */
      RSPXCFun *next_xc;           /* pointer to the next XC functional */
  };
```

and the functions related to the XC functionals:

85b ⟨*RSPXCFunAPIs* 85b⟩≡

```
  extern QErrorCode RSPXCFunCreate(RSPXCFun**,
                                   const QInt,
                                   const QcPertInt*,
                                   const QInt*,
  #if defined(OPENRSP_C_USER_CONTEXT)
                                   void*,
```

```
#endif
                                  const GetXCFunMat,
                                  const GetXCFunExp);
extern QErrorCode RSPXCFunAdd(RSPXCFun*,
                              const QInt,
                              const QcPertInt*,
                              const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                              void*,
#endif
                              const GetXCFunMat,
                              const GetXCFunExp);
extern QErrorCode RSPXCFunAssemble(RSPXCFun*,const RSPPert*);
extern QErrorCode RSPXCFunWrite(RSPXCFun*,FILE*);
extern QErrorCode RSPXCFunGetMat(RSPXCFun*,
                                 const QInt,
                                 const QcPertInt*,
                                 const QInt,
                                 const QInt,
                                 const QInt*,
                                 const QInt,
                                 QcMat*[],
                                 const QInt,
                                 QcMat*[]);
extern QErrorCode RSPXCFunGetExp(RSPXCFun*,
                                 const QInt,
                                 const QcPertInt*,
                                 const QInt,
                                 const QInt,
                                 const QInt*,
                                 const QInt,
                                 QcMat*[],
                                 const QInt,
                                 QReal*);
extern QErrorCode RSPXCFunDestroy(RSPXCFun**);
```

The functions are implemented as follows:

86 ⟨*RSPXCFun.c* 86⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩
*/

#include "RSPXCFun.h"

/* <function name='RSPXCFunCreate'
             attr='private'
             author='Bin Gao'
             date='2015-06-23'>
     Create a node of a linked list for a given XC functional, should
     be called at first
     <param name='xc_fun' direction='inout'>
       The linked list of XC functionals
     </param>
```

```
      <param name='num_pert_lab' direction='in'>
        Number of all different perturbation labels that can act as
        perturbations on the XC functional
      </param>
      <param name='pert_labels' direction='in'>
        All the different perturbation labels
      </param>
      <param name='pert_max_orders' direction='in'>
        Allowed maximal order of a perturbation described by exactly one of
        the above different labels
      </param>
      <param name='user_ctx' direction='in'>
        User-defined callback-function context
      </param>
      <param name='get_xc_fun_mat' direction='in'>
        User-specified function for calculating integral matrices of the
        XC functional and its derivatives
      </param>
      <param name='get_xc_fun_exp' direction='in'>
        User-specified function for calculating expectation values of the
        XC functional and its derivatives
      </param>
      <return>Error information</return>
   </function> */
QErrorCode RSPXCFunCreate(RSPXCFun **xc_fun,
                          const QInt num_pert_lab,
                          const QcPertInt *pert_labels,
                          const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                          void *user_ctx,
#endif
                          const GetXCFunMat get_xc_fun_mat,
                          const GetXCFunExp get_xc_fun_exp)
{
    RSPXCFun *new_xc;  /* new XC functional */
    QInt ilab;         /* incremental recorders over perturbation labels */
    QInt jlab;
    new_xc = (RSPXCFun *)malloc(sizeof(RSPXCFun));
    if (new_xc==NULL) {
        QErrorExit(FILE_AND_LINE, "allocates memory for XC functional");
    }
    if (num_pert_lab<0) {
        printf("RSPXCFunCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        QErrorExit(FILE_AND_LINE, "invalid number of perturbation labels");
    }
    else if (num_pert_lab>OPENRSP_PERT_LABEL_MAX) {
        printf("RSPXCFunCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        printf("RSPXCFunCreate>> maximal value for pert. labels %"QCPERTINT_FMT"\n",
               OPENRSP_PERT_LABEL_MAX);
        QErrorExit(FILE_AND_LINE, "too many perturbation labels");
```

```
}
new_xc->num_pert_lab = num_pert_lab;
if (new_xc->num_pert_lab>0) {
    new_xc->pert_max_orders = (QInt *)malloc(num_pert_lab*sizeof(QInt));
    if (new_xc->pert_max_orders==NULL) {
        printf("RSPXCFunCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        QErrorExit(FILE_AND_LINE, "allocates memory for allowed maximal orders");
    }
    new_xc->pert_labels = (QcPertInt *)malloc(num_pert_lab*sizeof(QcPertInt));
    if (new_xc->pert_labels==NULL) {
        printf("RSPXCFunCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        QErrorExit(FILE_AND_LINE, "allocates memory for perturbation labels");
    }
    new_xc->xc_len_tuple = 0;
    for (ilab=0; ilab<num_pert_lab; ilab++) {
        if (pert_labels[ilab]>OPENRSP_PERT_LABEL_MAX) {
            printf("RSPXCFunCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
                   ilab,
                   pert_labels[ilab]);
            printf("RSPXCFunCreate>> maximal value for pert. labels %"QCPERTINT_FMT"\n"
                   OPENRSP_PERT_LABEL_MAX);
            QErrorExit(FILE_AND_LINE, "invalid perturbation label");
        }
        /* each element of <pert_labels> should be unique */
        for (jlab=0; jlab<ilab; jlab++) {
            if (pert_labels[jlab]==pert_labels[ilab]) {
                printf("RSPXCFunCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n"
                       jlab,
                       pert_labels[jlab]);
                printf("RSPXCFunCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n"
                       ilab,
                       pert_labels[ilab]);
                QErrorExit(FILE_AND_LINE, "repeated perturbation labels not allowed");
            }
        }
        new_xc->pert_labels[ilab] = pert_labels[ilab];
        if (pert_max_orders[ilab]<1) {
            printf("RSPXCFunCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\n",
                   ilab,
                   pert_labels[ilab]);
            printf("RSPXCFunCreate>> allowed maximal order is %"QINT_FMT"\n",
                   pert_max_orders[ilab]);
            QErrorExit(FILE_AND_LINE, "only positive order allowed");
        }
        new_xc->pert_max_orders[ilab] = pert_max_orders[ilab];
        new_xc->xc_len_tuple += pert_max_orders[ilab];
    }
    new_xc->xc_pert_tuple = (QcPertInt *)malloc(new_xc->xc_len_tuple*sizeof(QcPertInt))
    if (new_xc->xc_pert_tuple==NULL) {
        printf("RSPXCFunCreate>> length of perturbation tuple %"QINT_FMT"\n",
```

```
                    new_xc->xc_len_tuple);
            QErrorExit(FILE_AND_LINE, "allocates memory for pert. tuple on XC functional");
        }
    }
    else {
        new_xc->pert_max_orders = NULL;
        new_xc->pert_labels = NULL;
        new_xc->xc_pert_tuple = NULL;
    }
#if defined(OPENRSP_C_USER_CONTEXT)
    new_xc->user_ctx = user_ctx;
#endif
    new_xc->get_xc_fun_mat = get_xc_fun_mat;
    new_xc->get_xc_fun_exp = get_xc_fun_exp;
    new_xc->next_xc = NULL;
    *xc_fun = new_xc;
    return QSUCCESS;
}
```

As shown here, we allow for an XC functional that does not depend on any peraturbation—`num_pert_lab==0`, i.e. any perturbed integral matrix and expectation value of this XC functional is zero.

89 ⟨*RSPXCFun.c* 86⟩+≡

```
/* <function name='RSPXCFunAdd'
             attr='private'
             author='Bin Gao'
             date='2015-06-23'>
     Add a given XC functional to the linked list
     <param name='xc_fun' direction='inout'>
       The linked list of XC functionals
     </param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels that can act as
       perturbations on the XC functional
     </param>
     <param name='pert_labels' direction='in'>
       All the different perturbation labels
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different labels
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback-function context
     </param>
     <param name='get_xc_fun_mat' direction='in'>
       User-specified function for calculating integral matrices of the
       XC functional and its derivatives
     </param>
     <param name='get_xc_fun_exp' direction='in'>
       User-specified function for calculating expectation values of the
       XC functional and its derivatives
```

```
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPXCFunAdd(RSPXCFun *xc_fun,
                       const QInt num_pert_lab,
                       const QcPertInt *pert_labels,
                       const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                       void *user_ctx,
#endif
                       const GetXCFunMat get_xc_fun_mat,
                       const GetXCFunExp get_xc_fun_exp)
{
    RSPXCFun *new_xc;  /* new XC functional */
    RSPXCFun *cur_xc;  /* current XC functional */
    QErrorCode ierr;   /* error information */
    /* creates the new XC functional */
    ierr = RSPXCFunCreate(&new_xc,
                          num_pert_lab,
                          pert_labels,
                          pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                          user_ctx,
#endif
                          get_xc_fun_mat,
                          get_xc_fun_exp);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPXCFunCreate()");
    /* walks to the last XC functional */
    cur_xc = xc_fun;
    while (cur_xc->next_xc!=NULL) {
        cur_xc = cur_xc->next_xc;
    }
    /* inserts the new XC functional to the tail of the linked list */
    cur_xc->next_xc = new_xc;
    return QSUCCESS;
}

/* <function name='RSPXCFunAssemble'
             attr='private'
             author='Bin Gao'
             date='2015-06-23'>
     Assembles the linked list of XC functionals
     <param name='xc_fun' direction='inout'>
       The linked list of XC functionals
     </param>
     <param name='rsp_pert' direction='in'>
       The context of perturbations
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPXCFunAssemble(RSPXCFun *xc_fun, const RSPPert *rsp_pert)
{
```

```
    QInt ixc;          /* incremental recorder over XC functionals */
    RSPXCFun *cur_xc;  /* current XC functional */
    QErrorCode ierr;   /* error information */
    ixc = 0;
    cur_xc = xc_fun;
    do {
        if (cur_xc->num_pert_lab>0 &&
            (cur_xc->pert_labels==NULL || cur_xc->pert_max_orders==NULL)) {
            printf("RSPXCFunAssemble>> %"QINT_FMT"-th XC functional\n",
                   ixc);
            QErrorExit(FILE_AND_LINE, "perturbations of XC functional not set");
        }
        if (cur_xc->get_xc_fun_mat==NULL || cur_xc->get_xc_fun_exp==NULL) {
            printf("RSPXCFunAssemble>> %"QINT_FMT"-th XC functional\n",
                   ixc);
            QErrorExit(FILE_AND_LINE, "callback functions of XC functional not set");
        }
        /* checks perturbation labels and allowed maximal orders against
           all known perturbations */
        ierr = RSPPertValidateLabelOrder(rsp_pert,
                                         cur_xc->num_pert_lab,
                                         cur_xc->pert_labels,
                                         cur_xc->pert_max_orders);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertValidateLabelOrder()");
        /* moves to the next XC functional */
        ixc++;
        cur_xc = cur_xc->next_xc;
    } while (cur_xc!=NULL);
    return QSUCCESS;
}

/* <function name='RSPXCFunWrite'
             attr='private'
             author='Bin Gao'
             date='2015-06-23'>
     Writes the linked list of XC functionals
     <param name='xc_fun' direction='in'>
       The linked list of XC functionals
     </param>
     <param name='fp_xc' direction='inout'>File pointer</param>
     <return>Error information</return>
   </function> */
QErrorCode RSPXCFunWrite(RSPXCFun *xc_fun, FILE *fp_xc)
{
    QInt ixc;          /* incremental recorder over XC functionals */
    RSPXCFun *cur_xc;  /* current XC functional */
    QInt ilab;         /* incremental recorder over perturbation labels */
    ixc = 0;
    cur_xc = xc_fun;
    do {
        fprintf(fp_xc, "RSPXCFunWrite>> XC functional %"QINT_FMT"\n", ixc);
        fprintf(fp_xc,
```

```
                "RSPXCFunWrite>> number of pert. labels that XC functional depends on %"QIN
                cur_xc->num_pert_lab);
        fprintf(fp_xc, "RSPXCFunWrite>> label           maximum-order\n");
        for (ilab=0; ilab<cur_xc->num_pert_lab; ilab++) {
            fprintf(fp_xc,
                    "RSPXCFunWrite>>       %"QCPERTINT_FMT"                  %"QINT_FMT"\n"
                    cur_xc->pert_labels[ilab],
                    cur_xc->pert_max_orders[ilab]);
        }
#if defined(OPENRSP_C_USER_CONTEXT)
        if (cur_xc->user_ctx!=NULL) {
            fprintf(fp_xc, "RSPXCFunWrite>> user-defined function context given\n");
        }
#endif
        /* moves to the next XC functional */
        ixc++;
        cur_xc = cur_xc->next_xc;
    } while (cur_xc!=NULL);
    return QSUCCESS;
}

/* <function name='RSPXCFunGetMat'
             attr='private'
             author='Bin Gao'
             date='2015-10-15'>
     Calculates integral matrices of the linked list of XC functionals
     <param name='xc_fun' direction='inout'>
       The linked list of XC functionals
     </param>
     <param name='xc_len_tuple' direction='in'>
       Length of the perturbation tuple on the linked list of XC functionals
     </param>
     <param name='xc_pert_tuple' direction='in'>
       Perturbation tuple on the linked list of XC functionals
     </param>
     <param name='num_freq_configs' direction='in'>
       The number of different frequency configurations to be considered for
       the perturbation tuple
     </param>
     <param name='dmat_num_tuple' direction='in'>
       The number of different perturbation tuples of the atomic orbital (AO)
       based density matrices passed
     </param>
     <param name='dmat_idx_tuple' direction='in'>
       Indices of the density matrix perturbation tuples passed (canonically
       ordered)
     </param>
     <param name='num_dmat' direction='in'>
       Number of collected AO based density matrices for the passed density
       matrix perturbation tuples and all frequency configurations
     </param>
     <param name='dens_mat' direction='in'>
```

```
       The collected AO based density matrices
     </param>
     <param name='num_int' direction='in'>
       Number of the integral matrices
     </param>
     <param name='val_int' direction='inout'>
       The integral matrices
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPXCFunGetMat(RSPXCFun *xc_fun,
                          const QInt xc_len_tuple,
                          const QcPertInt *xc_pert_tuple,
                          const QInt num_freq_configs,
                          const QInt dmat_num_tuple,
                          const QInt *dmat_idx_tuple,
                          const QInt num_dmat,
                          QcMat *dens_mat[],
                          const QInt num_int,
                          QcMat *val_int[])
{
    RSPXCFun *cur_xc;  /* current XC functional */
    QErrorCode ierr;   /* error information */
    cur_xc = xc_fun;
    do {
        /* gets the host program's perturbation tuple on the XC functional */
        ierr = RSPPertInternTupleToHostTuple(xc_len_tuple,
                                             xc_pert_tuple,
                                             cur_xc->num_pert_lab,
                                             cur_xc->pert_labels,
                                             cur_xc->pert_max_orders,
                                             &cur_xc->xc_len_tuple,
                                             cur_xc->xc_pert_tuple);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostTuple()");
        /* checks if the perturbations on the XC functional result in
           zero values */
        if (cur_xc->xc_len_tuple<0) continue;
        /* calculates integral matrices using the callback function */
        cur_xc->get_xc_fun_mat(cur_xc->xc_len_tuple,
                               cur_xc->xc_pert_tuple,
                               num_freq_configs,
                               dmat_num_tuple,
                               dmat_idx_tuple,
                               num_dmat,
                               dens_mat,
#if defined(OPENRSP_C_USER_CONTEXT)
                               cur_xc->user_ctx,
#endif
                               num_int,
                               val_int);
        /* moves to the next XC functional */
        cur_xc = cur_xc->next_xc;
```

```
    } while (cur_xc!=NULL);
    return QSUCCESS;
}

/* <function name='RSPXCFunGetExp'
             attr='private'
             author='Bin Gao'
             date='2015-10-15'>
     Calculates expectation values of the linked list of XC functionals
     <param name='xc_fun' direction='inout'>
       The linked list of XC functionals
     </param>
     <param name='xc_len_tuple' direction='in'>
       Length of the perturbation tuple on the linked list of XC functionals
     </param>
     <param name='xc_pert_tuple' direction='in'>
       Perturbation tuple on the linked list of XC functionals
     </param>
     <param name='num_freq_configs' direction='in'>
       The number of different frequency configurations to be considered for
       the perturbation tuple
     </param>
     <param name='dmat_num_tuple' direction='in'>
       The number of different perturbation tuples of the atomic orbital (AO)
       based density matrices passed
     </param>
     <param name='dmat_idx_tuple' direction='in'>
       Indices of the density matrix perturbation tuples passed (canonically
       ordered)
     </param>
     <param name='num_dmat' direction='in'>
       Number of collected AO based density matrices for the passed density
       matrix perturbation tuples and all frequency configurations
     </param>
     <param name='dens_mat' direction='in'>
       The collected AO based density matrices
     </param>
     <param name='num_exp' direction='in'>
       Number of the expectation values
     </param>
     <param name='val_exp' direction='inout'>
       The expectation values
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPXCFunGetExp(RSPXCFun *xc_fun,
                          const QInt xc_len_tuple,
                          const QcPertInt *xc_pert_tuple,
                          const QInt num_freq_configs,
                          const QInt dmat_num_tuple,
                          const QInt *dmat_idx_tuple,
                          const QInt num_dmat,
```

```
                          QcMat *dens_mat[],
                          const QInt num_exp,
                          QReal *val_exp)
{
    RSPXCFun *cur_xc;  /* current XC functional */
    QErrorCode ierr;   /* error information */
    cur_xc = xc_fun;
    do {
        /* gets the host program's perturbation tuple on the XC functional */
        ierr = RSPPertInternTupleToHostTuple(xc_len_tuple,
                                             xc_pert_tuple,
                                             cur_xc->num_pert_lab,
                                             cur_xc->pert_labels,
                                             cur_xc->pert_max_orders,
                                             &cur_xc->xc_len_tuple,
                                             cur_xc->xc_pert_tuple);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostTuple()");
        /* checks if the perturbations on the XC functional result in
           zero values */
        if (cur_xc->xc_len_tuple<0) continue;
        /* calculates expectation values using the callback function */
        cur_xc->get_xc_fun_exp(cur_xc->xc_len_tuple,
                               cur_xc->xc_pert_tuple,
                               num_freq_configs,
                               dmat_num_tuple,
                               dmat_idx_tuple,
                               num_dmat,
                               dens_mat,
#if defined(OPENRSP_C_USER_CONTEXT)
                               cur_xc->user_ctx,
#endif
                               num_exp,
                               val_exp);
        /* moves to the next XC functional */
        cur_xc = cur_xc->next_xc;
    } while (cur_xc!=NULL);
    return QSUCCESS;
}

/* <function name='RSPXCFunDestroy'
             attr='private'
             author='Bin Gao'
             date='2015-06-23'>
     Destroys the linked list of XC functionals, should be called at the end
     <param name='xc_fun' direction='inout'>
       The linked list of XC functionals
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPXCFunDestroy(RSPXCFun **xc_fun)
{
    RSPXCFun *cur_xc;   /* current XC functional */
```

```
    RSPXCFun *next_xc;  /* next XC functional */
    /* walks to the last XC functional */
    cur_xc = *xc_fun;
    while (cur_xc!=NULL) {
        if (cur_xc->pert_max_orders!=NULL) {
            free(cur_xc->pert_max_orders);
            cur_xc->pert_max_orders = NULL;
        }
        if (cur_xc->pert_labels!=NULL) {
            free(cur_xc->pert_labels);
            cur_xc->pert_labels = NULL;
        }
        if (cur_xc->xc_pert_tuple!=NULL) {
            free(cur_xc->xc_pert_tuple);
            cur_xc->xc_pert_tuple = NULL;
        }
#if defined(OPENRSP_C_USER_CONTEXT)
        cur_xc->user_ctx = NULL;
#endif
        cur_xc->get_xc_fun_mat = NULL;
        cur_xc->get_xc_fun_exp = NULL;
        next_xc = cur_xc->next_xc;
        free(cur_xc);
        cur_xc = NULL;
        cur_xc = next_xc;
    }
    return QSUCCESS;
}
```

3.8 Nuclear Hamiltonian

Users can use the following API to set nuclear Hamiltonian (nuclear repulsion and nuclei-field interaction):

97 ⟨*OpenRSP.c* 17a⟩+≡

```
/* <function name='OpenRSPSetNucHamilton' author='Bin Gao' date='2015-02-12'>
     Set the context of nuclear Hamiltonian
     <param name='open_rsp' direction='inout'>
       The context of response theory calculations
     </param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels that can act on the
       nuclear Hamiltonian
     </param>
     <param name='pert_labels' direction='in'>
       All the different perturbation labels involved
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different labels
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback function context
     </param>
     <param name='get_nuc_contrib' direction='in'>
       User-specified callback function to calculate nuclear contributions
     </param>
     <param name='num_atoms' direction='in'>
       Number of atoms
     </param>
     <return>Error information</return>
   </function> */
QErrorCode OpenRSPSetNucHamilton(OpenRSP *open_rsp,
                                 const QInt num_pert_lab,
                                 const QcPertInt *pert_labels,
                                 const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                                 void *user_ctx,
#endif

                                 const GetNucContrib get_nuc_contrib,
/*FIXME: num_atoms to be removed after perturbation free scheme implemented*/
                                 const QInt num_atoms)
{
    QErrorCode ierr;  /* error information */
    /* creates the context of nuclear Hamiltonian */
    if (open_rsp->nuc_hamilton!=NULL) {
        ierr = RSPNucHamiltonDestroy(open_rsp->nuc_hamilton);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPNucHamiltonDestroy()");
    }
    else {
        open_rsp->nuc_hamilton = (RSPNucHamilton *)malloc(sizeof(RSPNucHamilton));
```

```
        if (open_rsp->nuc_hamilton==NULL) {
            QErrorExit(FILE_AND_LINE, "allocates memory for nuclear Hamiltonian");
        }
    }
    ierr = RSPNucHamiltonCreate(open_rsp->nuc_hamilton,
                                num_pert_lab,
                                pert_labels,
                                pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                                user_ctx,
#endif
                                get_nuc_contrib,
/*FIXME: num_atoms to be removed after perturbation free scheme implemented*/
                                num_atoms);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPNucHamiltonCreate()");
    return QSUCCESS;
}
```

The header file of the nuclear Hamiltonian is:
The following header file defines all quantities we need for the nuclear Hamiltonian. Type `GetNucContrib` defines the requirements of the host-program's callback function to calculate the contribution of nuclear Hamiltonian and its derivatives.

98 ⟨*RSPNucHamilton.h* 98⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

  <header name='RSPNucHamilton.h' author='Bin Gao' date='2014-12-11'>
    The header file of nuclear Hamiltonian used inside OpenRSP
  </header>
*/

#if !defined(RSP_NUCHAMILTON_H)
#define RSP_NUCHAMILTON_H

#include "qcmatrix.h"
#include "RSPPerturbation.h"

typedef void (*GetNucContrib)(const QInt,
                              const QcPertInt*,
                              const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                              void*,
#endif
                              const QInt,
                              QReal*);

⟨RSNucHamiltonStruct 99a⟩

⟨RSPNucHamiltonAPIs 99b⟩

#endif
```

The context of nuclear Hamiltonian is:

99a ⟨*RSNucHamiltonStruct* 99a⟩≡

```
typedef struct {
    QInt num_pert_lab;              /* number of different perturbation labels
                                       that can act as perturbations on the
                                       nuclear Hamiltonian */
    QInt nuc_num_pert;              /* number of perturbations on the
                                       nuclear Hamiltonian, only used for
                                       callback functions */
    QInt *pert_max_orders;          /* allowed maximal order of a perturbation
                                       described by exactly one of these
                                       different labels */
    QInt *nuc_pert_orders;          /* orders of perturbations on the
                                       nuclear Hamiltonian, only used for
                                       callback functions */
    QcPertInt *pert_labels;         /* all the different perturbation labels */
    QcPertInt *nuc_pert_labels;     /* labels of perturbations on the
                                       nuclear Hamiltonian, only used for
                                       callback functions */
#if defined(OPENRSP_C_USER_CONTEXT)
    void *user_ctx;                 /* user-defined callback-function context */
#endif
    GetNucContrib get_nuc_contrib;  /* user-specified function for calculating
                                       contribution from the nuclear Hamiltonian */
/*FIXME: num_atoms to be removed after perturbation free scheme implemented*/
    QInt num_atoms;
} RSPNucHamilton;
```

and the functions related to the nuclear Hamiltonian:

99b ⟨*RSPNucHamiltonAPIs* 99b⟩≡

```
extern QErrorCode RSPNucHamiltonCreate(RSPNucHamilton*,
                                       const QInt,
                                       const QcPertInt*,
                                       const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                       void*,
#endif
                                       const GetNucContrib,
/*FIXME: num_atoms to be removed after perturbation free scheme implemented*/
                                       const QInt);
extern QErrorCode RSPNucHamiltonAssemble(RSPNucHamilton*,const RSPPert*);
extern QErrorCode RSPNucHamiltonWrite(const RSPNucHamilton*,FILE*);
extern QErrorCode RSPNucHamiltonGetContributions(RSPNucHamilton*,
                                                 const QInt,
                                                 const QcPertInt*,
                                                 const QInt,
                                                 QReal*);
extern QErrorCode RSPNucHamiltonDestroy(RSPNucHamilton*);
/*FIXME: RSPNucHamiltonGetNumAtoms() to be removed after perturbation free scheme implement
extern QErrorCode RSPNucHamiltonGetNumAtoms(const RSPNucHamilton*,QInt*);
```

The functions are implemented as follows:

99c ⟨*RSPNucHamilton.c* 99c⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩
```

```
*/

#include "RSPNucHamilton.h"

/* <function name='RSPNucHamiltonCreate'
             attr='private'
             author='Bin Gao'
             date='2015-02-12'>
     Create the context of nuclear Hamiltonian, should be called at first
     <param name='nuc_hamilton' direction='inout'>
       The context of nuclear Hamiltonian
     </param>
     <param name='num_pert_lab' direction='in'>
       Number of all different perturbation labels that can act as
       perturbations on the nuclear Hamiltonian
     </param>
     <param name='pert_labels' direction='in'>
       All the different perturbation labels
     </param>
     <param name='pert_max_orders' direction='in'>
       Allowed maximal order of a perturbation described by exactly one of
       the above different labels
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback-function context
     </param>
     <param name='get_nuc_contrib' direction='in'>
       User-specified function for calculating contribution of the
       nuclear Hamiltonian and its derivatives
     </param>
     <param name='num_atoms' direction='in'>
       Number of atoms
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPNucHamiltonCreate(RSPNucHamilton *nuc_hamilton,
                                const QInt num_pert_lab,
                                const QcPertInt *pert_labels,
                                const QInt *pert_max_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                                void *user_ctx,
#endif
                                const GetNucContrib get_nuc_contrib,
/*FIXME: num_atoms to be removed after perturbation free scheme implemented*/
                                const QInt num_atoms)
{
    QInt ilab;  /* incremental recorders over perturbation labels */
    QInt jlab;
    if (num_pert_lab<0) {
        printf("RSPNucHamiltonCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        QErrorExit(FILE_AND_LINE, "invalid number of perturbation labels");
```

```
}
else if (num_pert_lab>OPENRSP_PERT_LABEL_MAX) {
    printf("RSPNucHamiltonCreate>> number of perturbation labels %"QINT_FMT"\n",
           num_pert_lab);
    printf("RSPNucHamiltonCreate>> maximal value for pert. labels %"QCPERTINT_FMT"\n",
           OPENRSP_PERT_LABEL_MAX);
    QErrorExit(FILE_AND_LINE, "too many perturbation labels");
}
nuc_hamilton->num_pert_lab = num_pert_lab;
if (nuc_hamilton->num_pert_lab>0) {
    nuc_hamilton->pert_max_orders = (QInt *)malloc(num_pert_lab*sizeof(QInt));
    if (nuc_hamilton->pert_max_orders==NULL) {
        printf("RSPNucHamiltonCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        QErrorExit(FILE_AND_LINE, "allocates memory for allowed maximal orders");
    }
    nuc_hamilton->nuc_pert_orders = (QInt *)malloc(num_pert_lab*sizeof(QInt));
    if (nuc_hamilton->nuc_pert_orders==NULL) {
        printf("RSPNucHamiltonCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        QErrorExit(FILE_AND_LINE, "allocates memory for pert. orders on nuclear Hamilto
    }
    nuc_hamilton->pert_labels = (QcPertInt *)malloc(num_pert_lab*sizeof(QcPertInt));
    if (nuc_hamilton->pert_labels==NULL) {
        printf("RSPNucHamiltonCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        QErrorExit(FILE_AND_LINE, "allocates memory for perturbation labels");
    }
    nuc_hamilton->nuc_pert_labels = (QcPertInt *)malloc(num_pert_lab*sizeof(QcPertInt))
    if (nuc_hamilton->nuc_pert_labels==NULL) {
        printf("RSPNucHamiltonCreate>> number of perturbation labels %"QINT_FMT"\n",
               num_pert_lab);
        QErrorExit(FILE_AND_LINE, "allocates memory for pert. labels on nuclear Hamilto
    }
    for (ilab=0; ilab<num_pert_lab; ilab++) {
        if (pert_labels[ilab]>OPENRSP_PERT_LABEL_MAX) {
            printf("RSPNucHamiltonCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\
                   ilab,
                   pert_labels[ilab]);
            printf("RSPNucHamiltonCreate>> maximal value for pert. labels %"QCPERTINT_F
                   OPENRSP_PERT_LABEL_MAX);
            QErrorExit(FILE_AND_LINE, "invalid perturbation label");
        }
        /* each element of <pert_labels> should be unique */
        for (jlab=0; jlab<ilab; jlab++) {
            if (pert_labels[jlab]==pert_labels[ilab]) {
                printf("RSPNucHamiltonCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_F
                       jlab,
                       pert_labels[jlab]);
                printf("RSPNucHamiltonCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_F
                       ilab,
                       pert_labels[ilab]);
```

```
                    QErrorExit(FILE_AND_LINE, "repeated perturbation labels not allowed");
                }
            }
            nuc_hamilton->pert_labels[ilab] = pert_labels[ilab];
            if (pert_max_orders[ilab]<1) {
                printf("RSPNucHamiltonCreate>> %"QINT_FMT"-th pert. label %"QCPERTINT_FMT"\
                       ilab,
                       pert_labels[ilab]);
                printf("RSPNucHamiltonCreate>> allowed maximal order is %"QINT_FMT"\n",
                       pert_max_orders[ilab]);
                QErrorExit(FILE_AND_LINE, "only positive order allowed");
            }
            nuc_hamilton->pert_max_orders[ilab] = pert_max_orders[ilab];
        }
    }
    else {
        nuc_hamilton->pert_max_orders = NULL;
        nuc_hamilton->nuc_pert_orders = NULL;
        nuc_hamilton->pert_labels = NULL;
        nuc_hamilton->nuc_pert_labels = NULL;
    }
#if defined(OPENRSP_C_USER_CONTEXT)
    nuc_hamilton->user_ctx = user_ctx;
#endif
    nuc_hamilton->get_nuc_contrib = get_nuc_contrib;
/*FIXME: num_atoms to be removed after perturbation free scheme implemented*/
    nuc_hamilton->num_atoms = num_atoms;
    return QSUCCESS;
}
```

As shown here, we allow for an nuclear Hamiltonian that does not depend on any peraturbation—`num_pert_lab==0`, i.e. any perturbed contribution of this nuclear Hamiltonian is zero.

102 ⟨*RSPNucHamilton.c* 99c⟩+≡

```
/* <function name='RSPNucHamiltonAssemble'
             attr='private'
             author='Bin Gao'
             date='2015-02-12'>
     Assembles the context of nuclear Hamiltonian
     <param name='nuc_hamilton' direction='inout'>
       The context of nuclear Hamiltonian
     </param>
     <param name='rsp_pert' direction='in'>
       The context of perturbations
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPNucHamiltonAssemble(RSPNucHamilton *nuc_hamilton,
                                  const RSPPert *rsp_pert)
{
    QErrorCode ierr;  /* error information */
    if (nuc_hamilton->num_pert_lab>0 &&
        (nuc_hamilton->pert_labels==NULL || nuc_hamilton->pert_max_orders==NULL)) {
```

```
        QErrorExit(FILE_AND_LINE, "perturbations of nuclear Hamiltonian not set");
    }
    if (nuc_hamilton->get_nuc_contrib==NULL) {
        QErrorExit(FILE_AND_LINE, "callback function of nuclear Hamiltonian not set");
    }
    /* checks perturbation labels and allowed maximal orders against
       all known perturbations */
    ierr = RSPPertValidateLabelOrder(rsp_pert,
                                     nuc_hamilton->num_pert_lab,
                                     nuc_hamilton->pert_labels,
                                     nuc_hamilton->pert_max_orders);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertValidateLabelOrder()");
    return QSUCCESS;
}

/* <function name='RSPNucHamiltonWrite'
             attr='private'
             author='Bin Gao'
             date='2015-02-12'>
     Writes the context of nuclear Hamiltonian
     <param name='nuc_hamilton' direction='in'>
       The context of nuclear Hamiltonian
     </param>
     <param name='fp_nuc' direction='inout'>File pointer</param>
     <return>Error information</return>
   </function> */
QErrorCode RSPNucHamiltonWrite(const RSPNucHamilton *nuc_hamilton,
                               FILE *fp_nuc)
{
    QInt ilab;  /* incremental recorder over perturbation labels */
    fprintf(fp_nuc,
            "RSPNucHamiltonWrite>> number of pert. labels that nuclear Hamiltonian depends
            nuc_hamilton->num_pert_lab);
    fprintf(fp_nuc, "RSPNucHamiltonWrite>> label           maximum-order\n");
    for (ilab=0; ilab<nuc_hamilton->num_pert_lab; ilab++) {
        fprintf(fp_nuc,
                "RSPNucHamiltonWrite>>       %"QCPERTINT_FMT"                  %"QINT_FMT"\
                nuc_hamilton->pert_labels[ilab],
                nuc_hamilton->pert_max_orders[ilab]);
    }
#if defined(OPENRSP_C_USER_CONTEXT)
    if (nuc_hamilton->user_ctx!=NULL) {
        fprintf(fp_nuc, "RSPNucHamiltonWrite>> user-defined function context given\n");
    }
#endif
/*FIXME: num_atoms to be removed after perturbation free scheme implemented*/
    fprintf(fp_nuc,
            "RSPNucHamiltonWrite>> number of atoms %"QINT_FMT"\n",
            nuc_hamilton->num_atoms);
    return QSUCCESS;
}
```

```
/* <function name='RSPNucHamiltonGetContributions'
             attr='private'
             author='Bin Gao'
             date='2015-10-15'>
     Calculates contribution of the nuclear Hamiltonian
     <param name='nuc_hamilton' direction='inout'>
       The context of nuclear Hamiltonian
     </param>
     <param name='nuc_len_tuple' direction='in'>
       Length of the perturbation tuple on the nuclear Hamiltonian
     </param>
     <param name='nuc_pert_tuple' direction='in'>
       Perturbation tuple on the nuclear Hamiltonian
     </param>
     <param name='size_pert' direction='in'>
       Size of the perturbations on the nuclear Hamiltonian
     </param>
     <param name='val_nuc' direction='inout'>
       The contribution of the nuclear Hamiltonian
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPNucHamiltonGetContributions(RSPNucHamilton *nuc_hamilton,
                                          const QInt nuc_len_tuple,
                                          const QcPertInt *nuc_pert_tuple,
                                          const QInt size_pert,
                                          QReal *val_nuc)
{
    QErrorCode ierr;  /* error information */
    /* gets perturbation labels and corresponding orders out of the internal
       perturbation tuple on the nuclear Hamiltonian */
    ierr = RSPPertInternTupleToHostLabelOrder(nuc_len_tuple,
                                              nuc_pert_tuple,
                                              nuc_hamilton->num_pert_lab,
                                              nuc_hamilton->pert_labels,
                                              nuc_hamilton->pert_max_orders,
                                              &nuc_hamilton->nuc_num_pert,
                                              nuc_hamilton->nuc_pert_labels,
                                              nuc_hamilton->nuc_pert_orders);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPPertInternTupleToHostLabelOrder()");
    /* checks if the perturbations on the nuclear Hamiltonian
       result in zero values */
    if (nuc_hamilton->nuc_num_pert<0) return QSUCCESS;
    /* calculates contribution of nuclear Hamiltonian using the
       callback function */
    nuc_hamilton->get_nuc_contrib(nuc_hamilton->nuc_num_pert,
                                  nuc_hamilton->nuc_pert_labels,
                                  nuc_hamilton->nuc_pert_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                                  nuc_hamilton->user_ctx,
#endif
                                  size_pert,
```

```
                                    val_nuc);
    return QSUCCESS;
}

/*% \brief gets the number of atoms
    \author Bin Gao
    \date 2015-02-12
    \param[RSPNucHamilton:struct]{in} nuc_hamilton the context of nuclear Hamiltonian
    \param[QInt:int]{out} num_atoms number of atoms
    \return[QErrorCode:int] error information
*/
QErrorCode RSPNucHamiltonGetNumAtoms(const RSPNucHamilton *nuc_hamilton,
                                     QInt *num_atoms)
{
    *num_atoms = nuc_hamilton->num_atoms;
    return QSUCCESS;
}

/* <function name='RSPNucHamiltonDestroy'
             attr='private'
             author='Bin Gao'
             date='2015-02-12'>
     Destroys the context of nuclear Hamiltonian, should be called at the end
     <param name='nuc_hamilton' direction='inout'>
       The context of nuclear Hamiltonian
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPNucHamiltonDestroy(RSPNucHamilton *nuc_hamilton)
{
    if (nuc_hamilton->pert_max_orders!=NULL) {
        free(nuc_hamilton->pert_max_orders);
        nuc_hamilton->pert_max_orders = NULL;
    }
    if (nuc_hamilton->nuc_pert_orders!=NULL) {
        free(nuc_hamilton->nuc_pert_orders);
        nuc_hamilton->nuc_pert_orders = NULL;
    }
    if (nuc_hamilton->pert_labels!=NULL) {
        free(nuc_hamilton->pert_labels);
        nuc_hamilton->pert_labels = NULL;
    }
    if (nuc_hamilton->nuc_pert_labels!=NULL) {
        free(nuc_hamilton->nuc_pert_labels);
        nuc_hamilton->nuc_pert_labels = NULL;
    }
#if defined(OPENRSP_C_USER_CONTEXT)
    nuc_hamilton->user_ctx = NULL;
#endif
    nuc_hamilton->get_nuc_contrib = NULL;
    return QSUCCESS;
}
```

3.9 Linear Response Equation Solver

Users can use the following API to set the linear response equation solver:

107a ⟨*OpenRSP.c* 17a⟩+≡

```
/* <function name='OpenRSPSetLinearRSPSolver' author='Bin Gao' date='2014-08-06'>
     Set the context of linear response equation solver
     <param name='open_rsp' direction='inout'>
       The context of response theory calculations
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback function context
     </param>
     <param name='get_linear_rsp_solution' direction='in'>
       User-specified callback function of linear response equation solver
     </param>
     <return>Error information</return>
   </function> */
QErrorCode OpenRSPSetLinearRSPSolver(OpenRSP *open_rsp,
#if defined(OPENRSP_C_USER_CONTEXT)
                                     void *user_ctx,
#endif
                                     const GetLinearRSPSolution get_linear_rsp_solution)
{
    QErrorCode ierr;  /* error information */
    /* creates the context of response equation solver */
    if (open_rsp->rsp_solver!=NULL) {
        ierr = RSPSolverDestroy(open_rsp->rsp_solver);
        QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPSolverDestroy()");
    }
    else {
        open_rsp->rsp_solver = (RSPSolver *)malloc(sizeof(RSPSolver));
        if (open_rsp->rsp_solver==NULL) {
            QErrorExit(FILE_AND_LINE, "allocates memory for solver");
        }
    }
    ierr = RSPSolverCreate(open_rsp->rsp_solver,
#if defined(OPENRSP_C_USER_CONTEXT)
                           user_ctx,
#endif
                           get_linear_rsp_solution);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling RSPSolverCreate()");
    return QSUCCESS;
}
```

The following header file defines all quantities we need for the linear response equation solver. Type `GetLinearRSPSolution` defines the requirements of the callback function of the linear response equation solver.

107b ⟨*RSPSolver.h* 107b⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

  <header name='RSPSolver.h' author='Bin Gao' date='2014-08-06'>
```

```
    The header file of linear response equation solver used inside OpenRSP
  </header>
*/

#if !defined(RSP_SOLVER_H)
#define RSP_SOLVER_H

#include "qcmatrix.h"

typedef void (*GetLinearRSPSolution)(const QInt,
                                     const QReal*,
                                     const QInt,
                                     QcMat*[],
#if defined(OPENRSP_C_USER_CONTEXT)
                                     void*,
#endif
                                     QcMat*[]);
```

⟨*RSPSolverStruct* 108a⟩

⟨*RSPSolverAPIs* 108b⟩

```
#endif
```

The context of linear response equation solver is:

108a ⟨*RSPSolverStruct* 108a⟩≡

```
typedef struct {
#if defined(OPENRSP_C_USER_CONTEXT)
    void *user_ctx;                                  /* user-defined callback-function
                                                        context */
#endif
    GetLinearRSPSolution get_linear_rsp_solution;  /* user-specified function of
                                                        linear response equation solver */
} RSPSolver;
```

and the related functions are:

108b ⟨*RSPSolverAPIs* 108b⟩≡

```
extern QErrorCode RSPSolverCreate(RSPSolver*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                  void*,
#endif
                                  const GetLinearRSPSolution);
extern QErrorCode RSPSolverAssemble(RSPSolver*);
extern QErrorCode RSPSolverWrite(const RSPSolver*,FILE*);
extern QErrorCode RSPSolverGetLinearRSPSolution(const RSPSolver*,
                                                const QInt,
                                                const QReal*,
                                                const QInt,
                                                QcMat*[],
                                                QcMat*[]);
extern QErrorCode RSPSolverDestroy(RSPSolver*);
```

These functions are implemented as follows:

108c ⟨*RSPSolver.c* 108c⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩
*/

#include "RSPSolver.h"

/* <function name='RSPSolverCreate'
             attr='private'
             author='Bin Gao'
             date='2014-08-06'>
     Create the context of response equation solver, should be called at first
     <param name='rsp_solver' direction='inout'>
       The context of response equation solver
     </param>
     <param name='user_ctx' direction='in'>
       User-defined callback function context
     </param>
     <param name='get_linear_rsp_solution' direction='in'>
       User-specified callback function of linear response equation solver
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPSolverCreate(RSPSolver *rsp_solver,
#if defined(OPENRSP_C_USER_CONTEXT)
                           void *user_ctx,
#endif
                           const GetLinearRSPSolution get_linear_rsp_solution)
{
#if defined(OPENRSP_C_USER_CONTEXT)
    rsp_solver->user_ctx = user_ctx;
#endif
    rsp_solver->get_linear_rsp_solution = get_linear_rsp_solution;
    return QSUCCESS;
}

/* <function name='RSPSolverAssemble'
             attr='private'
             author='Bin Gao'
             date='2014-08-06'>
     Assembles the context of response equation solver
     <param name='rsp_solver' direction='inout'>
       The context of response equation solver
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPSolverAssemble(RSPSolver *rsp_solver)
{
/*FIXME: to implement? */
    return QSUCCESS;
}

/* <function name='RSPSolverWrite'
```

```
              attr='private'
              author='Bin Gao'
              date='2014-08-06'>
     Writes the context of response equation solver
     <param name='rsp_solver' direction='in'>
       The context of response equation solver
     </param>
     <param name='fp_solver' direction='inout'>File pointer</param>
     <return>Error information</return>
   </function> */
QErrorCode RSPSolverWrite(const RSPSolver *rsp_solver, FILE *fp_solver)
{
#if defined(OPENRSP_C_USER_CONTEXT)
    if (rsp_solver->user_ctx!=NULL) {
        fprintf(fp_solver, "RSPSolverWrite>> user-defined function context given\n");
    }
#endif
    return QSUCCESS;
}

/* <function name='RSPSolverGetLinearRSPSolution'
              attr='private'
              author='Bin Gao'
              date='2014-08-06'>
     Solve the linear response equation
     <param name='rsp_solver' direction='in'>
       The context of response equation solver
     </param>
     <param name='num_freq_sums' direction='in'>
       Number of complex frequency sums on the left hand side of the linear
       response equation
     </param>
     <param name='freq_sums' direction='in'>
       The complex frequency sums on the left hand side
     </param>
     <param name='size_pert' direction='in'>
       Size of perturbations acting on the time-dependent
       self-consistent-field (TDSCF) equation
     </param>
     <param name='RHS_mat' direction='in'>
       Right-hand-side (RHS) matrices, size is the product of <size_pert>
       and <num_freq_sums>
     <param name='rsp_param' direction='inout'>
       Solved response parameters, size is the product of <size_pert>
       and <num_freq_sums>
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPSolverGetLinearRSPSolution(const RSPSolver *rsp_solver,
                                         const QInt num_freq_sums,
                                         const QReal *freq_sums,
                                         const QInt size_pert,
```

```
                                        QcMat *RHS_mat[],
                                        QcMat *rsp_param[])
{
    rsp_solver->get_linear_rsp_solution(num_freq_sums,
                                        freq_sums,
                                        size_pert,
                                        RHS_mat,
#if defined(OPENRSP_C_USER_CONTEXT)
                                        rsp_solver->user_ctx,
#endif
                                        rsp_param);
    return QSUCCESS;
}

/* <function name='RSPSolverDestroy'
             attr='private'
             author='Bin Gao'
             date='2014-08-05'>
     Destroys the context of response equation solver, should be called at the end
     <param name='rsp_solver' direction='inout'>
       The context of response equation solver
     </param>
     <return>Error information</return>
   </function> */
QErrorCode RSPSolverDestroy(RSPSolver *rsp_solver)
{
#if defined(OPENRSP_C_USER_CONTEXT)
    rsp_solver->user_ctx = NULL;
#endif
    rsp_solver->get_linear_rsp_solution = NULL;
    return QSUCCESS;
}
```

3.10 Response Functions

Users can use the following API to get the response functions:

112 ⟨*OpenRSP.c* 17a⟩+≡

```
void OpenRSPGetRSPFun_f(const QInt num_props,
                        const QInt *len_tuple,
                        const QcPertInt *pert_tuple,
                        const QInt *num_freq_configs,
                        const QReal *pert_freqs,
                        const QInt *kn_rules,
                        const QcMat *ref_ham,
                        const QcMat *ref_overlap,
                        const QcMat *ref_state,
                        RSPSolver *rsp_solver,
                        RSPNucHamilton *nuc_hamilton,
                        RSPOverlap *overlap,
                        RSPOneOper *one_oper,
                        RSPTwoOper *two_oper,
                        RSPXCFun *xc_fun,
                        const QInt size_rsp_funs,
                        QReal *rsp_funs);

/*@% \brief gets the response functions for given perturbations
     \author Bin Gao
     \date 2014-07-31
     \param[OpenRSP:struct]{inout} open_rsp the context of response theory calculations
     \param[QcMat:struct]{in} ref_ham Hamiltonian of referenced state
     \param[QcMat:struct]{in} ref_state electronic state of referenced state
     \param[QcMat:struct]{in} ref_overlap overlap integral matrix of referenced state
     \param[QInt:int]{in} num_props number of properties to calculate
     \param[QInt:int]{in} len_tuple length of perturbation tuple for each property
     \param[QInt:int]{in} pert_tuple ordered list of perturbation labels
         for each property
     \param[QInt:int]{in} num_freq_configs number of different frequency
         configurations for each property
     \param[QReal:real]{in} pert_freqs complex frequencies of each perturbation label
         (except for the perturbation a) over all frequency configurations
     \param[QInt:int]{in} kn_rules number k for the kn rule for each property
     \param[QInt:int]{in} size_rsp_funs size of the response functions
     \param[QReal:real]{out} rsp_funs the response functions
     \return[QErrorCode:int] error information
*/
QErrorCode OpenRSPGetRSPFun(OpenRSP *open_rsp,
                            const QcMat *ref_ham,
                            const QcMat *ref_state,
                            const QcMat *ref_overlap,
                            const QInt num_props,
                            const QInt *len_tuple,
                            const QcPertInt *pert_tuple,
                            const QInt *num_freq_configs,
                            const QReal *pert_freqs,
                            const QInt *kn_rules,
```

```
                            const QInt size_rsp_funs,
                            QReal *rsp_funs)
{
    //QErrorCode ierr;  /* error information */
    if (open_rsp->assembled==QFALSE) {
        QErrorExit(FILE_AND_LINE, "OpenRSPAssemble() should be called before calculations")
    }
    //switch (open_rsp->elec_wav_type) {
    ///* density matrix-based response theory */
    //case ELEC_AO_D_MATRIX:
        OpenRSPGetRSPFun_f(num_props,
                           len_tuple,
                           pert_tuple,
                           num_freq_configs,
                           pert_freqs,
                           kn_rules,
                           ref_ham,
                           ref_overlap,
                           ref_state,
                           open_rsp->rsp_solver,
                           open_rsp->nuc_hamilton,
                           open_rsp->overlap,
                           open_rsp->one_oper,
                           open_rsp->two_oper,
                           open_rsp->xc_fun,
                           //id_outp,
                           size_rsp_funs,
                           rsp_funs);
    //    break;
    ///* molecular orbital (MO) coefficient matrix-based response theory */
    //case ELEC_MO_C_MATRIX:
    //    break;
    ///* couple cluster-based response theory */
    //case ELEC_COUPLED_CLUSTER:
    //    break;
    //default:
    //    printf("OpenRSPGetRSPFun>> type of (electronic) wave function %d\n",
    //           open_rsp->elec_wav_type);
    //    QErrorExit(FILE_AND_LINE, "invalid type of (electronic) wave function");
    //}
    return QSUCCESS;
}
```

3.11 Residues

Users can use the following API to get the residues:

114 ⟨*OpenRSP.c* 17a⟩+≡

```
/*@% \brief gets the residues for given perturbations
    \author Bin Gao
    \date 2014-07-31
    \param[OpenRSP:struct]{inout} open_rsp the context of response theory calculations
    \param[QcMat:struct]{in} ref_ham Hamiltonian of referenced state
    \param[QcMat:struct]{in} ref_state electronic state of referenced state
    \param[QcMat:struct]{in} ref_overlap overlap integral matrix of referenced state
    \param[QInt:int]{in} order_residue order of residues, that is also the length of
        each excitation tuple
    \param[QInt:int]{in} num_excit number of excitation tuples that will be used for
        residue calculations
    \param[QReal:real]{in} excit_energy excitation energies of all tuples, size is
        ``order_residue`` :math:`\times` ``num_excit``, and arranged
        as ``[num_excit][order_residue]``; that is, there will be
        ``order_residue`` frequencies of perturbation labels (or sums
        of frequencies of perturbation labels) respectively equal to
        the ``order_residue`` excitation energies per tuple
        ``excit_energy[i][:]`` (``i`` runs from ``0`` to ``num_excit-1``)
    \param[QcMat:struct]{in} eigen_vector eigenvectors (obtained from the generalized
        eigenvalue problem) of all excitation tuples, size is ``order_residue``
        :math:`\times` ``num_excit``, and also arranged in memory
        as ``[num_excit][order_residue]`` so that each eigenvector has
        its corresponding excitation energy in ``excit_energy``
    \param[QInt:int]{in} num_props number of properties to calculate
    \param[QInt:int]{in} len_tuple length of perturbation tuple for each property
    \param[QInt:int]{in} pert_tuple ordered list of perturbation labels
        for each property
    \param[QInt:int]{in} residue_num_pert for each property and each excitation energy
        in the tuple, the number of perturbation labels whose sum of
        frequencies equals to that excitation energy, size is ``order_residue``
        :math:`\times` ``num_props``, and arragned as ``[num_props][order_residue]``;
        a negative ``residue_num_pert[i][j]`` (``i`` runs from ``0`` to
        ``num_props-1``) means that the sum of frequencies of perturbation
        labels equals to ``-excit_energy[:][j]``
    \param[QInt:int]{in} residue_idx_pert for each property and each excitation energy
        in the tuple, the indices of perturbation labels whose sum of
        frequencies equals to that excitation energy, size is
        ``sum(residue_num_pert)``, and arranged as ``[residue_num_pert]``
    \param[QInt:int]{in} num_freq_configs number of different frequency
        configurations for each property
    \param[QReal:real]{in} pert_freqs complex frequencies of each perturbation
        label (except for the perturbation a) over all frequency configurations
        and excitation tuples
    \param[QInt:int]{in} kn_rules number k for the kn rule for each property
    \param[QInt:int]{in} size_residues size of the residues
    \param[QReal:real]{out} residues the residues
    \return[QErrorCode:int] error information
*/
```

```
QErrorCode OpenRSPGetResidue(OpenRSP *open_rsp,
                             const QcMat *ref_ham,
                             const QcMat *ref_state,
                             const QcMat *ref_overlap,
                             const QInt order_residue,
                             const QInt num_excit,
                             const QReal *excit_energy,
                             QcMat *eigen_vector[],
                             const QInt num_props,
                             const QInt *len_tuple,
                             const QcPertInt *pert_tuple,
                             const QInt *residue_num_pert,
                             const QInt *residue_idx_pert,
                             const QInt *num_freq_configs,
                             const QReal *pert_freqs,
                             const QInt *kn_rules,
                             const QInt size_residues,
                             QReal *residues)
{
    //QErrorCode ierr;  /* error information */
    if (open_rsp->assembled==QFALSE) {
        QErrorExit(FILE_AND_LINE, "OpenRSPAssemble() should be invoked before any calculati
    }
    //switch (open_rsp->elec_wav_type) {
    ///* density matrix-based response theory */
    //case ELEC_AO_D_MATRIX:
    //    break;
    ///* molecular orbital (MO) coefficient matrix-based response theory */
    //case ELEC_MO_C_MATRIX:
    //    break;
    ///* couple cluster-based response theory */
    //case ELEC_COUPLED_CLUSTER:
    //    break;
    //default:
    //    printf("OpenRSPGetResidue>> type of (electronic) wave function %d\n",
    //           open_rsp->elec_wav_type);
    //    QErrorExit(FILE_AND_LINE, "invalid type of (electronic) wave function");
    //}
    return QSUCCESS;
}
```

3.12 Fortran APIs

This section will implement APIs for Fortran users by using the Fortran `ISO_C_BINDING`.

We also plan to release this part of the OpenRSP under the GNU Lesser General Public License:

116a ⟨*OpenRSPLicenseFortran* 116a⟩≡

```
!! OpenRSP: open-ended library for response theory
!! Copyright 2015 Radovan Bast,
!!                Daniel H. Friese,
!!                Bin Gao,
!!                Dan J. Jonsson,
!!                Magnus Ringholm,
!!                Kenneth Ruud,
!!                Andreas Thorvaldsen
!!
!! OpenRSP is free software: you can redistribute it and/or modify
!! it under the terms of the GNU Lesser General Public License as
!! published by the Free Software Foundation, either version 3 of
!! the License, or (at your option) any later version.
!!
!! OpenRSP is distributed in the hope that it will be useful,
!! but WITHOUT ANY WARRANTY; without even the implied warranty of
!! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
!! GNU Lesser General Public License for more details.
!!
!! You should have received a copy of the GNU Lesser General Public
!! License along with OpenRSP. If not, see <http://www.gnu.org/licenses/>.
```

Here is the organization of the module file:

116b ⟨*OpenRSP.F90* 116b⟩≡

```
!!
!! <QCLANG='Fortran'>
!! <para>
!!   Fortran users should use the module <OpenRSP_f> in their codes to access
!!   the functionalities of OpenRSP. We have used the same name for Fortran
!!   data types and constants, for instance <OpenRSP>; macro definitions are
!!   also controlled by the same names, such as <OPENRSP_USER_CONTEXT>; however
!!   all Fortran modules and functions are appended by <c>_f</c>.
!! </para>
!!
⟨OpenRSPLicenseFortran 116a⟩
!! <module name='OpenRSP_f' author='Bin Gao' date='2014-07-12'>
!!   The module file of OpenRSP library for Fortran users
!! </module>

! basic data types
#include "api/qcmatrix_c_type.h"

module OpenRSP_f

    use, intrinsic :: iso_c_binding
    use qcmatrix_f, only: QINT,     &
                          QREAL,    &
```

```
                      QFAILURE, &
                      QcMat,    &
                      QcMat_C_LOC
use RSPSolver_f, only: SolverFun_f,       &
                       RSPSolverCreate_f, &
                       RSPSolverDestroy_f
use RSPPert_f, only: QcPertInt,       &
                     PertFun_f,       &
                     RSPPertCreate_f, &
                     RSPPertDestroy_f
use RSPOverlap_f, only: OverlapFun_f,       &
                        RSPOverlapCreate_f, &
                        RSPOverlapDestroy_f
use RSPOneOper_f, only: OneOperFun_f,       &
                        RSPOneOperCreate_f, &
                        RSPOneOperDestroy_f
use RSPTwoOper_f, only: TwoOperFun_f,       &
                        RSPTwoOperCreate_f, &
                        RSPTwoOperDestroy_f
use RSPXCFun_f, only: XCFunFun_f,       &
                      RSPXCFunCreate_f, &
                      RSPXCFunDestroy_f
use RSPNucHamilton_f, only: NucHamiltonFun_f,       &
                            RSPNucHamiltonCreate_f, &
                            RSPNucHamiltonDestroy_f

implicit none

! type of equation of motion (EOM) of electrons
integer(kind=QINT), parameter, public :: ELEC_AO_D_MATRIX = 0
integer(kind=QINT), parameter, public :: ELEC_MO_C_MATRIX = 1
integer(kind=QINT), parameter, public :: ELEC_COUPLED_CLUSTER = 2

! linked list of context of callback subroutines of one-electron operators
type, private :: OneOperList_f
    type(OneOperFun_f), pointer :: one_oper_fun => null()
    type(OneOperList_f), pointer :: next_one_oper => null()
end type OneOperList_f

! linked list of context of callback subroutines of two-electron operators
type, private :: TwoOperList_f
    type(TwoOperFun_f), pointer :: two_oper_fun => null()
    type(TwoOperList_f), pointer :: next_two_oper => null()
end type TwoOperList_f

! linked list of context of callback subroutines of XC functionals
type, private :: XCFunList_f
    type(XCFunFun_f), pointer :: xcfun_fun => null()
    type(XCFunList_f), pointer :: next_xc_fun => null()
end type XCFunList_f

! OpenRSP type (inspired by http://wiki.rac.manchester.ac.uk/community/GPU/GpuFaq/Fortr
```

```
type, public :: OpenRSP
    private
    type(C_PTR) :: c_rsp = C_NULL_PTR
    type(SolverFun_f), pointer :: solver_fun => null()
    type(PertFun_f), pointer :: pert_fun => null()
    type(OverlapFun_f), pointer :: overlap_fun => null()
    type(OneOperList_f), pointer :: list_one_oper => null()
    type(TwoOperList_f), pointer :: list_two_oper => null()
    type(XCFunList_f), pointer :: list_xc_fun => null()
    type(NucHamiltonFun_f), pointer :: nuc_hamilton_fun => null()
end type OpenRSP

! functions provided by the Fortran APIs
public :: OpenRSPCreate_f
!public :: OpenRSPSetElecEOM_f
public :: OpenRSPSetLinearRSPSolver_f
public :: OpenRSPSetPerturbations_f
public :: OpenRSPSetOverlap_f
public :: OpenRSPAddOneOper_f
public :: OpenRSPAddTwoOper_f
public :: OpenRSPAddXCFun_f
public :: OpenRSPSetNucHamilton_f
public :: OpenRSPAssemble_f
public :: OpenRSPWrite_f
public :: OpenRSPGetRSPFun_f
!public :: OpenRSPGetResidue_f
public :: OpenRSPDestroy_f

interface
    integer(C_INT) function OpenRSPCreateFortranAdapter(open_rsp) &
        bind(C, name="OpenRSPCreateFortranAdapter")
        use, intrinsic :: iso_c_binding
        type(C_PTR), intent(inout) :: open_rsp
    end function OpenRSPCreateFortranAdapter
    !integer(C_INT) function f_api_OpenRSPSetElecEOM(open_rsp,      &
    !                                                elec_EOM_type) &
    !    bind(C, name="f_api_OpenRSPSetElecEOM")
    !    use, intrinsic :: iso_c_binding
    !    type(C_PTR), intent(inout) :: open_rsp
    !    integer(kind=C_QINT), value, intent(in) :: elec_EOM_type
    !end function f_api_OpenRSPSetElecEOM
    integer(C_INT) function OpenRSPSetLinearRSPSolver(open_rsp,         &
                                                      user_ctx,         &
                                                      get_linear_rsp_solution) &
        bind(C, name="OpenRSPSetLinearRSPSolver")
        use, intrinsic :: iso_c_binding
        type(C_PTR), value, intent(in) :: open_rsp
        type(C_PTR), value, intent(in) :: user_ctx
        type(C_FUNPTR), value, intent(in) :: get_linear_rsp_solution
    end function OpenRSPSetLinearRSPSolver
    integer(C_INT) function OpenRSPSetPerturbations(open_rsp,               &
                                                    num_pert_lab,           &
```

```
                                                  pert_labels,            &
                                                  pert_max_orders,        &
                                                  pert_num_comps,         &
                                                  user_ctx,               &
                                                  get_pert_concatenation) &
    bind(C, name="OpenRSPSetPerturbations")
    use, intrinsic :: iso_c_binding
    use RSPPertBasicTypes_f, only: C_QCPERTINT
    type(C_PTR), value, intent(in) :: open_rsp
    integer(kind=C_QINT), value, intent(in) :: num_pert_lab
    integer(kind=C_QCPERTINT), intent(in) :: pert_labels(num_pert_lab)
    integer(kind=C_QINT), intent(in) :: pert_max_orders(num_pert_lab)
    integer(kind=C_QINT), intent(in) :: pert_num_comps(sum(pert_max_orders))
    type(C_PTR), value, intent(in) :: user_ctx
    type(C_FUNPTR), value, intent(in) :: get_pert_concatenation
end function OpenRSPSetPerturbations
integer(C_INT) function OpenRSPSetOverlap(open_rsp,        &
                                          num_pert_lab,    &
                                          pert_labels,     &
                                          pert_max_orders, &
                                          user_ctx,        &
                                          get_overlap_mat, &
                                          get_overlap_exp) &
    bind(C, name="OpenRSPSetOverlap")
    use, intrinsic :: iso_c_binding
    use RSPPertBasicTypes_f, only: C_QCPERTINT
    type(C_PTR), value, intent(in) :: open_rsp
    integer(kind=C_QINT), value, intent(in) :: num_pert_lab
    integer(kind=C_QCPERTINT), intent(in) :: pert_labels(num_pert_lab)
    integer(kind=C_QINT), intent(in) :: pert_max_orders(num_pert_lab)
    type(C_PTR), value, intent(in) :: user_ctx
    type(C_FUNPTR), value, intent(in) :: get_overlap_mat
    type(C_FUNPTR), value, intent(in) :: get_overlap_exp
end function OpenRSPSetOverlap
integer(C_INT) function OpenRSPAddOneOper(open_rsp,         &
                                          num_pert_lab,     &
                                          pert_labels,      &
                                          pert_max_orders,  &
                                          user_ctx,         &
                                          get_one_oper_mat, &
                                          get_one_oper_exp) &
    bind(C, name="OpenRSPAddOneOper")
    use, intrinsic :: iso_c_binding
    use RSPPertBasicTypes_f, only: C_QCPERTINT
    type(C_PTR), value, intent(in) :: open_rsp
    integer(kind=C_QINT), value, intent(in) :: num_pert_lab
    integer(kind=C_QCPERTINT), intent(in) :: pert_labels(num_pert_lab)
    integer(kind=C_QINT), intent(in) :: pert_max_orders(num_pert_lab)
    type(C_PTR), value, intent(in) :: user_ctx
    type(C_FUNPTR), value, intent(in) :: get_one_oper_mat
    type(C_FUNPTR), value, intent(in) :: get_one_oper_exp
end function OpenRSPAddOneOper
```

```
integer(C_INT) function OpenRSPAddTwoOper(open_rsp,         &
                                          num_pert_lab,     &
                                          pert_labels,      &
                                          pert_max_orders,  &
                                          user_ctx,         &
                                          get_two_oper_mat, &
                                          get_two_oper_exp) &
    bind(C, name="OpenRSPAddTwoOper")
    use, intrinsic :: iso_c_binding
    use RSPPertBasicTypes_f, only: C_QCPERTINT
    type(C_PTR), value, intent(in) :: open_rsp
    integer(kind=C_QINT), value, intent(in) :: num_pert_lab
    integer(kind=C_QCPERTINT), intent(in) :: pert_labels(num_pert_lab)
    integer(kind=C_QINT), intent(in) :: pert_max_orders(num_pert_lab)
    type(C_PTR), value, intent(in) :: user_ctx
    type(C_FUNPTR), value, intent(in) :: get_two_oper_mat
    type(C_FUNPTR), value, intent(in) :: get_two_oper_exp
end function OpenRSPAddTwoOper
integer(C_INT) function OpenRSPAddXCFun(open_rsp,        &
                                        num_pert_lab,    &
                                        pert_labels,     &
                                        pert_max_orders, &
                                        user_ctx,        &
                                        get_xc_fun_mat,  &
                                        get_xc_fun_exp)  &
    bind(C, name="OpenRSPAddXCFun")
    use, intrinsic :: iso_c_binding
    use RSPPertBasicTypes_f, only: C_QCPERTINT
    type(C_PTR), value, intent(in) :: open_rsp
    integer(kind=C_QINT), value, intent(in) :: num_pert_lab
    integer(kind=C_QCPERTINT), intent(in) :: pert_labels(num_pert_lab)
    integer(kind=C_QINT), intent(in) :: pert_max_orders(num_pert_lab)
    type(C_PTR), value, intent(in) :: user_ctx
    type(C_FUNPTR), value, intent(in) :: get_xc_fun_mat
    type(C_FUNPTR), value, intent(in) :: get_xc_fun_exp
end function OpenRSPAddXCFun
integer(C_INT) function OpenRSPSetNucHamilton(open_rsp,        &
                                              num_pert_lab,    &
                                              pert_labels,     &
                                              pert_max_orders, &
                                              user_ctx,        &
                                              get_nuc_contrib, &
                                              num_atoms)       &
    bind(C, name="OpenRSPSetNucHamilton")
    use, intrinsic :: iso_c_binding
    use RSPPertBasicTypes_f, only: C_QCPERTINT
    type(C_PTR), value, intent(in) :: open_rsp
    integer(kind=C_QINT), value, intent(in) :: num_pert_lab
    integer(kind=C_QCPERTINT), intent(in) :: pert_labels(num_pert_lab)
    integer(kind=C_QINT), intent(in) :: pert_max_orders(num_pert_lab)
    type(C_PTR), value, intent(in) :: user_ctx
    type(C_FUNPTR), value, intent(in) :: get_nuc_contrib
```

```
        integer(kind=C_QINT), value, intent(in) :: num_atoms
    end function OpenRSPSetNucHamilton
    integer(C_INT) function OpenRSPAssemble(open_rsp) &
        bind(C, name="OpenRSPAssemble")
        use, intrinsic :: iso_c_binding
        type(C_PTR), value, intent(in) :: open_rsp
    end function OpenRSPAssemble
    integer(C_INT) function OpenRSPWrite(open_rsp, file_name) &
        bind(C, name="OpenRSPWrite")
        use, intrinsic :: iso_c_binding
        type(C_PTR), value, intent(in) :: open_rsp
        character(C_CHAR), intent(in) :: file_name(*)
    end function OpenRSPWrite
    integer(C_INT) function OpenRSPGetRSPFun(open_rsp,         &
                                             ref_ham,          &
                                             ref_state,        &
                                             ref_overlap,      &
                                             num_props,        &
                                             len_tuple,        &
                                             pert_tuple,       &
                                             num_freq_configs, &
                                             pert_freqs,       &
                                             kn_rules,         &
                                             size_rsp_funs,    &
                                             rsp_funs)         &
        bind(C, name="OpenRSPGetRSPFun")
        use, intrinsic :: iso_c_binding
        use RSPPertBasicTypes_f, only: C_QCPERTINT
        type(C_PTR), value, intent(in) :: open_rsp
        type(C_PTR), value, intent(in) :: ref_ham
        type(C_PTR), value, intent(in) :: ref_state
        type(C_PTR), value, intent(in) :: ref_overlap
        integer(kind=C_QINT), value, intent(in) :: num_props
        integer(kind=C_QINT), intent(in) :: len_tuple(num_props)
        integer(kind=C_QCPERTINT), intent(in) :: pert_tuple(sum(len_tuple))
        integer(kind=C_QINT), intent(in) :: num_freq_configs(num_props)
        real(kind=C_QREAL), intent(in) :: pert_freqs(2*dot_product(len_tuple,num_freq_c
        integer(kind=C_QINT), intent(in) :: kn_rules(num_props)
        integer(kind=C_QINT), value, intent(in) :: size_rsp_funs
        real(kind=C_QREAL), intent(out) :: rsp_funs(2*size_rsp_funs)
    end function OpenRSPGetRSPFun
    integer(C_INT) function OpenRSPDestroyFortranAdapter(open_rsp) &
        bind(C, name="OpenRSPDestroyFortranAdapter")
        use, intrinsic :: iso_c_binding
        type(C_PTR), intent(inout) :: open_rsp
    end function OpenRSPDestroyFortranAdapter
end interface

contains

function OpenRSPCreate_f(open_rsp) result(ierr)
    integer(kind=4) :: ierr
```

```
        type(OpenRSP), intent(inout) :: open_rsp
        ierr = OpenRSPCreateFortranAdapter(open_rsp%c_rsp)
        nullify(open_rsp%solver_fun)
        nullify(open_rsp%pert_fun)
        nullify(open_rsp%overlap_fun)
        nullify(open_rsp%list_one_oper)
        nullify(open_rsp%list_two_oper)
        nullify(open_rsp%list_xc_fun)
        nullify(open_rsp%nuc_hamilton_fun)
    end function OpenRSPCreate_f

    !function OpenRSPSetElecEOM_f(open_rsp, elec_EOM_type) result(ierr)
    !    integer(kind=4) :: ierr
    !    type(OpenRSP), intent(inout) :: open_rsp
    !    integer(kind=QINT), intent(in) :: elec_EOM_type
    !    ierr = f_api_OpenRSPSetElecEOM(open_rsp%c_rsp, elec_EOM_type)
    !end function OpenRSPSetElecEOM_f

    function OpenRSPSetLinearRSPSolver_f(open_rsp,        &
#if defined(OPENRSP_F_USER_CONTEXT)
                                         user_ctx,        &
#endif
                                         get_linear_rsp_solution) result(ierr)
        integer(kind=4) :: ierr
        type(OpenRSP), intent(inout) :: open_rsp
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        interface
            subroutine get_linear_rsp_solution(num_freq_sums, &
                                               freq_sums,     &
                                               size_pert,     &
                                               RHS_mat,       &
#if defined(OPENRSP_F_USER_CONTEXT)
                                               len_ctx,       &
                                               user_ctx,      &
#endif
                                               rsp_param)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                integer(kind=QINT), intent(in) :: num_freq_sums
                real(kind=QREAL), intent(in) :: freq_sums(2*num_freq_sums)
                integer(kind=QINT), intent(in) :: size_pert
                type(QcMat), intent(in) :: RHS_mat(num_freq_sums*size_pert)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                type(QcMat), intent(inout) :: rsp_param(num_freq_sums*size_pert)
            end subroutine get_linear_rsp_solution
            subroutine RSPSolverGetLinearRSPSolution_f(num_freq_sums, &
                                                       freq_sums,     &
                                                       size_pert,     &
```

```
                                                    RHS_mat,       &
                                                    user_ctx,      &
                                                    rsp_param)     &
                bind(C, name="RSPSolverGetLinearRSPSolution_f")
                use, intrinsic :: iso_c_binding
                integer(kind=C_QINT), value, intent(in) :: num_freq_sums
                real(kind=C_QREAL), intent(in) :: freq_sums(2*num_freq_sums)
                integer(kind=C_QINT), value, intent(in) :: size_pert
                type(C_PTR), intent(in) :: RHS_mat(num_freq_sums*size_pert)
                type(C_PTR), value, intent(in) :: user_ctx
                type(C_PTR), intent(inout) :: rsp_param(num_freq_sums*size_pert)
            end subroutine RSPSolverGetLinearRSPSolution_f
        end interface
        if (associated(open_rsp%solver_fun)) then
            call RSPSolverDestroy_f(open_rsp%solver_fun)
        else
            allocate(open_rsp%solver_fun)
        end if
        ! adds context of callback function of response equation solver
        call RSPSolverCreate_f(open_rsp%solver_fun, &
#if defined(OPENRSP_F_USER_CONTEXT)
                               user_ctx,            &
#endif
                               get_linear_rsp_solution)
        ierr = OpenRSPSetLinearRSPSolver(open_rsp%c_rsp,             &
                                         c_loc(open_rsp%solver_fun), &
                                         c_funloc(RSPSolverGetLinearRSPSolution_f))
    end function OpenRSPSetLinearRSPSolver_f

    function OpenRSPSetPerturbations_f(open_rsp,        &
                                       num_pert_lab,    &
                                       pert_labels,     &
                                       pert_max_orders, &
                                       pert_num_comps,  &
#if defined(OPENRSP_F_USER_CONTEXT)
                                       user_ctx,        &
#endif
                                       get_pert_concatenation) result(ierr)
        integer(kind=4) :: ierr
        type(OpenRSP), intent(inout) :: open_rsp
        integer(kind=QINT), intent(in) :: num_pert_lab
        integer(kind=QcPertInt), intent(in) :: pert_labels(num_pert_lab)
        integer(kind=QINT), intent(in) :: pert_max_orders(num_pert_lab)
        integer(kind=QINT), intent(in) :: pert_num_comps(:)
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        interface
            subroutine get_pert_concatenation(pert_label,     &
                                              first_cat_comp, &
                                              num_cat_comps,  &
                                              num_sub_tuples, &
```

```
                                            len_sub_tuples, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                            len_ctx,        &
                                            user_ctx,       &
#endif
                                            rank_sub_comps)
                use qcmatrix_f, only: QINT
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QcPertInt), intent(in) :: pert_label
                integer(kind=QINT), intent(in) :: first_cat_comp
                integer(kind=QINT), intent(in) :: num_cat_comps
                integer(kind=QINT), intent(in) :: num_sub_tuples
                integer(kind=QINT), intent(in) :: len_sub_tuples(num_sub_tuples)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(out) :: rank_sub_comps(num_sub_tuples*num_cat_co
            end subroutine get_pert_concatenation
            subroutine RSPPertGetConcatenation_f(pert_label,     &
                                                 first_cat_comp, &
                                                 num_cat_comps,  &
                                                 num_sub_tuples, &
                                                 len_sub_tuples, &
                                                 user_ctx,       &
                                                 rank_sub_comps) &
                bind(C, name="RSPPertGetConcatenation_f")
                use, intrinsic :: iso_c_binding
                use RSPPertBasicTypes_f, only: C_QCPERTINT
                integer(kind=C_QCPERTINT), value, intent(in) :: pert_label
                integer(kind=C_QINT), value, intent(in) :: first_cat_comp
                integer(kind=C_QINT), value, intent(in) :: num_cat_comps
                integer(kind=C_QINT), value, intent(in) :: num_sub_tuples
                integer(kind=C_QINT), intent(in) :: len_sub_tuples(num_sub_tuples)
                type(C_PTR), value, intent(in) :: user_ctx
                integer(kind=C_QINT), intent(out) :: rank_sub_comps(num_sub_tuples*num_cat_
            end subroutine RSPPertGetConcatenation_f
        end interface
        if (associated(open_rsp%pert_fun)) then
            call RSPPertDestroy_f(open_rsp%pert_fun)
        else
            allocate(open_rsp%pert_fun)
        end if
        ! adds context of callback functions of perturbations
        call RSPPertCreate_f(open_rsp%pert_fun, &
#if defined(OPENRSP_F_USER_CONTEXT)
                             user_ctx,          &
#endif
                             get_pert_concatenation)
        ierr = OpenRSPSetPerturbations(open_rsp%c_rsp,             &
                                       num_pert_lab,               &
                                       pert_labels,                &
```

```
                                        pert_max_orders,            &
                                        pert_num_comps,             &
                                        c_loc(open_rsp%pert_fun),   &
                                        c_funloc(RSPPertGetConcatenation_f))
    end function OpenRSPSetPerturbations_f

    function OpenRSPSetOverlap_f(open_rsp,        &
                                 num_pert_lab,    &
                                 pert_labels,     &
                                 pert_max_orders, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                 user_ctx,        &
#endif
                                 get_overlap_mat, &
                                 get_overlap_exp) result(ierr)
        integer(kind=4) :: ierr
        type(OpenRSP), intent(inout) :: open_rsp
        integer(kind=QINT), intent(in) :: num_pert_lab
        integer(kind=QcPertInt), intent(in) :: pert_labels(num_pert_lab)
        integer(kind=QINT), intent(in) :: pert_max_orders(num_pert_lab)
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        interface
            subroutine get_overlap_mat(bra_num_pert,     &
                                       bra_pert_labels,  &
                                       bra_pert_orders,  &
                                       ket_num_pert,     &
                                       ket_pert_labels,  &
                                       ket_pert_orders,  &
                                       oper_num_pert,    &
                                       oper_pert_labels, &
                                       oper_pert_orders, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                       len_ctx,          &
                                       user_ctx,         &
#endif
                                       num_int,          &
                                       val_int)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: bra_num_pert
                integer(kind=QcPertInt), intent(in) :: bra_pert_labels(bra_num_pert)
                integer(kind=QINT), intent(in) :: bra_pert_orders(bra_num_pert)
                integer(kind=QINT), intent(in) :: ket_num_pert
                integer(kind=QcPertInt), intent(in) :: ket_pert_labels(ket_num_pert)
                integer(kind=QINT), intent(in) :: ket_pert_orders(ket_num_pert)
                integer(kind=QINT), intent(in) :: oper_num_pert
                integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
```

```
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_int
                type(QcMat), intent(inout) :: val_int(num_int)
            end subroutine get_overlap_mat
            subroutine get_overlap_exp(bra_num_pert,     &
                                       bra_pert_labels,  &
                                       bra_pert_orders,  &
                                       ket_num_pert,     &
                                       ket_pert_labels,  &
                                       ket_pert_orders,  &
                                       oper_num_pert,    &
                                       oper_pert_labels, &
                                       oper_pert_orders, &
                                       num_dmat,         &
                                       dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                       len_ctx,          &
                                       user_ctx,         &
#endif
                                       num_exp,          &
                                       val_exp)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: bra_num_pert
                integer(kind=QcPertInt), intent(in) :: bra_pert_labels(bra_num_pert)
                integer(kind=QINT), intent(in) :: bra_pert_orders(bra_num_pert)
                integer(kind=QINT), intent(in) :: ket_num_pert
                integer(kind=QcPertInt), intent(in) :: ket_pert_labels(ket_num_pert)
                integer(kind=QINT), intent(in) :: ket_pert_orders(ket_num_pert)
                integer(kind=QINT), intent(in) :: oper_num_pert
                integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
                integer(kind=QINT), intent(in) :: num_dmat
                type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_exp
                real(kind=QREAL), intent(inout) :: val_exp(num_exp)
            end subroutine get_overlap_exp
            subroutine RSPOverlapGetMat_f(bra_num_pert,     &
                                          bra_pert_labels,  &
                                          bra_pert_orders,  &
                                          ket_num_pert,     &
                                          ket_pert_labels,  &
                                          ket_pert_orders,  &
                                          oper_num_pert,    &
                                          oper_pert_labels, &
                                          oper_pert_orders, &
                                          user_ctx,         &
```

```
                                num_int,          &
                                val_int)          &
            bind(C, name="RSPOverlapGetMat_f")
            use, intrinsic :: iso_c_binding
            use RSPPertBasicTypes_f, only: C_QCPERTINT
            integer(kind=C_QINT), value, intent(in) :: bra_num_pert
            integer(kind=C_QCPERTINT), intent(in) :: bra_pert_labels(bra_num_pert)
            integer(kind=C_QINT), intent(in) :: bra_pert_orders(bra_num_pert)
            integer(kind=C_QINT), value, intent(in) :: ket_num_pert
            integer(kind=C_QCPERTINT), intent(in) :: ket_pert_labels(ket_num_pert)
            integer(kind=C_QINT), intent(in) :: ket_pert_orders(ket_num_pert)
            integer(kind=C_QINT), value, intent(in) :: oper_num_pert
            integer(kind=C_QCPERTINT), intent(in) :: oper_pert_labels(oper_num_pert)
            integer(kind=C_QINT), intent(in) :: oper_pert_orders(oper_num_pert)
            type(C_PTR), value, intent(in) :: user_ctx
            integer(kind=C_QINT), value, intent(in) :: num_int
            type(C_PTR), intent(inout) :: val_int(num_int)
        end subroutine RSPOverlapGetMat_f
        subroutine RSPOverlapGetExp_f(bra_num_pert,     &
                                bra_pert_labels,  &
                                bra_pert_orders,  &
                                ket_num_pert,     &
                                ket_pert_labels,  &
                                ket_pert_orders,  &
                                oper_num_pert,    &
                                oper_pert_labels, &
                                oper_pert_orders, &
                                num_dmat,         &
                                dens_mat,         &
                                user_ctx,         &
                                num_exp,          &
                                val_exp)          &
            bind(C, name="RSPOverlapGetExp_f")
            use, intrinsic :: iso_c_binding
            use RSPPertBasicTypes_f, only: C_QCPERTINT
            integer(kind=C_QINT), value, intent(in) :: bra_num_pert
            integer(kind=C_QCPERTINT), intent(in) :: bra_pert_labels(bra_num_pert)
            integer(kind=C_QINT), intent(in) :: bra_pert_orders(bra_num_pert)
            integer(kind=C_QINT), value, intent(in) :: ket_num_pert
            integer(kind=C_QCPERTINT), intent(in) :: ket_pert_labels(ket_num_pert)
            integer(kind=C_QINT), intent(in) :: ket_pert_orders(ket_num_pert)
            integer(kind=C_QINT), value, intent(in) :: oper_num_pert
            integer(kind=C_QCPERTINT), intent(in) :: oper_pert_labels(oper_num_pert)
            integer(kind=C_QINT), intent(in) :: oper_pert_orders(oper_num_pert)
            integer(kind=C_QINT), value, intent(in) :: num_dmat
            type(C_PTR), intent(in) :: dens_mat(num_dmat)
            type(C_PTR), value, intent(in) :: user_ctx
            integer(kind=C_QINT), value, intent(in) :: num_exp
            real(kind=C_QREAL), intent(inout) :: val_exp(num_exp)
        end subroutine RSPOverlapGetExp_f
    end interface
    if (associated(open_rsp%overlap_fun)) then
```

```
            call RSPOverlapDestroy_f(open_rsp%overlap_fun)
        else
            allocate(open_rsp%overlap_fun)
        end if
        ! adds context of callback functions of the overlap integrals
        call RSPOverlapCreate_f(open_rsp%overlap_fun, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                user_ctx,             &
#endif
                                get_overlap_mat,      &
                                get_overlap_exp)
        ierr = OpenRSPSetOverlap(open_rsp%c_rsp,              &
                                 num_pert_lab,                &
                                 pert_labels,                 &
                                 pert_max_orders,             &
                                 c_loc(open_rsp%overlap_fun), &
                                 c_funloc(RSPOverlapGetMat_f), &
                                 c_funloc(RSPOverlapGetExp_f))
    end function OpenRSPSetOverlap_f

    function OpenRSPAddOneOper_f(open_rsp,         &
                                 num_pert_lab,     &
                                 pert_labels,      &
                                 pert_max_orders,  &
#if defined(OPENRSP_F_USER_CONTEXT)
                                 user_ctx,         &
#endif
                                 get_one_oper_mat, &
                                 get_one_oper_exp) result(ierr)
        integer(kind=4) :: ierr
        type(OpenRSP), intent(inout) :: open_rsp
        integer(kind=QINT), intent(in) :: num_pert_lab
        integer(kind=QcPertInt), intent(in) :: pert_labels(num_pert_lab)
        integer(kind=QINT), intent(in) :: pert_max_orders(num_pert_lab)
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        interface
            subroutine get_one_oper_mat(oper_num_pert,    &
                                        oper_pert_labels, &
                                        oper_pert_orders, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                        len_ctx,          &
                                        user_ctx,         &
#endif
                                        num_int,          &
                                        val_int)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: oper_num_pert
                integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
```

```
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_int
                type(QcMat), intent(inout) :: val_int(num_int)
            end subroutine get_one_oper_mat
            subroutine get_one_oper_exp(oper_num_pert,    &
                                        oper_pert_labels, &
                                        oper_pert_orders, &
                                        num_dmat,         &
                                        dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                        len_ctx,          &
                                        user_ctx,         &
#endif
                                        num_exp,          &
                                        val_exp)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: oper_num_pert
                integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
                integer(kind=QINT), intent(in) :: num_dmat
                type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_exp
                real(kind=QREAL), intent(inout) :: val_exp(num_exp)
            end subroutine get_one_oper_exp
            subroutine RSPOneOperGetMat_f(oper_num_pert,    &
                                          oper_pert_labels, &
                                          oper_pert_orders, &
                                          user_ctx,         &
                                          num_int,          &
                                          val_int)          &
                bind(C, name="RSPOneOperGetMat_f")
                use, intrinsic :: iso_c_binding
                use RSPPertBasicTypes_f, only: C_QCPERTINT
                integer(kind=C_QINT), value, intent(in) :: oper_num_pert
                integer(kind=C_QCPERTINT), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=C_QINT), intent(in) :: oper_pert_orders(oper_num_pert)
                type(C_PTR), value, intent(in) :: user_ctx
                integer(kind=C_QINT), value, intent(in) :: num_int
                type(C_PTR), intent(inout) :: val_int(num_int)
            end subroutine RSPOneOperGetMat_f
            subroutine RSPOneOperGetExp_f(oper_num_pert,    &
                                          oper_pert_labels, &
                                          oper_pert_orders, &
                                          num_dmat,         &
```

```
                                        dens_mat,         &
                                        user_ctx,         &
                                        num_exp,          &
                                        val_exp)          &
                bind(C, name="RSPOneOperGetExp_f")
                use, intrinsic :: iso_c_binding
                use RSPPertBasicTypes_f, only: C_QCPERTINT
                integer(kind=C_QINT), value, intent(in) :: oper_num_pert
                integer(kind=C_QCPERTINT), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=C_QINT), intent(in) :: oper_pert_orders(oper_num_pert)
                integer(kind=C_QINT), value, intent(in) :: num_dmat
                type(C_PTR), intent(in) :: dens_mat(num_dmat)
                type(C_PTR), value, intent(in) :: user_ctx
                integer(kind=C_QINT), value, intent(in) :: num_exp
                real(kind=C_QREAL), intent(inout) :: val_exp(num_exp)
            end subroutine RSPOneOperGetExp_f
        end interface
        type(OneOperList_f), pointer :: cur_one_oper  !current one-electron operator
        ! inserts the context of callback functions to the tail of the linked list
        if (associated(open_rsp%list_one_oper)) then
            cur_one_oper => open_rsp%list_one_oper
            do while (associated(cur_one_oper%next_one_oper))
                cur_one_oper => cur_one_oper%next_one_oper
            end do
            allocate(cur_one_oper%next_one_oper)
            cur_one_oper => cur_one_oper%next_one_oper
        else
            allocate(open_rsp%list_one_oper)
            cur_one_oper => open_rsp%list_one_oper
        end if
        allocate(cur_one_oper%one_oper_fun)
        nullify(cur_one_oper%next_one_oper)
        ! adds context of callback functions of the new one-electron operator
        call RSPOneOperCreate_f(cur_one_oper%one_oper_fun, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                user_ctx,                  &
#endif
                                get_one_oper_mat,          &
                                get_one_oper_exp)
        ierr = OpenRSPAddOneOper(open_rsp%c_rsp,                   &
                                 num_pert_lab,                     &
                                 pert_labels,                      &
                                 pert_max_orders,                  &
                                 c_loc(cur_one_oper%one_oper_fun), &
                                 c_funloc(RSPOneOperGetMat_f),     &
                                 c_funloc(RSPOneOperGetExp_f))
    end function OpenRSPAddOneOper_f

    function OpenRSPAddTwoOper_f(open_rsp,         &
                                 num_pert_lab,     &
                                 pert_labels,      &
                                 pert_max_orders,  &
```

```
#if defined(OPENRSP_F_USER_CONTEXT)
                                user_ctx,         &
#endif
                                get_two_oper_mat, &
                                get_two_oper_exp) result(ierr)
        integer(kind=4) :: ierr
        type(OpenRSP), intent(inout) :: open_rsp
        integer(kind=QINT), intent(in) :: num_pert_lab
        integer(kind=QcPertInt), intent(in) :: pert_labels(num_pert_lab)
        integer(kind=QINT), intent(in) :: pert_max_orders(num_pert_lab)
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        interface
            subroutine get_two_oper_mat(oper_num_pert,    &
                                        oper_pert_labels, &
                                        oper_pert_orders, &
                                        num_dmat,         &
                                        dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                        len_ctx,          &
                                        user_ctx,         &
#endif
                                        num_int,          &
                                        val_int)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: oper_num_pert
                integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
                integer(kind=QINT), intent(in) :: num_dmat
                type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_int
                type(QcMat), intent(inout) :: val_int(num_int)
            end subroutine get_two_oper_mat
            subroutine get_two_oper_exp(oper_num_pert,    &
                                        oper_pert_labels, &
                                        oper_pert_orders, &
                                        dmat_len_tuple,   &
                                        num_LHS_dmat,     &
                                        LHS_dens_mat,     &
                                        num_RHS_dmat,     &
                                        RHS_dens_mat,     &
#if defined(OPENRSP_F_USER_CONTEXT)
                                        len_ctx,          &
                                        user_ctx,         &
#endif
                                        num_exp,          &
```

```
                                    val_exp)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: oper_num_pert
                integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
                integer(kind=QINT), intent(in) :: dmat_len_tuple
                integer(kind=QINT), intent(in) :: num_LHS_dmat(dmat_len_tuple)
                type(QcMat), intent(in) :: LHS_dens_mat(sum(num_LHS_dmat))
                integer(kind=QINT), intent(in) :: num_RHS_dmat(dmat_len_tuple)
                type(QcMat), intent(in) :: RHS_dens_mat(sum(num_RHS_dmat))
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_exp
                real(kind=QREAL), intent(inout) :: val_exp(num_exp)
            end subroutine get_two_oper_exp
            subroutine RSPTwoOperGetMat_f(oper_num_pert,    &
                                          oper_pert_labels, &
                                          oper_pert_orders, &
                                          num_dmat,         &
                                          dens_mat,         &
                                          user_ctx,         &
                                          num_int,          &
                                          val_int)          &
                bind(C, name="RSPTwoOperGetMat_f")
                use, intrinsic :: iso_c_binding
                use RSPPertBasicTypes_f, only: C_QCPERTINT
                integer(kind=C_QINT), value, intent(in) :: oper_num_pert
                integer(kind=C_QCPERTINT), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=C_QINT), intent(in) :: oper_pert_orders(oper_num_pert)
                integer(kind=C_QINT), value, intent(in) :: num_dmat
                type(C_PTR), intent(in) :: dens_mat(num_dmat)
                type(C_PTR), value, intent(in) :: user_ctx
                integer(kind=C_QINT), value, intent(in) :: num_int
                type(C_PTR), intent(inout) :: val_int(num_int)
            end subroutine RSPTwoOperGetMat_f
            subroutine RSPTwoOperGetExp_f(oper_num_pert,    &
                                          oper_pert_labels, &
                                          oper_pert_orders, &
                                          dmat_len_tuple,   &
                                          num_LHS_dmat,     &
                                          LHS_dens_mat,     &
                                          num_RHS_dmat,     &
                                          RHS_dens_mat,     &
                                          user_ctx,         &
                                          num_exp,          &
                                          val_exp)          &
                bind(C, name="RSPTwoOperGetExp_f")
                use, intrinsic :: iso_c_binding
                use RSPPertBasicTypes_f, only: C_QCPERTINT
```

```
                integer(kind=C_QINT), value, intent(in) :: oper_num_pert
                integer(kind=C_QCPERTINT), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=C_QINT), intent(in) :: oper_pert_orders(oper_num_pert)
                integer(kind=C_QINT), value, intent(in) :: dmat_len_tuple
                integer(kind=C_QINT), intent(in) :: num_LHS_dmat(dmat_len_tuple)
                type(C_PTR), intent(in) :: LHS_dens_mat(sum(num_LHS_dmat))
                integer(kind=C_QINT), intent(in) :: num_RHS_dmat(dmat_len_tuple)
                type(C_PTR), intent(in) :: RHS_dens_mat(sum(num_RHS_dmat))
                type(C_PTR), value, intent(in) :: user_ctx
                integer(kind=C_QINT), value, intent(in) :: num_exp
                real(kind=C_QREAL), intent(inout) :: val_exp(num_exp)
            end subroutine RSPTwoOperGetExp_f
        end interface
        type(TwoOperList_f), pointer :: cur_two_oper  !current two-electron operator
        ! inserts the context of callback functions to the tail of the linked list
        if (associated(open_rsp%list_two_oper)) then
            cur_two_oper => open_rsp%list_two_oper
            do while (associated(cur_two_oper%next_two_oper))
                cur_two_oper => cur_two_oper%next_two_oper
            end do
            allocate(cur_two_oper%next_two_oper)
            cur_two_oper => cur_two_oper%next_two_oper
        else
            allocate(open_rsp%list_two_oper)
            cur_two_oper => open_rsp%list_two_oper
        end if
        allocate(cur_two_oper%two_oper_fun)
        nullify(cur_two_oper%next_two_oper)
        ! adds context of callback functions of the new two-electron operator
        call RSPTwoOperCreate_f(cur_two_oper%two_oper_fun, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                user_ctx,                  &
#endif
                                get_two_oper_mat,          &
                                get_two_oper_exp)
        ierr = OpenRSPAddTwoOper(open_rsp%c_rsp,                   &
                                 num_pert_lab,                     &
                                 pert_labels,                      &
                                 pert_max_orders,                  &
                                 c_loc(cur_two_oper%two_oper_fun), &
                                 c_funloc(RSPTwoOperGetMat_f),     &
                                 c_funloc(RSPTwoOperGetExp_f))
    end function OpenRSPAddTwoOper_f

    function OpenRSPAddXCFun_f(open_rsp,        &
                               num_pert_lab,    &
                               pert_labels,     &
                               pert_max_orders, &
#if defined(OPENRSP_F_USER_CONTEXT)
                               user_ctx,        &
#endif
                               get_xc_fun_mat,  &
```

```
                                get_xc_fun_exp) result(ierr)
        integer(kind=4) :: ierr
        type(OpenRSP), intent(inout) :: open_rsp
        integer(kind=QINT), intent(in) :: num_pert_lab
        integer(kind=QcPertInt), intent(in) :: pert_labels(num_pert_lab)
        integer(kind=QINT), intent(in) :: pert_max_orders(num_pert_lab)
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        interface
            subroutine get_xc_fun_mat(xc_len_tuple,     &
                                      xc_pert_tuple,    &
                                      num_freq_configs, &
                                      dmat_num_tuple,   &
                                      dmat_idx_tuple,   &
                                      num_dmat,         &
                                      dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                      len_ctx,          &
                                      user_ctx,         &
#endif
                                      num_int,          &
                                      val_int)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: xc_len_tuple
                integer(kind=QcPertInt), intent(in) :: xc_pert_tuple(xc_len_tuple)
                integer(kind=QINT), intent(in) :: num_freq_configs
                integer(kind=QINT), intent(in) :: dmat_num_tuple
                integer(kind=QINT), intent(in) :: dmat_idx_tuple(dmat_num_tuple)
                integer(kind=QINT), intent(in) :: num_dmat
                type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_int
                type(QcMat), intent(inout) :: val_int(num_int)
            end subroutine get_xc_fun_mat
            subroutine get_xc_fun_exp(xc_len_tuple,     &
                                      xc_pert_tuple,    &
                                      num_freq_configs, &
                                      dmat_num_tuple,   &
                                      dmat_idx_tuple,   &
                                      num_dmat,         &
                                      dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                      len_ctx,          &
                                      user_ctx,         &
#endif
                                      num_exp,          &
                                      val_exp)
```

```
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: xc_len_tuple
                integer(kind=QcPertInt), intent(in) :: xc_pert_tuple(xc_len_tuple)
                integer(kind=QINT), intent(in) :: num_freq_configs
                integer(kind=QINT), intent(in) :: dmat_num_tuple
                integer(kind=QINT), intent(in) :: dmat_idx_tuple(dmat_num_tuple)
                integer(kind=QINT), intent(in) :: num_dmat
                type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_exp
                real(kind=QREAL), intent(inout) :: val_exp(num_exp)
            end subroutine get_xc_fun_exp
            subroutine RSPXCFunGetMat_f(xc_len_tuple,     &
                                        xc_pert_tuple,    &
                                        num_freq_configs, &
                                        dmat_num_tuple,   &
                                        dmat_idx_tuple,   &
                                        num_dmat,         &
                                        dens_mat,         &
                                        user_ctx,         &
                                        num_int,          &
                                        val_int)          &
                bind(C, name="RSPXCFunGetMat_f")
                use, intrinsic :: iso_c_binding
                use RSPPertBasicTypes_f, only: C_QCPERTINT
                integer(kind=C_QINT), value, intent(in) :: xc_len_tuple
                integer(kind=C_QCPERTINT), intent(in) :: xc_pert_tuple(xc_len_tuple)
                integer(kind=C_QINT), value, intent(in) :: num_freq_configs
                integer(kind=C_QINT), value, intent(in) :: dmat_num_tuple
                integer(kind=C_QINT), intent(in) :: dmat_idx_tuple(dmat_num_tuple)
                integer(kind=C_QINT), value, intent(in) :: num_dmat
                type(C_PTR), intent(in) :: dens_mat(num_dmat)
                type(C_PTR), value, intent(in) :: user_ctx
                integer(kind=C_QINT), value, intent(in) :: num_int
                type(C_PTR), intent(inout) :: val_int(num_int)
            end subroutine RSPXCFunGetMat_f
            subroutine RSPXCFunGetExp_f(xc_len_tuple,     &
                                        xc_pert_tuple,    &
                                        num_freq_configs, &
                                        dmat_num_tuple,   &
                                        dmat_idx_tuple,   &
                                        num_dmat,         &
                                        dens_mat,         &
                                        user_ctx,         &
                                        num_exp,          &
                                        val_exp)          &
                bind(C, name="RSPXCFunGetExp_f")
                use, intrinsic :: iso_c_binding
```

```
                use RSPPertBasicTypes_f, only: C_QCPERTINT
                integer(kind=C_QINT), value, intent(in) :: xc_len_tuple
                integer(kind=C_QCPERTINT), intent(in) :: xc_pert_tuple(xc_len_tuple)
                integer(kind=C_QINT), value, intent(in) :: num_freq_configs
                integer(kind=C_QINT), value, intent(in) :: dmat_num_tuple
                integer(kind=C_QINT), intent(in) :: dmat_idx_tuple(dmat_num_tuple)
                integer(kind=C_QINT), value, intent(in) :: num_dmat
                type(C_PTR), intent(in) :: dens_mat(num_dmat)
                type(C_PTR), value, intent(in) :: user_ctx
                integer(kind=C_QINT), value, intent(in) :: num_exp
                real(kind=C_QREAL), intent(inout) :: val_exp(num_exp)
            end subroutine RSPXCFunGetExp_f
        end interface
        type(XCFunList_f), pointer :: cur_xc_fun  !current XC functional
        ! inserts the context of callback functions to the tail of the linked list
        if (associated(open_rsp%list_xc_fun)) then
            cur_xc_fun => open_rsp%list_xc_fun
            do while (associated(cur_xc_fun%next_xc_fun))
                cur_xc_fun => cur_xc_fun%next_xc_fun
            end do
            allocate(cur_xc_fun%next_xc_fun)
            cur_xc_fun => cur_xc_fun%next_xc_fun
        else
            allocate(open_rsp%list_xc_fun)
            cur_xc_fun => open_rsp%list_xc_fun
        end if
        allocate(cur_xc_fun%xcfun_fun)
        nullify(cur_xc_fun%next_xc_fun)
        ! adds context of callback functions of the new XC functional
        call RSPXCFunCreate_f(cur_xc_fun%xcfun_fun, &
#if defined(OPENRSP_F_USER_CONTEXT)
                              user_ctx,             &
#endif
                              get_xc_fun_mat,       &
                              get_xc_fun_exp)
        ierr = OpenRSPAddXCFun(open_rsp%c_rsp,              &
                               num_pert_lab,                &
                               pert_labels,                 &
                               pert_max_orders,             &
                               c_loc(cur_xc_fun%xcfun_fun), &
                               c_funloc(RSPXCFunGetMat_f),  &
                               c_funloc(RSPXCFunGetExp_f))
    end function OpenRSPAddXCFun_f

    function OpenRSPSetNucHamilton_f(open_rsp,        &
                                     num_pert_lab,    &
                                     pert_labels,     &
                                     pert_max_orders, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                     user_ctx,        &
#endif
                                     get_nuc_contrib, &
```

```
                                    num_atoms) result(ierr)
        integer(kind=4) :: ierr
        type(OpenRSP), intent(inout) :: open_rsp
        integer(kind=QINT), intent(in) :: num_pert_lab
        integer(kind=QcPertInt), intent(in) :: pert_labels(num_pert_lab)
        integer(kind=QINT), intent(in) :: pert_max_orders(num_pert_lab)
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        integer(kind=QINT), intent(in) :: num_atoms
        interface
            subroutine get_nuc_contrib(nuc_num_pert,    &
                                       nuc_pert_labels, &
                                       nuc_pert_orders, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                       len_ctx,         &
                                       user_ctx,        &
#endif
                                       size_pert,       &
                                       val_nuc)
                use qcmatrix_f, only: QINT,QREAL
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: nuc_num_pert
                integer(kind=QcPertInt), intent(in) :: nuc_pert_labels(nuc_num_pert)
                integer(kind=QINT), intent(in) :: nuc_pert_orders(nuc_num_pert)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: size_pert
                real(kind=QREAL), intent(inout) :: val_nuc(size_pert)
            end subroutine get_nuc_contrib
            subroutine RSPNucHamiltonGetContributions_f(nuc_num_pert,    &
                                                        nuc_pert_labels, &
                                                        nuc_pert_orders, &
                                                        user_ctx,        &
                                                        size_pert,       &
                                                        val_nuc)         &
                bind(C, name="RSPNucHamiltonGetContributions_f")
                use, intrinsic :: iso_c_binding
                use RSPPertBasicTypes_f, only: C_QCPERTINT
                integer(kind=C_QINT), value, intent(in) :: nuc_num_pert
                integer(kind=C_QCPERTINT), intent(in) :: nuc_pert_labels(nuc_num_pert)
                integer(kind=C_QINT), intent(in) :: nuc_pert_orders(nuc_num_pert)
                type(C_PTR), value, intent(in) :: user_ctx
                integer(kind=C_QINT), value, intent(in) :: size_pert
                real(kind=C_QREAL), intent(inout) :: val_nuc(size_pert)
            end subroutine RSPNucHamiltonGetContributions_f
        end interface
        if (associated(open_rsp%nuc_hamilton_fun)) then
            call RSPNucHamiltonDestroy_f(open_rsp%nuc_hamilton_fun)
        else
```

```
            allocate(open_rsp%nuc_hamilton_fun)
        end if
        ! adds context of callback function of the nuclear Hamiltonian
        call RSPNucHamiltonCreate_f(open_rsp%nuc_hamilton_fun, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                    user_ctx,                  &
#endif
                                    get_nuc_contrib)
        ierr = OpenRSPSetNucHamilton(open_rsp%c_rsp,                              &
                                     num_pert_lab,                                &
                                     pert_labels,                                 &
                                     pert_max_orders,                             &
                                     c_loc(open_rsp%nuc_hamilton_fun),            &
                                     c_funloc(RSPNucHamiltonGetContributions_f), &
                                     num_atoms)
    end function OpenRSPSetNucHamilton_f

    function OpenRSPAssemble_f(open_rsp) result(ierr)
        integer(kind=4) :: ierr
        type(OpenRSP), intent(inout) :: open_rsp
        ierr = OpenRSPAssemble(open_rsp%c_rsp)
    end function OpenRSPAssemble_f

    function OpenRSPWrite_f(open_rsp, file_name) result(ierr)
        integer(kind=4) :: ierr
        type(OpenRSP), intent(in) :: open_rsp
        character*(*), intent(in) :: file_name
        ierr = OpenRSPWrite(open_rsp%c_rsp, file_name//C_NULL_CHAR)
    end function OpenRSPWrite_f

    function OpenRSPGetRSPFun_f(open_rsp,         &
                                ref_ham,          &
                                ref_state,        &
                                ref_overlap,      &
                                num_props,        &
                                len_tuple,        &
                                pert_tuple,       &
                                num_freq_configs, &
                                pert_freqs,       &
                                kn_rules,         &
                                size_rsp_funs,    &
                                rsp_funs) result(ierr)
        integer(kind=4) :: ierr
        type(OpenRSP), intent(in) :: open_rsp
        type(QcMat), target, intent(in) :: ref_ham
        type(QcMat), target, intent(in) :: ref_state
        type(QcMat), target, intent(in) :: ref_overlap
        integer(kind=QINT), intent(in) :: num_props
        integer(kind=QINT), intent(in) :: len_tuple(num_props)
        integer(kind=QcPertInt), intent(in) :: pert_tuple(sum(len_tuple))
        integer(kind=QINT), intent(in) :: num_freq_configs(num_props)
        real(kind=QREAL), intent(in) :: pert_freqs(2*dot_product(len_tuple,num_freq_configs
```

```
    integer(kind=QINT), intent(in) :: kn_rules(num_props)
    integer(kind=QINT), intent(in) :: size_rsp_funs
    real(kind=QREAL), intent(out) :: rsp_funs(2*size_rsp_funs)
    type(C_PTR) c_ref_ham(1)
    type(C_PTR) c_ref_state(1)
    type(C_PTR) c_ref_overlap(1)
    ierr = QcMat_C_LOC((/ref_ham/), c_ref_ham)
    if (ierr==QFAILURE) return
    ierr = QcMat_C_LOC((/ref_state/), c_ref_state)
    if (ierr==QFAILURE) return
    ierr = QcMat_C_LOC((/ref_overlap/), c_ref_overlap)
    if (ierr==QFAILURE) return
    ierr = OpenRSPGetRSPFun(open_rsp%c_rsp,    &
                            c_ref_ham(1),      &
                            c_ref_state(1),    &
                            c_ref_overlap(1),  &
                            num_props,         &
                            len_tuple,         &
                            pert_tuple,        &
                            num_freq_configs,  &
                            pert_freqs,        &
                            kn_rules,          &
                            size_rsp_funs,     &
                            rsp_funs)
    c_ref_ham(1) = C_NULL_PTR
    c_ref_state(1) = C_NULL_PTR
    c_ref_overlap(1) = C_NULL_PTR
end function OpenRSPGetRSPFun_f

function OpenRSPDestroy_f(open_rsp) result(ierr)
    integer(kind=4) :: ierr
    type(OpenRSP), intent(inout) :: open_rsp
    type(OneOperList_f), pointer :: cur_one_oper   !current one-electron operator
    type(OneOperList_f), pointer :: next_one_oper  !next one-electron operator
    type(TwoOperList_f), pointer :: cur_two_oper   !current two-electron operator
    type(TwoOperList_f), pointer :: next_two_oper  !next two-electron operator
    type(XCFunList_f), pointer :: cur_xc_fun       !current XC functional
    type(XCFunList_f), pointer :: next_xc_fun      !next XC functional
    ierr = OpenRSPDestroyFortranAdapter(open_rsp%c_rsp)
    ! cleans up callback subroutine of response equation solver
    if (associated(open_rsp%solver_fun)) then
        call RSPSolverDestroy_f(open_rsp%solver_fun)
        deallocate(open_rsp%solver_fun)
        nullify(open_rsp%solver_fun)
    end if
    ! cleans up callback subroutines of perturbations
    if (associated(open_rsp%pert_fun)) then
        call RSPPertDestroy_f(open_rsp%pert_fun)
        deallocate(open_rsp%pert_fun)
        nullify(open_rsp%pert_fun)
    end if
    ! cleans up callback subroutines of overlap integrals
```

```
    if (associated(open_rsp%overlap_fun)) then
        call RSPOverlapDestroy_f(open_rsp%overlap_fun)
        deallocate(open_rsp%overlap_fun)
        nullify(open_rsp%overlap_fun)
    end if
    ! cleans up the linked list of context of callback subroutines of one-electron oper
    cur_one_oper => open_rsp%list_one_oper
    do while (associated(cur_one_oper))
        next_one_oper => cur_one_oper%next_one_oper
        if (associated(cur_one_oper%one_oper_fun)) then
            call RSPOneOperDestroy_f(cur_one_oper%one_oper_fun)
            deallocate(cur_one_oper%one_oper_fun)
            nullify(cur_one_oper%one_oper_fun)
        end if
        deallocate(cur_one_oper)
        nullify(cur_one_oper)
        cur_one_oper => next_one_oper
    end do
    ! cleans up the linked list of context of callback subroutines of two-electron oper
    cur_two_oper => open_rsp%list_two_oper
    do while (associated(cur_two_oper))
        next_two_oper => cur_two_oper%next_two_oper
        if (associated(cur_two_oper%two_oper_fun)) then
            call RSPTwoOperDestroy_f(cur_two_oper%two_oper_fun)
            deallocate(cur_two_oper%two_oper_fun)
            nullify(cur_two_oper%two_oper_fun)
        end if
        deallocate(cur_two_oper)
        nullify(cur_two_oper)
        cur_two_oper => next_two_oper
    end do
    ! cleans up the linked list of context of callback subroutines of XC functionals
    cur_xc_fun => open_rsp%list_xc_fun
    do while (associated(cur_xc_fun))
        next_xc_fun => cur_xc_fun%next_xc_fun
        if (associated(cur_xc_fun%xcfun_fun)) then
            call RSPXCFunDestroy_f(cur_xc_fun%xcfun_fun)
            deallocate(cur_xc_fun%xcfun_fun)
            nullify(cur_xc_fun%xcfun_fun)
        end if
        deallocate(cur_xc_fun)
        nullify(cur_xc_fun)
        cur_xc_fun => next_xc_fun
    end do
    ! cleans up callback subroutine of nuclear Hamiltonian
    if (associated(open_rsp%nuc_hamilton_fun)) then
        call RSPNucHamiltonDestroy_f(open_rsp%nuc_hamilton_fun)
        deallocate(open_rsp%nuc_hamilton_fun)
        nullify(open_rsp%nuc_hamilton_fun)
    end if
end function OpenRSPDestroy_f
```

```
end module OpenRSP_f
```

141a ⟨*RSPPertBasicTypes.F90* 141a⟩≡

```
module RSPPertBasicTypes_f
    use, intrinsic :: iso_c_binding

    implicit none

    ! <datatype name='QcPertInt'>
    !   Data type of integers to represent perturbation labels
    ! </datatype>
    ! <datatype name='C_QCPERTINT'>
    !   Integers of perturbation labels to interoperates with C code
    ! </datatype>
    ! <constant name='QCPERTINT_MAX'>
    !   Maximal value of an object of the <QcPertInt> type
    ! </constant>
    !integer(kind=4), parameter, public :: QcPertInt = 8
    !integer, parameter, public :: C_QCPERTINT = C_LONG
    integer(kind=4), parameter, public :: QcPertInt = 4
    integer, parameter, public :: C_QCPERTINT = C_INT
end module RSPPertBasicTypes_f
```

141b ⟨*RSPPerturbation.F90* 141b⟩≡

⟨*OpenRSPLicenseFortran* 116a⟩

```
!!  2014-08-18, Bin Gao
!!  * first version

! basic data types
#include "api/qcmatrix_c_type.h"

module RSPPert_f

    use, intrinsic :: iso_c_binding
    use qcmatrix_f, only: QINT
    use RSPPertBasicTypes_f, only: QcPertInt, &
                                   C_QCPERTINT

    implicit none

    integer(kind=4), private, parameter :: STDOUT = 6

    ! user specified callback subroutines
    abstract interface
        subroutine GetPertConcatenation_f(pert_label,     &
                                          first_cat_comp, &
                                          num_cat_comps,  &
                                          num_sub_tuples, &
                                          len_sub_tuples, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                          len_ctx,        &
```

```
                                        user_ctx,       &
#endif
                                        rank_sub_comps)
            use qcmatrix_f, only: QINT
            use RSPPertBasicTypes_f, only: QcPertInt
            integer(kind=QcPertInt), intent(in) :: pert_label
            integer(kind=QINT), intent(in) :: first_cat_comp
            integer(kind=QINT), intent(in) :: num_cat_comps
            integer(kind=QINT), intent(in) :: num_sub_tuples
            integer(kind=QINT), intent(in) :: len_sub_tuples(num_sub_tuples)
#if defined(OPENRSP_F_USER_CONTEXT)
            integer(kind=QINT), intent(in) :: len_ctx
            character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
            integer(kind=QINT), intent(out) :: rank_sub_comps(num_sub_tuples*num_cat_comps)
        end subroutine GetPertConcatenation_f
    end interface

    ! context of callback subroutine of response equation solver
    type, public :: PertFun_f
        private
#if defined(OPENRSP_F_USER_CONTEXT)
        ! user-defined callback function context
        integer(kind=QINT) :: len_ctx = 0
        character(len=1), allocatable :: user_ctx(:)
#endif
        ! callback functions
        procedure(GetPertConcatenation_f), nopass, pointer :: get_pert_concatenation
    end type PertFun_f

    public :: RSPPertCreate_f
    public :: RSPPertGetConcatenation_f
    public :: RSPPertDestroy_f

    contains

    !% \brief creates the context of callback subroutines of perturbations
    !  \author Bin Gao
    !  \date 2014-08-18
    !  \param[PertFun_f:type]{inout} pert_fun the context of callback subroutines
    !  \param[character]{in} user_ctx user-defined callback function context
    !  \param[subroutine]{in} get_pert_concatenation user specified function for
    !%     getting rank of a perturbation
    subroutine RSPPertCreate_f(pert_fun,      &
#if defined(OPENRSP_F_USER_CONTEXT)
                               user_ctx,      &
#endif
                               get_pert_concatenation)
        type(PertFun_f), intent(inout) :: pert_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
```

```
        interface
            subroutine get_pert_concatenation(pert_label,     &
                                              first_cat_comp, &
                                              num_cat_comps,  &
                                              num_sub_tuples, &
                                              len_sub_tuples, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                              len_ctx,        &
                                              user_ctx,       &
#endif
                                              rank_sub_comps)
                use qcmatrix_f, only: QINT
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QcPertInt), intent(in) :: pert_label
                integer(kind=QINT), intent(in) :: first_cat_comp
                integer(kind=QINT), intent(in) :: num_cat_comps
                integer(kind=QINT), intent(in) :: num_sub_tuples
                integer(kind=QINT), intent(in) :: len_sub_tuples(num_sub_tuples)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(out) :: rank_sub_comps(num_sub_tuples*num_cat_co
            end subroutine get_pert_concatenation
        end interface
#if defined(OPENRSP_F_USER_CONTEXT)
        integer(kind=4) ierr  !error information
        pert_fun%len_ctx = size(user_ctx)
        allocate(pert_fun%user_ctx(pert_fun%len_ctx), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPPertCreate_f>> length", pert_fun%len_ctx
            stop "RSPPertCreate_f>> failed to allocate memory for user_ctx"
        end if
        pert_fun%user_ctx = user_ctx
#endif
        pert_fun%get_pert_concatenation => get_pert_concatenation
    end subroutine RSPPertCreate_f

    !% \brief calls Fortran callback subroutine to get the rank of
    !      a perturbation with its components
    !  \author Bin Gao
    !  \date 2014-08-18
    !  \param[integer]{in} pert_label lable of the perturbation
    !  \param[integer]{in} pert_num_comp number of components of the perturbation
    !  \param[integer]{in} pert_components components of the perturbation
    !  \param[integer]{in} pert_comp_orders orders of the components
    !  \param[C_PTR:type]{in} user_ctx user-defined callback function context
    !% \param[integer]{out} pert_rank the rank of the perturbation
    subroutine RSPPertGetConcatenation_f(pert_label,     &
                                         first_cat_comp, &
                                         num_cat_comps,  &
                                         num_sub_tuples, &
```

```
                                        len_sub_tuples, &
                                        user_ctx,       &
                                        rank_sub_comps) &
        bind(C, name="RSPPertGetConcatenation_f")
        integer(kind=C_QCPERTINT), value, intent(in) :: pert_label
        integer(kind=C_QINT), value, intent(in) :: first_cat_comp
        integer(kind=C_QINT), value, intent(in) :: num_cat_comps
        integer(kind=C_QINT), value, intent(in) :: num_sub_tuples
        integer(kind=C_QINT), intent(in) :: len_sub_tuples(num_sub_tuples)
        type(C_PTR), value, intent(in) :: user_ctx
        integer(kind=C_QINT), intent(out) :: rank_sub_comps(num_sub_tuples*num_cat_comps)
        type(PertFun_f), pointer :: pert_fun   !context of callback subroutines
        ! gets the Fortran callback subroutine
        call c_f_pointer(user_ctx, pert_fun)
        ! invokes Fortran callback subroutine to get the rank of the perturbation
        call pert_fun%get_pert_concatenation(pert_label,        &
                                             first_cat_comp,    &
                                             num_cat_comps,     &
                                             num_sub_tuples,    &
                                             len_sub_tuples,    &
#if defined(OPENRSP_F_USER_CONTEXT)
                                             pert_fun%len_ctx,  &
                                             pert_fun%user_ctx, &
#endif
                                             rank_sub_comps)
        ! cleans up
        nullify(pert_fun)
        return
    end subroutine RSPPertGetConcatenation_f

    !% \brief cleans the context of callback subroutines of perturbations
    !  \author Bin Gao
    !  \date 2014-08-18
    !% \param[PertFun_f:type]{inout} pert_fun the context of callback subroutines
    subroutine RSPPertDestroy_f(pert_fun)
        type(PertFun_f), intent(inout) :: pert_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        pert_fun%len_ctx = 0
        deallocate(pert_fun%user_ctx)
#endif
        nullify(pert_fun%get_pert_concatenation)
    end subroutine RSPPertDestroy_f

end module RSPPert_f
```

144 ⟨*RSPOverlap.F90* 144⟩≡
⟨*OpenRSPLicenseFortran* 116a⟩

```
!!  2014-08-05, Bin Gao
!!  * first version

! basic data types
```

```
#include "api/qcmatrix_c_type.h"

#define OPENRSP_API_SRC "src/fortran/RSPOverlap.F90"

module RSPOverlap_f

    use, intrinsic :: iso_c_binding
    use qcmatrix_f, only: QINT,QREAL,QcMat,QcMat_C_F_POINTER,QcMat_C_NULL_PTR
    use RSPPertBasicTypes_f, only: QcPertInt, &
                                   C_QCPERTINT

    implicit none

    integer(kind=4), private, parameter :: STDOUT = 6

    ! user specified callback subroutines
    abstract interface
        subroutine OverlapGetMat_f(bra_num_pert,     &
                                   bra_pert_labels,  &
                                   bra_pert_orders,  &
                                   ket_num_pert,     &
                                   ket_pert_labels,  &
                                   ket_pert_orders,  &
                                   oper_num_pert,    &
                                   oper_pert_labels, &
                                   oper_pert_orders, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                   len_ctx,          &
                                   user_ctx,         &
#endif
                                   num_int,          &
                                   val_int)
            use qcmatrix_f, only: QINT,QREAL,QcMat
            use RSPPertBasicTypes_f, only: QcPertInt
            integer(kind=QINT), intent(in) :: bra_num_pert
            integer(kind=QcPertInt), intent(in) :: bra_pert_labels(bra_num_pert)
            integer(kind=QINT), intent(in) :: bra_pert_orders(bra_num_pert)
            integer(kind=QINT), intent(in) :: ket_num_pert
            integer(kind=QcPertInt), intent(in) :: ket_pert_labels(ket_num_pert)
            integer(kind=QINT), intent(in) :: ket_pert_orders(ket_num_pert)
            integer(kind=QINT), intent(in) :: oper_num_pert
            integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
            integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
#if defined(OPENRSP_F_USER_CONTEXT)
            integer(kind=QINT), intent(in) :: len_ctx
            character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
            integer(kind=QINT), intent(in) :: num_int
            type(QcMat), intent(inout) :: val_int(num_int)
        end subroutine OverlapGetMat_f
        subroutine OverlapGetExp_f(bra_num_pert,     &
                                   bra_pert_labels,  &
```

```
                                    bra_pert_orders,  &
                                    ket_num_pert,     &
                                    ket_pert_labels,  &
                                    ket_pert_orders,  &
                                    oper_num_pert,    &
                                    oper_pert_labels, &
                                    oper_pert_orders, &
                                    num_dmat,         &
                                    dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                    len_ctx,          &
                                    user_ctx,         &
#endif
                                    num_exp,          &
                                    val_exp)
            use qcmatrix_f, only: QINT,QREAL,QcMat
            use RSPPertBasicTypes_f, only: QcPertInt
            integer(kind=QINT), intent(in) :: bra_num_pert
            integer(kind=QcPertInt), intent(in) :: bra_pert_labels(bra_num_pert)
            integer(kind=QINT), intent(in) :: bra_pert_orders(bra_num_pert)
            integer(kind=QINT), intent(in) :: ket_num_pert
            integer(kind=QcPertInt), intent(in) :: ket_pert_labels(ket_num_pert)
            integer(kind=QINT), intent(in) :: ket_pert_orders(ket_num_pert)
            integer(kind=QINT), intent(in) :: oper_num_pert
            integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
            integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
            integer(kind=QINT), intent(in) :: num_dmat
            type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
            integer(kind=QINT), intent(in) :: len_ctx
            character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
            integer(kind=QINT), intent(in) :: num_exp
            real(kind=QREAL), intent(inout) :: val_exp(num_exp)
        end subroutine OverlapGetExp_f
    end interface

    ! context of callback subroutines of overlap integrals
    type, public :: OverlapFun_f
        private
#if defined(OPENRSP_F_USER_CONTEXT)
        ! user-defined callback function context
        integer(kind=QINT) :: len_ctx = 0
        character(len=1), allocatable :: user_ctx(:)
#endif
        ! callback functions
        procedure(OverlapGetMat_f), nopass, pointer :: get_overlap_mat
        procedure(OverlapGetExp_f), nopass, pointer :: get_overlap_exp
    end type OverlapFun_f

    public :: RSPOverlapCreate_f
    public :: RSPOverlapGetMat_f
```

```
    public :: RSPOverlapGetExp_f
    public :: RSPOverlapDestroy_f

    contains

    !% \brief creates the context of callback subroutines of overlap integrals
    !  \author Bin Gao
    !  \date 2014-08-05
    !  \param[OverlapFun_f:type]{inout} overlap_fun the context of callback subroutines
    !  \param[character]{in} user_ctx user-defined callback function context
    !  \param[subroutine]{in} get_overlap_mat user specified function for
    !      getting integral matrices
    !  \param[subroutine]{in} get_overlap_exp user specified function for
    !     getting expectation values
    subroutine RSPOverlapCreate_f(overlap_fun,     &
#if defined(OPENRSP_F_USER_CONTEXT)
                                  user_ctx,        &
#endif
                                  get_overlap_mat, &
                                  get_overlap_exp)
        type(OverlapFun_f), intent(inout) :: overlap_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        interface
            subroutine get_overlap_mat(bra_num_pert,     &
                                       bra_pert_labels,  &
                                       bra_pert_orders,  &
                                       ket_num_pert,     &
                                       ket_pert_labels,  &
                                       ket_pert_orders,  &
                                       oper_num_pert,    &
                                       oper_pert_labels, &
                                       oper_pert_orders, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                       len_ctx,          &
                                       user_ctx,         &
#endif
                                       num_int,          &
                                       val_int)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: bra_num_pert
                integer(kind=QcPertInt), intent(in) :: bra_pert_labels(bra_num_pert)
                integer(kind=QINT), intent(in) :: bra_pert_orders(bra_num_pert)
                integer(kind=QINT), intent(in) :: ket_num_pert
                integer(kind=QcPertInt), intent(in) :: ket_pert_labels(ket_num_pert)
                integer(kind=QINT), intent(in) :: ket_pert_orders(ket_num_pert)
                integer(kind=QINT), intent(in) :: oper_num_pert
                integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
#if defined(OPENRSP_F_USER_CONTEXT)
```

```
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_int
                type(QcMat), intent(inout) :: val_int(num_int)
            end subroutine get_overlap_mat
            subroutine get_overlap_exp(bra_num_pert,     &
                                       bra_pert_labels,  &
                                       bra_pert_orders,  &
                                       ket_num_pert,     &
                                       ket_pert_labels,  &
                                       ket_pert_orders,  &
                                       oper_num_pert,    &
                                       oper_pert_labels, &
                                       oper_pert_orders, &
                                       num_dmat,         &
                                       dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                       len_ctx,          &
                                       user_ctx,         &
#endif
                                       num_exp,          &
                                       val_exp)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: bra_num_pert
                integer(kind=QcPertInt), intent(in) :: bra_pert_labels(bra_num_pert)
                integer(kind=QINT), intent(in) :: bra_pert_orders(bra_num_pert)
                integer(kind=QINT), intent(in) :: ket_num_pert
                integer(kind=QcPertInt), intent(in) :: ket_pert_labels(ket_num_pert)
                integer(kind=QINT), intent(in) :: ket_pert_orders(ket_num_pert)
                integer(kind=QINT), intent(in) :: oper_num_pert
                integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
                integer(kind=QINT), intent(in) :: num_dmat
                type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_exp
                real(kind=QREAL), intent(inout) :: val_exp(num_exp)
            end subroutine get_overlap_exp
        end interface
#if defined(OPENRSP_F_USER_CONTEXT)
        integer(kind=4) ierr  !error information
        overlap_fun%len_ctx = size(user_ctx)
        allocate(overlap_fun%user_ctx(overlap_fun%len_ctx), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPOverlapCreate_f>> length", overlap_fun%len_ctx
            stop "RSPOverlapCreate_f>> failed to allocate memory for user_ctx"
        end if
```

```
        overlap_fun%user_ctx = user_ctx
#endif
        overlap_fun%get_overlap_mat => get_overlap_mat
        overlap_fun%get_overlap_exp => get_overlap_exp
    end subroutine RSPOverlapCreate_f

    !% \brief calls Fortran callback subroutine to get integral matrices of overlap integra
    !  \author Bin Gao
    !  \date 2014-08-05
    !  \param[integer]{in} bra_len_tuple length of the perturbation tuple on the bra
    !  \param[integer]{in} bra_pert_tuple perturbation tuple on the bra
    !  \param[integer]{in} ket_len_tuple length of the perturbation tuple on the ket
    !  \param[integer]{in} ket_pert_tuple perturbation tuple on the ket
    !  \param[integer]{in} len_tuple length of perturbation tuple on the overlap integrals
    !  \param[integer]{in} pert_tuple perturbation tuple on the overlap integrals
    !  \param[C_PTR:type]{in} user_ctx user-defined callback function context
    !  \param[integer]{in} num_int number of the integral matrices
    !% \param[C_PTR:type]{inout} val_int the integral matrices
    subroutine RSPOverlapGetMat_f(bra_num_pert,     &
                                  bra_pert_labels,  &
                                  bra_pert_orders,  &
                                  ket_num_pert,     &
                                  ket_pert_labels,  &
                                  ket_pert_orders,  &
                                  oper_num_pert,    &
                                  oper_pert_labels, &
                                  oper_pert_orders, &
                                  user_ctx,         &
                                  num_int,          &
                                  val_int)          &
        bind(C, name="RSPOverlapGetMat_f")
        integer(kind=C_QINT), value, intent(in) :: bra_num_pert
        integer(kind=C_QCPERTINT), intent(in) :: bra_pert_labels(bra_num_pert)
        integer(kind=C_QINT), intent(in) :: bra_pert_orders(bra_num_pert)
        integer(kind=C_QINT), value, intent(in) :: ket_num_pert
        integer(kind=C_QCPERTINT), intent(in) :: ket_pert_labels(ket_num_pert)
        integer(kind=C_QINT), intent(in) :: ket_pert_orders(ket_num_pert)
        integer(kind=C_QINT), value, intent(in) :: oper_num_pert
        integer(kind=C_QCPERTINT), intent(in) :: oper_pert_labels(oper_num_pert)
        integer(kind=C_QINT), intent(in) :: oper_pert_orders(oper_num_pert)
        type(C_PTR), value, intent(in) :: user_ctx
        integer(kind=C_QINT), value, intent(in) :: num_int
        type(C_PTR), intent(inout) :: val_int(num_int)
        type(OverlapFun_f), pointer :: overlap_fun  !context of callback subroutines
        type(QcMat), allocatable :: f_val_int(:)    !integral matrices
        integer(kind=4) ierr                        !error information
        ! converts C pointer to Fortran QcMat type
        allocate(f_val_int(num_int), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPOverlapGetMat_f>> num_int", num_int
            stop "RSPOverlapGetMat_f>> failed to allocate memory for f_val_int"
        end if
```

```
        ierr = QcMat_C_F_POINTER(A=f_val_int, c_A=val_int)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        ! gets the Fortran callback subroutine
        call c_f_pointer(user_ctx, overlap_fun)
        ! invokes Fortran callback subroutine to calculate the integral matrices
        call overlap_fun%get_overlap_mat(bra_num_pert,         &
                                         bra_pert_labels,      &
                                         bra_pert_orders,      &
                                         ket_num_pert,         &
                                         ket_pert_labels,      &
                                         ket_pert_orders,      &
                                         oper_num_pert,        &
                                         oper_pert_labels,     &
                                         oper_pert_orders,     &
#if defined(OPENRSP_F_USER_CONTEXT)
                                         overlap_fun%len_ctx,  &
                                         overlap_fun%user_ctx, &
#endif
                                         num_int,              &
                                         f_val_int)
        ! cleans up
        nullify(overlap_fun)
        ierr = QcMat_C_NULL_PTR(A=f_val_int)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        deallocate(f_val_int)
    end subroutine RSPOverlapGetMat_f

    !% \brief calls Fortran callback subroutine to get expectation values of overlap integr
    !  \author Bin Gao
    !  \date 2014-08-05
    !  \param[integer]{in} bra_len_tuple length of the perturbation tuple on the bra
    !  \param[integer]{in} bra_pert_tuple perturbation tuple on the bra
    !  \param[integer]{in} ket_len_tuple length of the perturbation tuple on the ket
    !  \param[integer]{in} ket_pert_tuple perturbation tuple on the ket
    !  \param[integer]{in} len_tuple length of perturbation tuple on the overlap integrals
    !  \param[integer]{in} pert_tuple perturbation tuple on the overlap integrals
    !  \param[integer]{in} num_dmat number of atomic orbital (AO) based density matrices
    !  \param[C_PTR:type]{inout} dens_mat the AO based density matrices
    !  \param[C_PTR:type]{in} user_ctx user-defined callback function context
    !  \param[integer]{in} num_exp number of expectation values
    !% \param[real]{out} val_exp the expectation values
    subroutine RSPOverlapGetExp_f(bra_num_pert,     &
                                  bra_pert_labels,  &
                                  bra_pert_orders,  &
                                  ket_num_pert,     &
                                  ket_pert_labels,  &
                                  ket_pert_orders,  &
                                  oper_num_pert,    &
                                  oper_pert_labels, &
                                  oper_pert_orders, &
                                  num_dmat,         &
                                  dens_mat,         &
```

```
                                user_ctx,         &
                                num_exp,          &
                                val_exp)          &
        bind(C, name="RSPOverlapGetExp_f")
        integer(kind=C_QINT), value, intent(in) :: bra_num_pert
        integer(kind=C_QCPERTINT), intent(in) :: bra_pert_labels(bra_num_pert)
        integer(kind=C_QINT), intent(in) :: bra_pert_orders(bra_num_pert)
        integer(kind=C_QINT), value, intent(in) :: ket_num_pert
        integer(kind=C_QCPERTINT), intent(in) :: ket_pert_labels(ket_num_pert)
        integer(kind=C_QINT), intent(in) :: ket_pert_orders(ket_num_pert)
        integer(kind=C_QINT), value, intent(in) :: oper_num_pert
        integer(kind=C_QCPERTINT), intent(in) :: oper_pert_labels(oper_num_pert)
        integer(kind=C_QINT), intent(in) :: oper_pert_orders(oper_num_pert)
        integer(kind=C_QINT), value, intent(in) :: num_dmat
        type(C_PTR), intent(in) :: dens_mat(num_dmat)
        type(C_PTR), value, intent(in) :: user_ctx
        integer(kind=C_QINT), value, intent(in) :: num_exp
        real(C_QREAL), intent(inout) :: val_exp(num_exp)
        type(OverlapFun_f), pointer :: overlap_fun  !context of callback subroutines
        type(QcMat), allocatable :: f_dens_mat(:)    !AO based density matrices
        integer(kind=4) ierr                        !error information
        ! converts C pointer to Fortran QcMat type
        allocate(f_dens_mat(num_dmat), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPOverlapGetExp_f>> num_dmat", num_dmat
            stop "RSPOverlapGetExp_f>> failed to allocate memory for f_dens_mat"
        end if
        ierr = QcMat_C_F_POINTER(A=f_dens_mat, c_A=dens_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        ! gets the Fortran callback subroutine
        call c_f_pointer(user_ctx, overlap_fun)
        ! invokes Fortran callback subroutine to calculate the expectation values
        call overlap_fun%get_overlap_exp(bra_num_pert,         &
                                         bra_pert_labels,      &
                                         bra_pert_orders,      &
                                         ket_num_pert,         &
                                         ket_pert_labels,      &
                                         ket_pert_orders,      &
                                         oper_num_pert,        &
                                         oper_pert_labels,     &
                                         oper_pert_orders,     &
                                         num_dmat,             &
                                         f_dens_mat,           &
#if defined(OPENRSP_F_USER_CONTEXT)
                                         overlap_fun%len_ctx,  &
                                         overlap_fun%user_ctx, &
#endif
                                         num_exp,              &
                                         val_exp)
        ! cleans up
        nullify(overlap_fun)
        ierr = QcMat_C_NULL_PTR(A=f_dens_mat)
```

```
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        deallocate(f_dens_mat)
        return
    end subroutine RSPOverlapGetExp_f

    !% \brief cleans the context of callback subroutines of overlap integrals
    !  \author Bin Gao
    !  \date 2014-08-05
    !% \param[OverlapFun_f:type]{inout} overlap_fun the context of callback subroutines
    subroutine RSPOverlapDestroy_f(overlap_fun)
        type(OverlapFun_f), intent(inout) :: overlap_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        overlap_fun%len_ctx = 0
        deallocate(overlap_fun%user_ctx)
#endif
        nullify(overlap_fun%get_overlap_mat)
        nullify(overlap_fun%get_overlap_exp)
    end subroutine RSPOverlapDestroy_f

end module RSPOverlap_f

#undef OPENRSP_API_SRC
```

152 ⟨*RSPOneOper.F90* 152⟩≡
⟨*OpenRSPLicenseFortran* 116a⟩

```
!!  2014-08-02, Bin Gao
!!  * first version

! basic data types
#include "api/qcmatrix_c_type.h"

#define OPENRSP_API_SRC "src/fortran/RSPOneOper.F90"

module RSPOneOper_f

    use, intrinsic :: iso_c_binding
    use qcmatrix_f, only: QINT,QREAL,QcMat,QcMat_C_F_POINTER,QcMat_C_NULL_PTR
    use RSPPertBasicTypes_f, only: QcPertInt, &
                                   C_QCPERTINT

    implicit none

    integer(kind=4), private, parameter :: STDOUT = 6

    ! user specified callback subroutines
    abstract interface
        subroutine OneOperGetMat_f(oper_num_pert,    &
                                   oper_pert_labels, &
                                   oper_pert_orders, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                   len_ctx,          &
```

```
                                    user_ctx,         &
#endif
                                    num_int,          &
                                    val_int)
            use qcmatrix_f, only: QINT,QREAL,QcMat
            use RSPPertBasicTypes_f, only: QcPertInt
            integer(kind=QINT), intent(in) :: oper_num_pert
            integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
            integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
#if defined(OPENRSP_F_USER_CONTEXT)
            integer(kind=QINT), intent(in) :: len_ctx
            character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
            integer(kind=QINT), intent(in) :: num_int
            type(QcMat), intent(inout) :: val_int(num_int)
        end subroutine OneOperGetMat_f
        subroutine OneOperGetExp_f(oper_num_pert,    &
                                    oper_pert_labels, &
                                    oper_pert_orders, &
                                    num_dmat,         &
                                    dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                    len_ctx,          &
                                    user_ctx,         &
#endif
                                    num_exp,          &
                                    val_exp)
            use qcmatrix_f, only: QINT,QREAL,QcMat
            use RSPPertBasicTypes_f, only: QcPertInt
            integer(kind=QINT), intent(in) :: oper_num_pert
            integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
            integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
            integer(kind=QINT), intent(in) :: num_dmat
            type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
            integer(kind=QINT), intent(in) :: len_ctx
            character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
            integer(kind=QINT), intent(in) :: num_exp
            real(kind=QREAL), intent(inout) :: val_exp(num_exp)
        end subroutine OneOperGetExp_f
    end interface

    ! context of callback subroutines of one-electron operator
    type, public :: OneOperFun_f
        private
#if defined(OPENRSP_F_USER_CONTEXT)
        ! user-defined callback function context
        integer(kind=QINT) :: len_ctx = 0
        character(len=1), allocatable :: user_ctx(:)
#endif
        ! callback functions
```

```
        procedure(OneOperGetMat_f), nopass, pointer :: get_one_oper_mat
        procedure(OneOperGetExp_f), nopass, pointer :: get_one_oper_exp
    end type OneOperFun_f

    public :: RSPOneOperCreate_f
    public :: RSPOneOperGetMat_f
    public :: RSPOneOperGetExp_f
    public :: RSPOneOperDestroy_f

    contains

    !% \brief creates the context of callback subroutines of one-electron operator
    !  \author Bin Gao
    !  \date 2014-08-03
    !  \param[OneOperFun_f:type]{inout} one_oper_fun the context of callback subroutines
    !  \param[character]{in} user_ctx user-defined callback function context
    !  \param[subroutine]{in} get_one_oper_mat user specified function for
    !      getting integral matrices
    !  \param[subroutine]{in} get_one_oper_exp user specified function for
    !%     getting expectation values
    subroutine RSPOneOperCreate_f(one_oper_fun,     &
#if defined(OPENRSP_F_USER_CONTEXT)
                                  user_ctx,         &
#endif
                                  get_one_oper_mat, &
                                  get_one_oper_exp)
        type(OneOperFun_f), intent(inout) :: one_oper_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        interface
            subroutine get_one_oper_mat(oper_num_pert,    &
                                        oper_pert_labels, &
                                        oper_pert_orders, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                        len_ctx,          &
                                        user_ctx,         &
#endif
                                        num_int,          &
                                        val_int)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: oper_num_pert
                integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_int
                type(QcMat), intent(inout) :: val_int(num_int)
            end subroutine get_one_oper_mat
```

```
            subroutine get_one_oper_exp(oper_num_pert,    &
                                        oper_pert_labels, &
                                        oper_pert_orders, &
                                        num_dmat,         &
                                        dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                        len_ctx,          &
                                        user_ctx,         &
#endif
                                        num_exp,          &
                                        val_exp)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: oper_num_pert
                integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
                integer(kind=QINT), intent(in) :: num_dmat
                type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_exp
                real(kind=QREAL), intent(inout) :: val_exp(num_exp)
            end subroutine get_one_oper_exp
        end interface
#if defined(OPENRSP_F_USER_CONTEXT)
        integer(kind=4) ierr  !error information
        one_oper_fun%len_ctx = size(user_ctx)
        allocate(one_oper_fun%user_ctx(one_oper_fun%len_ctx), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPOneOperCreate_f>> length", one_oper_fun%len_ctx
            stop "RSPOneOperCreate_f>> failed to allocate memory for user_ctx"
        end if
        one_oper_fun%user_ctx = user_ctx
#endif
        one_oper_fun%get_one_oper_mat => get_one_oper_mat
        one_oper_fun%get_one_oper_exp => get_one_oper_exp
    end subroutine RSPOneOperCreate_f

    !% \brief calls Fortran callback subroutine to get integral matrices of
    !      a one-electron operator
    !  \author Bin Gao
    !  \date 2014-08-02
    !  \param[integer]{in} len_tuple length of perturbation tuple on the one-electron opera
    !  \param[integer]{in} pert_tuple perturbation tuple on the one-electron operator
    !  \param[C_PTR:type]{in} user_ctx user-defined callback function context
    !  \param[integer]{in} num_int number of the integral matrices
    !% \param[C_PTR:type]{inout} val_int the integral matrices
    subroutine RSPOneOperGetMat_f(oper_num_pert,    &
                                  oper_pert_labels, &
                                  oper_pert_orders, &
```

```
                                user_ctx,         &
                                num_int,          &
                                val_int)          &
        bind(C, name="RSPOneOperGetMat_f")
        integer(kind=C_QINT), value, intent(in) :: oper_num_pert
        integer(kind=C_QCPERTINT), intent(in) :: oper_pert_labels(oper_num_pert)
        integer(kind=C_QINT), intent(in) :: oper_pert_orders(oper_num_pert)
        type(C_PTR), value, intent(in) :: user_ctx
        integer(kind=C_QINT), value, intent(in) :: num_int
        type(C_PTR), intent(inout) :: val_int(num_int)
        type(OneOperFun_f), pointer :: one_oper_fun  !context of callback subroutines
        type(QcMat), allocatable :: f_val_int(:)     !integral matrices
        integer(kind=4) ierr                         !error information
        ! converts C pointer to Fortran QcMat type
        allocate(f_val_int(num_int), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPOneOperGetMat_f>> num_int", num_int
            stop "RSPOneOperGetMat_f>> failed to allocate memory for f_val_int"
        end if
        ierr = QcMat_C_F_POINTER(A=f_val_int, c_A=val_int)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        ! gets the Fortran callback subroutine
        call c_f_pointer(user_ctx, one_oper_fun)
        ! invokes Fortran callback subroutine to calculate the integral matrices
        call one_oper_fun%get_one_oper_mat(oper_num_pert,         &
                                           oper_pert_labels,      &
                                           oper_pert_orders,      &
#if defined(OPENRSP_F_USER_CONTEXT)
                                           one_oper_fun%len_ctx,  &
                                           one_oper_fun%user_ctx, &
#endif
                                           num_int,               &
                                           f_val_int)
        ! cleans up
        nullify(one_oper_fun)
        ierr = QcMat_C_NULL_PTR(A=f_val_int)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        deallocate(f_val_int)
    end subroutine RSPOneOperGetMat_f

    !% \brief calls Fortran callback subroutine to get expectation values of
    !      a one-electron operator
    !  \author Bin Gao
    !  \date 2014-08-02
    !  \param[integer]{in} len_tuple length of perturbation tuple on the one-electron opera
    !  \param[integer]{in} pert_tuple perturbation tuple on the one-electron operator
    !  \param[integer]{in} num_dmat number of atomic orbital (AO) based density matrices
    !  \param[C_PTR:type]{inout} dens_mat the AO based density matrices
    !  \param[C_PTR:type]{in} user_ctx user-defined callback function context
    !  \param[integer]{in} num_exp number of expectation values
    !% \param[real]{out} val_exp the expectation values
    subroutine RSPOneOperGetExp_f(oper_num_pert,    &
```

```
                                    oper_pert_labels, &
                                    oper_pert_orders, &
                                    num_dmat,         &
                                    dens_mat,         &
                                    user_ctx,         &
                                    num_exp,          &
                                    val_exp)          &
        bind(C, name="RSPOneOperGetExp_f")
        integer(kind=C_QINT), value, intent(in) :: oper_num_pert
        integer(kind=C_QCPERTINT), intent(in) :: oper_pert_labels(oper_num_pert)
        integer(kind=C_QINT), intent(in) :: oper_pert_orders(oper_num_pert)
        integer(kind=C_QINT), value, intent(in) :: num_dmat
        type(C_PTR), intent(in) :: dens_mat(num_dmat)
        type(C_PTR), value, intent(in) :: user_ctx
        integer(kind=C_QINT), value, intent(in) :: num_exp
        real(kind=C_QREAL), intent(inout) :: val_exp(num_exp)
        type(OneOperFun_f), pointer :: one_oper_fun  !context of callback subroutines
        type(QcMat), allocatable :: f_dens_mat(:)     !AO based density matrices
        integer(kind=4) ierr                          !error information
        ! converts C pointer to Fortran QcMat type
        allocate(f_dens_mat(num_dmat), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPOneOperGetExp_f>> num_dmat", num_dmat
            stop "RSPOneOperGetExp_f>> failed to allocate memory for f_dens_mat"
        end if
        ierr = QcMat_C_F_POINTER(A=f_dens_mat, c_A=dens_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        ! gets the Fortran callback subroutine
        call c_f_pointer(user_ctx, one_oper_fun)
        ! invokes Fortran callback subroutine to calculate the expectation values
        call one_oper_fun%get_one_oper_exp(oper_num_pert,         &
                                           oper_pert_labels,      &
                                           oper_pert_orders,      &
                                           num_dmat,              &
                                           f_dens_mat,            &
#if defined(OPENRSP_F_USER_CONTEXT)
                                           one_oper_fun%len_ctx,  &
                                           one_oper_fun%user_ctx, &
#endif
                                           num_exp,               &
                                           val_exp)
        ! cleans up
        nullify(one_oper_fun)
        ierr = QcMat_C_NULL_PTR(A=f_dens_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        deallocate(f_dens_mat)
        return
    end subroutine RSPOneOperGetExp_f

    !% \brief cleans the context of callback subroutines of one-electron operator
    !  \author Bin Gao
    !  \date 2014-08-03
```

```
    !% \param[OneOperFun_f:type]{inout} one_oper_fun the context of callback subroutines
    subroutine RSPOneOperDestroy_f(one_oper_fun)
        type(OneOperFun_f), intent(inout) :: one_oper_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        one_oper_fun%len_ctx = 0
        deallocate(one_oper_fun%user_ctx)
#endif
        nullify(one_oper_fun%get_one_oper_mat)
        nullify(one_oper_fun%get_one_oper_exp)
    end subroutine RSPOneOperDestroy_f

end module RSPOneOper_f

#undef OPENRSP_API_SRC
```

158 ⟨*RSPTwoper.F90* 158⟩≡
⟨*OpenRSPLicenseFortran* 116a⟩

```
!!  2014-08-06, Bin Gao
!!  * first version

! basic data types
#include "api/qcmatrix_c_type.h"

#define OPENRSP_API_SRC "src/fortran/RSPTwoOper.F90"

module RSPTwoOper_f

    use, intrinsic :: iso_c_binding
    use qcmatrix_f, only: QINT,QREAL,QcMat,QcMat_C_F_POINTER,QcMat_C_NULL_PTR
    use RSPPertBasicTypes_f, only: QcPertInt, &
                                   C_QCPERTINT

    implicit none

    integer(kind=4), private, parameter :: STDOUT = 6

    ! user specified callback subroutines
    abstract interface
        subroutine TwoOperGetMat_f(oper_num_pert,    &
                                   oper_pert_labels, &
                                   oper_pert_orders, &
                                   num_dmat,         &
                                   dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                   len_ctx,          &
                                   user_ctx,         &
#endif
                                   num_int,          &
                                   val_int)
            use qcmatrix_f, only: QINT,QREAL,QcMat
            use RSPPertBasicTypes_f, only: QcPertInt
```

```
            integer(kind=QINT), intent(in) :: oper_num_pert
            integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
            integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
            integer(kind=QINT), intent(in) :: num_dmat
            type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
            integer(kind=QINT), intent(in) :: len_ctx
            character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
            integer(kind=QINT), intent(in) :: num_int
            type(QcMat), intent(inout) :: val_int(num_int)
        end subroutine TwoOperGetMat_f
        subroutine TwoOperGetExp_f(oper_num_pert,    &
                                   oper_pert_labels, &
                                   oper_pert_orders, &
                                   dmat_len_tuple,   &
                                   num_LHS_dmat,     &
                                   LHS_dens_mat,     &
                                   num_RHS_dmat,     &
                                   RHS_dens_mat,     &
#if defined(OPENRSP_F_USER_CONTEXT)
                                   len_ctx,          &
                                   user_ctx,         &
#endif
                                   num_exp,          &
                                   val_exp)
            use qcmatrix_f, only: QINT,QREAL,QcMat
            use RSPPertBasicTypes_f, only: QcPertInt
            integer(kind=QINT), intent(in) :: oper_num_pert
            integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
            integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
            integer(kind=QINT), intent(in) :: dmat_len_tuple
            integer(kind=QINT), intent(in) :: num_LHS_dmat(dmat_len_tuple)
            type(QcMat), intent(in) :: LHS_dens_mat(sum(num_LHS_dmat))
            integer(kind=QINT), intent(in) :: num_RHS_dmat(dmat_len_tuple)
            type(QcMat), intent(in) :: RHS_dens_mat(sum(num_RHS_dmat))
#if defined(OPENRSP_F_USER_CONTEXT)
            integer(kind=QINT), intent(in) :: len_ctx
            character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
            integer(kind=QINT), intent(in) :: num_exp
            real(kind=QREAL), intent(inout) :: val_exp(num_exp)
        end subroutine TwoOperGetExp_f
    end interface

    ! context of callback subroutines of two-electron operator
    type, public :: TwoOperFun_f
        private
#if defined(OPENRSP_F_USER_CONTEXT)
        ! user-defined callback function context
        integer(kind=QINT) :: len_ctx = 0
        character(len=1), allocatable :: user_ctx(:)
```

```
#endif
        ! callback functions
        procedure(TwoOperGetMat_f), nopass, pointer :: get_two_oper_mat
        procedure(TwoOperGetExp_f), nopass, pointer :: get_two_oper_exp
    end type TwoOperFun_f

    public :: RSPTwoOperCreate_f
    public :: RSPTwoOperGetMat_f
    public :: RSPTwoOperGetExp_f
    public :: RSPTwoOperDestroy_f

    contains

    !% \brief creates the context of callback subroutines of two-electron operator
    !  \author Bin Gao
    !  \date 2014-08-06
    !  \param[TwoOperFun_f:type]{inout} two_oper_fun the context of callback subroutines
    !  \param[character]{in} user_ctx user-defined callback function context
    !  \param[subroutine]{in} get_two_oper_mat user specified function for
    !      getting integral matrices
    !  \param[subroutine]{in} get_two_oper_exp user specified function for
    !%     getting expectation values
    subroutine RSPTwoOperCreate_f(two_oper_fun,     &
#if defined(OPENRSP_F_USER_CONTEXT)
                                  user_ctx,         &
#endif
                                  get_two_oper_mat, &
                                  get_two_oper_exp)
        type(TwoOperFun_f), intent(inout) :: two_oper_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        interface
            subroutine get_two_oper_mat(oper_num_pert,    &
                                        oper_pert_labels, &
                                        oper_pert_orders, &
                                        num_dmat,         &
                                        dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                        len_ctx,          &
                                        user_ctx,         &
#endif
                                        num_int,          &
                                        val_int)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: oper_num_pert
                integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
                integer(kind=QINT), intent(in) :: num_dmat
                type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
```

```
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_int
                type(QcMat), intent(inout) :: val_int(num_int)
            end subroutine get_two_oper_mat
            subroutine get_two_oper_exp(oper_num_pert,    &
                                        oper_pert_labels, &
                                        oper_pert_orders, &
                                        dmat_len_tuple,   &
                                        num_LHS_dmat,     &
                                        LHS_dens_mat,     &
                                        num_RHS_dmat,     &
                                        RHS_dens_mat,     &
#if defined(OPENRSP_F_USER_CONTEXT)
                                        len_ctx,          &
                                        user_ctx,         &
#endif
                                        num_exp,          &
                                        val_exp)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: oper_num_pert
                integer(kind=QcPertInt), intent(in) :: oper_pert_labels(oper_num_pert)
                integer(kind=QINT), intent(in) :: oper_pert_orders(oper_num_pert)
                integer(kind=QINT), intent(in) :: dmat_len_tuple
                integer(kind=QINT), intent(in) :: num_LHS_dmat(dmat_len_tuple)
                type(QcMat), intent(in) :: LHS_dens_mat(sum(num_LHS_dmat))
                integer(kind=QINT), intent(in) :: num_RHS_dmat(dmat_len_tuple)
                type(QcMat), intent(in) :: RHS_dens_mat(sum(num_RHS_dmat))
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_exp
                real(kind=QREAL), intent(inout) :: val_exp(num_exp)
            end subroutine get_two_oper_exp
        end interface
#if defined(OPENRSP_F_USER_CONTEXT)
        integer(kind=4) ierr  !error information
        two_oper_fun%len_ctx = size(user_ctx)
        allocate(two_oper_fun%user_ctx(two_oper_fun%len_ctx), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPTwoOperCreate_f>> length", two_oper_fun%len_ctx
            stop "RSPTwoOperCreate_f>> failed to allocate memory for user_ctx"
        end if
        two_oper_fun%user_ctx = user_ctx
#endif
        two_oper_fun%get_two_oper_mat => get_two_oper_mat
        two_oper_fun%get_two_oper_exp => get_two_oper_exp
    end subroutine RSPTwoOperCreate_f
```

```
!% \brief calls Fortran callback subroutine to get integral matrices of
!      a two-electron operator
!  \author Bin Gao
!  \date 2014-08-06
!  \param[integer]{in} len_tuple length of perturbation tuple on the two-electron opera
!  \param[integer]{in} pert_tuple perturbation tuple on the two-electron operator
!  \param[integer]{in} num_dmat number of AO based density matrices
!  \param[C_PTR:type]{in} dens_mat the AO based density matrices
!  \param[C_PTR:type]{in} user_ctx user-defined callback function context
!  \param[integer]{in} num_int number of the integral matrices
!% \param[C_PTR:type]{inout} val_int the integral matrices
subroutine RSPTwoOperGetMat_f(oper_num_pert,    &
                              oper_pert_labels, &
                              oper_pert_orders, &
                              num_dmat,         &
                              dens_mat,         &
                              user_ctx,         &
                              num_int,          &
                              val_int)          &
    bind(C, name="RSPTwoOperGetMat_f")
    integer(kind=C_QINT), value, intent(in) :: oper_num_pert
    integer(kind=C_QCPERTINT), intent(in) :: oper_pert_labels(oper_num_pert)
    integer(kind=C_QINT), intent(in) :: oper_pert_orders(oper_num_pert)
    integer(kind=C_QINT), value, intent(in) :: num_dmat
    type(C_PTR), intent(in) :: dens_mat(num_dmat)
    type(C_PTR), value, intent(in) :: user_ctx
    integer(kind=C_QINT), value, intent(in) :: num_int
    type(C_PTR), intent(inout) :: val_int(num_int)
    type(TwoOperFun_f), pointer :: two_oper_fun  !context of callback subroutines
    type(QcMat), allocatable :: f_dens_mat(:)    !AO based density matrices
    type(QcMat), allocatable :: f_val_int(:)     !integral matrices
    integer(kind=4) ierr                         !error information
    ! converts C pointer to Fortran QcMat type
    allocate(f_dens_mat(num_dmat), stat=ierr)
    if (ierr/=0) then
        write(STDOUT,"(A,I8)") "RSPTwoOperGetMat_f>> num_dmat", num_dmat
        stop "RSPTwoOperGetMat_f>> failed to allocate memory for f_dens_mat"
    end if
    ierr = QcMat_C_F_POINTER(A=f_dens_mat, c_A=dens_mat)
    call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
    allocate(f_val_int(num_int), stat=ierr)
    if (ierr/=0) then
        write(STDOUT,"(A,I8)") "RSPTwoOperGetMat_f>> num_int", num_int
        stop "RSPTwoOperGetMat_f>> failed to allocate memory for f_val_int"
    end if
    ierr = QcMat_C_F_POINTER(A=f_val_int, c_A=val_int)
    call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
    ! gets the Fortran callback subroutine
    call c_f_pointer(user_ctx, two_oper_fun)
    ! invokes Fortran callback subroutine to calculate the integral matrices
    call two_oper_fun%get_two_oper_mat(oper_num_pert,         &
                                       oper_pert_labels,      &
```

```
                                          oper_pert_orders,      &
                                          num_dmat,              &
                                          f_dens_mat,            &
#if defined(OPENRSP_F_USER_CONTEXT)
                                          two_oper_fun%len_ctx,  &
                                          two_oper_fun%user_ctx, &
#endif
                                          num_int,               &
                                          f_val_int)
        ! cleans up
        nullify(two_oper_fun)
        ierr = QcMat_C_NULL_PTR(A=f_val_int)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        ierr = QcMat_C_NULL_PTR(A=f_dens_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        deallocate(f_val_int)
        deallocate(f_dens_mat)
    end subroutine RSPTwoOperGetMat_f

    !% \brief calls Fortran callback subroutine to get expectation values of
    !      a two-electron operator
    !  \author Bin Gao
    !  \date 2014-08-06
    !  \param[integer]{in} len_tuple length of perturbation tuple on the two-electron opera
    !  \param[integer]{in} pert_tuple perturbation tuple on the two-electron operator
    !  \param[integer]{in} dmat_len_tuple length of different perturbation tuples
    !      of the left-hand-side (LHS) and right-hand-side (RHS) AO based density
    !      matrices passed
    !  \param[integer]{in} num_LHS_dmat number of LHS AO based density matrices
    !      passed for each LHS density matrix perturbation tuple
    !  \param[C_PTR:type]{in} LHS_dens_mat the LHS AO based density matrices
    !  \param[integer]{in} num_RHS_dmat number of RHS AO based density matrices
    !      passed for each RHS density matrix perturbation tuple
    !  \param[C_PTR:type]{in} RHS_dens_mat the RHS AO based density matrices
    !  \param[C_PTR:type]{in} user_ctx user-defined callback function context
    !  \param[integer]{in} num_exp number of expectation values
    !% \param[real]{out} val_exp the expectation values
    subroutine RSPTwoOperGetExp_f(oper_num_pert,    &
                                  oper_pert_labels, &
                                  oper_pert_orders, &
                                  dmat_len_tuple,   &
                                  num_LHS_dmat,     &
                                  LHS_dens_mat,     &
                                  num_RHS_dmat,     &
                                  RHS_dens_mat,     &
                                  user_ctx,         &
                                  num_exp,          &
                                  val_exp)          &
        bind(C, name="RSPTwoOperGetExp_f")
        integer(kind=C_QINT), value, intent(in) :: oper_num_pert
        integer(kind=C_QCPERTINT), intent(in) :: oper_pert_labels(oper_num_pert)
        integer(kind=C_QINT), intent(in) :: oper_pert_orders(oper_num_pert)
```

```
        integer(kind=C_QINT), value, intent(in) :: dmat_len_tuple
        integer(kind=C_QINT), intent(in) :: num_LHS_dmat(dmat_len_tuple)
        type(C_PTR), intent(in) :: LHS_dens_mat(sum(num_LHS_dmat))
        integer(kind=C_QINT), intent(in) :: num_RHS_dmat(dmat_len_tuple)
        type(C_PTR), intent(in) :: RHS_dens_mat(sum(num_RHS_dmat))
        type(C_PTR), value, intent(in) :: user_ctx
        integer(kind=C_QINT), value, intent(in) :: num_exp
        real(kind=C_QREAL), intent(inout) :: val_exp(num_exp)
        type(TwoOperFun_f), pointer :: two_oper_fun    !context of callback subroutines
        type(QcMat), allocatable :: f_LHS_dens_mat(:)  !LHS AO based density matrices
        type(QcMat), allocatable :: f_RHS_dens_mat(:)  !RHS AO based density matrices
        integer(kind=4) ierr                           !error information
        ! converts C pointer to Fortran QcMat type
        allocate(f_LHS_dens_mat(sum(num_LHS_dmat)), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPTwoOperGetExp_f>> sum(num_LHS_dmat)", &
                                   sum(num_LHS_dmat)
            stop "RSPTwoOperGetExp_f>> failed to allocate memory for f_LHS_dens_mat"
        end if
        ierr = QcMat_C_F_POINTER(A=f_LHS_dens_mat, c_A=LHS_dens_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        allocate(f_RHS_dens_mat(sum(num_RHS_dmat)), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPTwoOperGetExp_f>> sum(num_RHS_dmat)", &
                                   sum(num_RHS_dmat)
            stop "RSPTwoOperGetExp_f>> failed to allocate memory for f_RHS_dens_mat"
        end if
        ierr = QcMat_C_F_POINTER(A=f_RHS_dens_mat, c_A=RHS_dens_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        ! gets the Fortran callback subroutine
        call c_f_pointer(user_ctx, two_oper_fun)
        ! invokes Fortran callback subroutine to calculate the expectation values
        call two_oper_fun%get_two_oper_exp(oper_num_pert,         &
                                           oper_pert_labels,      &
                                           oper_pert_orders,      &
                                           dmat_len_tuple,        &
                                           num_LHS_dmat,          &
                                           f_LHS_dens_mat,        &
                                           num_RHS_dmat,          &
                                           f_RHS_dens_mat,        &
#if defined(OPENRSP_F_USER_CONTEXT)
                                           two_oper_fun%len_ctx,  &
                                           two_oper_fun%user_ctx, &
#endif
                                           num_exp,               &
                                           val_exp)
        ! cleans up
        nullify(two_oper_fun)
        ierr = QcMat_C_NULL_PTR(A=f_RHS_dens_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        ierr = QcMat_C_NULL_PTR(A=f_LHS_dens_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
```

```
        deallocate(f_RHS_dens_mat)
        deallocate(f_LHS_dens_mat)
        return
    end subroutine RSPTwoOperGetExp_f

    !% \brief cleans the context of callback subroutines of two-electron operator
    !  \author Bin Gao
    !  \date 2014-08-06
    !% \param[TwoOperFun_f:type]{inout} two_oper_fun the context of callback subroutines
    subroutine RSPTwoOperDestroy_f(two_oper_fun)
        type(TwoOperFun_f), intent(inout) :: two_oper_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        two_oper_fun%len_ctx = 0
        deallocate(two_oper_fun%user_ctx)
#endif
        nullify(two_oper_fun%get_two_oper_mat)
        nullify(two_oper_fun%get_two_oper_exp)
    end subroutine RSPTwoOperDestroy_f

end module RSPTwoOper_f

#undef OPENRSP_API_SRC
```

165 ⟨*RSPXCFun.F90* 165⟩≡
⟨*OpenRSPLicenseFortran* 116a⟩

```
!!  2015-06-23, Bin Gao
!!  * first version

! basic data types
#include "api/qcmatrix_c_type.h"

#define OPENRSP_API_SRC "src/fortran/RSPXCFun.F90"

module RSPXCFun_f

    use, intrinsic :: iso_c_binding
    use qcmatrix_f, only: QINT,QREAL,QcMat,QcMat_C_F_POINTER,QcMat_C_NULL_PTR
    use RSPPertBasicTypes_f, only: QcPertInt, &
                                   C_QCPERTINT

    implicit none

    integer(kind=4), private, parameter :: STDOUT = 6

    ! user specified callback subroutines
    abstract interface
        subroutine XCFunGetMat_f(xc_len_tuple,     &
                                 xc_pert_tuple,    &
                                 num_freq_configs, &
                                 dmat_num_tuple,   &
                                 dmat_idx_tuple,   &
```

```
                                num_dmat,         &
                                dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                len_ctx,          &
                                user_ctx,         &
#endif
                                num_int,          &
                                val_int)

            use qcmatrix_f, only: QINT,QREAL,QcMat
            use RSPPertBasicTypes_f, only: QcPertInt
            integer(kind=QINT), intent(in) :: xc_len_tuple
            integer(kind=QcPertInt), intent(in) :: xc_pert_tuple(xc_len_tuple)
            integer(kind=QINT), intent(in) :: num_freq_configs
            integer(kind=QINT), intent(in) :: dmat_num_tuple
            integer(kind=QINT), intent(in) :: dmat_idx_tuple(dmat_num_tuple)
            integer(kind=QINT), intent(in) :: num_dmat
            type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
            integer(kind=QINT), intent(in) :: len_ctx
            character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
            integer(kind=QINT), intent(in) :: num_int
            type(QcMat), intent(inout) :: val_int(num_int)
        end subroutine XCFunGetMat_f
        subroutine XCFunGetExp_f(xc_len_tuple,     &
                                xc_pert_tuple,    &
                                num_freq_configs, &
                                dmat_num_tuple,   &
                                dmat_idx_tuple,   &
                                num_dmat,         &
                                dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                len_ctx,          &
                                user_ctx,         &
#endif
                                num_exp,          &
                                val_exp)
            use qcmatrix_f, only: QINT,QREAL,QcMat
            use RSPPertBasicTypes_f, only: QcPertInt
            integer(kind=QINT), intent(in) :: xc_len_tuple
            integer(kind=QcPertInt), intent(in) :: xc_pert_tuple(xc_len_tuple)
            integer(kind=QINT), intent(in) :: num_freq_configs
            integer(kind=QINT), intent(in) :: dmat_num_tuple
            integer(kind=QINT), intent(in) :: dmat_idx_tuple(dmat_num_tuple)
            integer(kind=QINT), intent(in) :: num_dmat
            type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
            integer(kind=QINT), intent(in) :: len_ctx
            character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
            integer(kind=QINT), intent(in) :: num_exp
```

```
            real(kind=QREAL), intent(inout) :: val_exp(num_exp)
        end subroutine XCFunGetExp_f
    end interface

    ! context of callback subroutines of XC functional
    type, public :: XCFunFun_f
        private
#if defined(OPENRSP_F_USER_CONTEXT)
        ! user-defined callback function context
        integer(kind=QINT) :: len_ctx = 0
        character(len=1), allocatable :: user_ctx(:)
#endif
        ! callback functions
        procedure(XCFunGetMat_f), nopass, pointer :: get_xc_fun_mat
        procedure(XCFunGetExp_f), nopass, pointer :: get_xc_fun_exp
    end type XCFunFun_f

    public :: RSPXCFunCreate_f
    public :: RSPXCFunGetMat_f
    public :: RSPXCFunGetExp_f
    public :: RSPXCFunDestroy_f

    contains

    !% \brief creates the context of callback subroutines of XC functional
    !  \author Bin Gao
    !  \date 2015-06-23
    !  \param[XCFunFun_f:type]{inout} xcfun_fun the context of callback subroutines
    !  \param[character]{in} user_ctx user-defined callback function context
    !  \param[subroutine]{in} get_xc_fun_mat user specified function for
    !      getting integral matrices
    !  \param[subroutine]{in} get_xc_fun_exp user specified function for
    !%     getting expectation values
    subroutine RSPXCFunCreate_f(xcfun_fun,      &
#if defined(OPENRSP_F_USER_CONTEXT)
                                user_ctx,       &
#endif
                                get_xc_fun_mat, &
                                get_xc_fun_exp)
        type(XCFunFun_f), intent(inout) :: xcfun_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        interface
            subroutine get_xc_fun_mat(xc_len_tuple,     &
                                      xc_pert_tuple,    &
                                      num_freq_configs, &
                                      dmat_num_tuple,   &
                                      dmat_idx_tuple,   &
                                      num_dmat,         &
                                      dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
```

```
                                    len_ctx,          &
                                    user_ctx,         &
#endif
                                    num_int,          &
                                    val_int)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: xc_len_tuple
                integer(kind=QcPertInt), intent(in) :: xc_pert_tuple(xc_len_tuple)
                integer(kind=QINT), intent(in) :: num_freq_configs
                integer(kind=QINT), intent(in) :: dmat_num_tuple
                integer(kind=QINT), intent(in) :: dmat_idx_tuple(dmat_num_tuple)
                integer(kind=QINT), intent(in) :: num_dmat
                type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_int
                type(QcMat), intent(inout) :: val_int(num_int)
            end subroutine get_xc_fun_mat
            subroutine get_xc_fun_exp(xc_len_tuple,     &
                                    xc_pert_tuple,    &
                                    num_freq_configs, &
                                    dmat_num_tuple,   &
                                    dmat_idx_tuple,   &
                                    num_dmat,         &
                                    dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                    len_ctx,          &
                                    user_ctx,         &
#endif
                                    num_exp,          &
                                    val_exp)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: xc_len_tuple
                integer(kind=QcPertInt), intent(in) :: xc_pert_tuple(xc_len_tuple)
                integer(kind=QINT), intent(in) :: num_freq_configs
                integer(kind=QINT), intent(in) :: dmat_num_tuple
                integer(kind=QINT), intent(in) :: dmat_idx_tuple(dmat_num_tuple)
                integer(kind=QINT), intent(in) :: num_dmat
                type(QcMat), intent(in) :: dens_mat(num_dmat)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: num_exp
                real(kind=QREAL), intent(inout) :: val_exp(num_exp)
            end subroutine get_xc_fun_exp
        end interface
#if defined(OPENRSP_F_USER_CONTEXT)
```

```
        integer(kind=4) ierr  !error information
        xcfun_fun%len_ctx = size(user_ctx)
        allocate(xcfun_fun%user_ctx(xcfun_fun%len_ctx), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPXCFunCreate_f>> length", xcfun_fun%len_ctx
            stop "RSPXCFunCreate_f>> failed to allocate memory for user_ctx"
        end if
        xcfun_fun%user_ctx = user_ctx
#endif
        xcfun_fun%get_xc_fun_mat => get_xc_fun_mat
        xcfun_fun%get_xc_fun_exp => get_xc_fun_exp
    end subroutine RSPXCFunCreate_f

    !% \brief calls Fortran callback subroutine to get integral matrices of
    !      an XC functional
    !  \author Bin Gao
    !  \date 2015-06-23
    !  \param[integer]{in} len_tuple length of perturbation tuple on the XC functional
    !  \param[integer]{in} pert_tuple perturbation tuple on the XC functional
    !  \param[integer]{in} num_freq_configs the number of different frequency
    !      configurations to be considered for the perturbation tuple
    !  \param[integer]{in} dmat_len_tuple the number of different perturbation
    !      tuples of the AO based density matrices passed
    !  \param[integer]{in} dmat_idx_tuple indices of the density matrix
    !      perturbation tuples passed (canonically ordered)
    !  \param[integer]{in} num_dmat number of collected AO based density matrices for
    !      the passed density matrix perturbation tuples and all frequency configurations
    !  \param[C_PTR:type]{in} dens_mat the collected AO based density matrices
    !  \param[C_PTR:type]{in} user_ctx user-defined callback function context
    !  \param[integer]{in} num_int number of the integral matrices
    !% \param[C_PTR:type]{inout} val_int the integral matrices
    subroutine RSPXCFunGetMat_f(xc_len_tuple,     &
                                xc_pert_tuple,    &
                                num_freq_configs, &
                                dmat_num_tuple,   &
                                dmat_idx_tuple,   &
                                num_dmat,         &
                                dens_mat,         &
                                user_ctx,         &
                                num_int,          &
                                val_int)          &
        bind(C, name="RSPXCFunGetMat_f")
        integer(kind=C_QINT), value, intent(in) :: xc_len_tuple
        integer(kind=C_QCPERTINT), intent(in) :: xc_pert_tuple(xc_len_tuple)
        integer(kind=C_QINT), value, intent(in) :: num_freq_configs
        integer(kind=C_QINT), value, intent(in) :: dmat_num_tuple
        integer(kind=C_QINT), intent(in) :: dmat_idx_tuple(dmat_num_tuple)
        integer(kind=C_QINT), value, intent(in) :: num_dmat
        type(C_PTR), intent(in) :: dens_mat(num_dmat)
        type(C_PTR), value, intent(in) :: user_ctx
        integer(kind=C_QINT), value, intent(in) :: num_int
        type(C_PTR), intent(inout) :: val_int(num_int)
```

```
        type(XCFunFun_f), pointer :: xcfun_fun     !context of callback subroutines
        type(QcMat), allocatable :: f_dens_mat(:)  !AO based density matrices
        type(QcMat), allocatable :: f_val_int(:)   !integral matrices
        integer(kind=4) ierr                       !error information
        ! converts C pointer to Fortran QcMat type
        allocate(f_dens_mat(num_dmat), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPXCFunGetMat_f>> num_dmat", num_dmat
            stop "RSPXCFunGetMat_f>> failed to allocate memory for f_dens_mat"
        end if
        ierr = QcMat_C_F_POINTER(A=f_dens_mat, c_A=dens_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        allocate(f_val_int(num_int), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPXCFunGetMat_f>> num_int", num_int
            stop "RSPXCFunGetMat_f>> failed to allocate memory for f_val_int"
        end if
        ierr = QcMat_C_F_POINTER(A=f_val_int, c_A=val_int)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        ! gets the Fortran callback subroutine
        call c_f_pointer(user_ctx, xcfun_fun)
        ! invokes Fortran callback subroutine to calculate the integral matrices
        call xcfun_fun%get_xc_fun_mat(xc_len_tuple,       &
                                      xc_pert_tuple,      &
                                      num_freq_configs,   &
                                      dmat_num_tuple,     &
                                      dmat_idx_tuple,     &
                                      num_dmat,           &
                                      f_dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                      xcfun_fun%len_ctx,  &
                                      xcfun_fun%user_ctx, &
#endif
                                      num_int,            &
                                      f_val_int)
        ! cleans up
        nullify(xcfun_fun)
        ierr = QcMat_C_NULL_PTR(A=f_val_int)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        ierr = QcMat_C_NULL_PTR(A=f_dens_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        deallocate(f_val_int)
        deallocate(f_dens_mat)
    end subroutine RSPXCFunGetMat_f

    !% \brief calls Fortran callback subroutine to get expectation values of
    !      an XC functional
    !  \author Bin Gao
    !  \date 2015-06-23
    !  \param[integer]{in} len_tuple length of perturbation tuple on the XC functional
    !  \param[integer]{in} pert_tuple perturbation tuple on the XC functional
    !  \param[integer]{in} num_freq_configs the number of different frequency
```

```
!      configurations to be considered for the perturbation tuple
!  \param[integer]{in} dmat_len_tuple the number of different perturbation
!      tuples of the AO based density matrices passed
!  \param[integer]{in} dmat_idx_tuple indices of the density matrix
!      perturbation tuples passed (canonically ordered)
!  \param[integer]{in} num_dmat number of collected AO based density matrices for
!      the passed density matrix perturbation tuples and all frequency configurations
!  \param[C_PTR:type]{in} dens_mat the collected AO based density matrices
!  \param[C_PTR:type]{in} user_ctx user-defined callback function context
!  \param[integer]{in} num_exp number of expectation values
!% \param[real]{out} val_exp the expectation values
subroutine RSPXCFunGetExp_f(xc_len_tuple,     &
                            xc_pert_tuple,    &
                            num_freq_configs, &
                            dmat_num_tuple,   &
                            dmat_idx_tuple,   &
                            num_dmat,         &
                            dens_mat,         &
                            user_ctx,         &
                            num_exp,          &
                            val_exp)          &
    bind(C, name="RSPXCFunGetExp_f")
    integer(kind=C_QINT), value, intent(in) :: xc_len_tuple
    integer(kind=C_QCPERTINT), intent(in) :: xc_pert_tuple(xc_len_tuple)
    integer(kind=C_QINT), value, intent(in) :: num_freq_configs
    integer(kind=C_QINT), value, intent(in) :: dmat_num_tuple
    integer(kind=C_QINT), intent(in) :: dmat_idx_tuple(dmat_num_tuple)
    integer(kind=C_QINT), value, intent(in) :: num_dmat
    type(C_PTR), intent(in) :: dens_mat(num_dmat)
    type(C_PTR), value, intent(in) :: user_ctx
    integer(kind=C_QINT), value, intent(in) :: num_exp
    real(kind=C_QREAL), intent(inout) :: val_exp(num_exp)
    type(XCFunFun_f), pointer :: xcfun_fun     !context of callback subroutines
    type(QcMat), allocatable :: f_dens_mat(:)  !AO based density matrices
    integer(kind=4) ierr                       !error information
    ! converts C pointer to Fortran QcMat type
    allocate(f_dens_mat(num_dmat), stat=ierr)
    if (ierr/=0) then
        write(STDOUT,"(A,I8)") "RSPXCFunGetExp_f>> num_dmat", num_dmat
        stop "RSPXCFunGetExp_f>> failed to allocate memory for f_dens_mat"
    end if
    ierr = QcMat_C_F_POINTER(A=f_dens_mat, c_A=dens_mat)
    call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
    ! gets the Fortran callback subroutine
    call c_f_pointer(user_ctx, xcfun_fun)
    ! invokes Fortran callback subroutine to calculate the expectation values
    call xcfun_fun%get_xc_fun_exp(xc_len_tuple,       &
                                  xc_pert_tuple,      &
                                  num_freq_configs,   &
                                  dmat_num_tuple,     &
                                  dmat_idx_tuple,     &
                                  num_dmat,           &
```

```
                                        f_dens_mat,         &
#if defined(OPENRSP_F_USER_CONTEXT)
                                        xcfun_fun%len_ctx,  &
                                        xcfun_fun%user_ctx, &
#endif
                                        num_exp,            &
                                        val_exp)
        ! cleans up
        nullify(xcfun_fun)
        ierr = QcMat_C_NULL_PTR(A=f_dens_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        deallocate(f_dens_mat)
        return
    end subroutine RSPXCFunGetExp_f

    !% \brief cleans the context of callback subroutines of XC functional
    !  \author Bin Gao
    !  \date 2015-06-23
    !% \param[XCFunFun_f:type]{inout} xcfun_fun the context of callback subroutines
    subroutine RSPXCFunDestroy_f(xcfun_fun)
        type(XCFunFun_f), intent(inout) :: xcfun_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        xcfun_fun%len_ctx = 0
        deallocate(xcfun_fun%user_ctx)
#endif
        nullify(xcfun_fun%get_xc_fun_mat)
        nullify(xcfun_fun%get_xc_fun_exp)
    end subroutine RSPXCFunDestroy_f

end module RSPXCFun_f

#undef OPENRSP_API_SRC
```

172 ⟨*RSPNucHamilton.F90* 172⟩≡
⟨*OpenRSPLicenseFortran* 116a⟩

```
!!  2015-06-23, Bin Gao
!!  * first version

! basic data types
#include "api/qcmatrix_c_type.h"

#define OPENRSP_API_SRC "src/fortran/RSPNucHamilton.F90"

module RSPNucHamilton_f

    use, intrinsic :: iso_c_binding
    use qcmatrix_f, only: QINT,QREAL
    use RSPPertBasicTypes_f, only: QcPertInt, &
                                   C_QCPERTINT

    implicit none
```

```
    integer(kind=4), private, parameter :: STDOUT = 6

    ! user specified callback subroutine
    abstract interface
        subroutine NucHamiltonGetContrib_f(nuc_num_pert,    &
                                           nuc_pert_labels, &
                                           nuc_pert_orders, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                           len_ctx,         &
                                           user_ctx,        &
#endif
                                           size_pert,       &
                                           val_nuc)
            use qcmatrix_f, only: QINT,QREAL
            use RSPPertBasicTypes_f, only: QcPertInt
            integer(kind=QINT), intent(in) :: nuc_num_pert
            integer(kind=QcPertInt), intent(in) :: nuc_pert_labels(nuc_num_pert)
            integer(kind=QINT), intent(in) :: nuc_pert_orders(nuc_num_pert)
#if defined(OPENRSP_F_USER_CONTEXT)
            integer(kind=QINT), intent(in) :: len_ctx
            character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
            integer(kind=QINT), intent(in) :: size_pert
            real(kind=QREAL), intent(inout) :: val_nuc(size_pert)
        end subroutine NucHamiltonGetContrib_f
    end interface

    ! context of callback subroutine of nuclear Hamiltonian
    type, public :: NucHamiltonFun_f
        private
#if defined(OPENRSP_F_USER_CONTEXT)
        ! user-defined callback function context
        integer(kind=QINT) :: len_ctx = 0
        character(len=1), allocatable :: user_ctx(:)
#endif
        ! callback function
        procedure(NucHamiltonGetContrib_f), nopass, pointer :: get_nuc_contrib
    end type NucHamiltonFun_f

    public :: RSPNucHamiltonCreate_f
    public :: RSPNucHamiltonGetContributions_f
    public :: RSPNucHamiltonDestroy_f

    contains

    !% \brief creates the context of callback subroutine of nuclear Hamiltonian
    !  \author Bin Gao
    !  \date 2015-06-23
    !  \param[NucHamiltonFun_f:type]{inout} nuc_hamilton_fun the context of callback subrou
    !  \param[character]{in} user_ctx user-defined callback function context
    !  \param[subroutine]{in} get_nuc_contrib user specified function for
```

```
    !%     getting nuclear contributions
    subroutine RSPNucHamiltonCreate_f(nuc_hamilton_fun, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                      user_ctx,         &
#endif
                                      get_nuc_contrib)
        type(NucHamiltonFun_f), intent(inout) :: nuc_hamilton_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        interface
            subroutine get_nuc_contrib(nuc_num_pert,    &
                                       nuc_pert_labels, &
                                       nuc_pert_orders, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                       len_ctx,         &
                                       user_ctx,        &
#endif
                                       size_pert,       &
                                       val_nuc)
                use qcmatrix_f, only: QINT,QREAL
                use RSPPertBasicTypes_f, only: QcPertInt
                integer(kind=QINT), intent(in) :: nuc_num_pert
                integer(kind=QcPertInt), intent(in) :: nuc_pert_labels(nuc_num_pert)
                integer(kind=QINT), intent(in) :: nuc_pert_orders(nuc_num_pert)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                integer(kind=QINT), intent(in) :: size_pert
                real(kind=QREAL), intent(inout) :: val_nuc(size_pert)
            end subroutine get_nuc_contrib
        end interface
#if defined(OPENRSP_F_USER_CONTEXT)
        integer(kind=4) ierr  !error information
        nuc_hamilton_fun%len_ctx = size(user_ctx)
        allocate(nuc_hamilton_fun%user_ctx(nuc_hamilton_fun%len_ctx), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPNucHamiltonCreate_f>> length", nuc_hamilton_fun%len_
            stop "RSPNucHamiltonCreate_f>> failed to allocate memory for user_ctx"
        end if
        nuc_hamilton_fun%user_ctx = user_ctx
#endif
        nuc_hamilton_fun%get_nuc_contrib => get_nuc_contrib
    end subroutine RSPNucHamiltonCreate_f

    !% \brief calls Fortran callback subroutine to get nuclear contributions
    !  \author Bin Gao
    !  \date 2015-06-23
    !  \param[integer]{in} len_tuple length of perturbation tuple on the nuclear Hamiltonia
    !  \param[integer]{in} pert_tuple perturbation tuple on the nuclear Hamiltonian
    !  \param[C_PTR:type]{in} user_ctx user-defined callback function context
```

```
    !  \param[integer]{in} size_pert size of the perturbations on the nuclear Hamiltonian
    !% \param[real]{out} val_nuc the nuclear contributions
    subroutine RSPNucHamiltonGetContributions_f(nuc_num_pert,    &
                                                nuc_pert_labels, &
                                                nuc_pert_orders, &
                                                user_ctx,        &
                                                size_pert,       &
                                                val_nuc)         &
        bind(C, name="RSPNucHamiltonGetContributions_f")
        integer(kind=C_QINT), value, intent(in) :: nuc_num_pert
        integer(kind=C_QCPERTINT), intent(in) :: nuc_pert_labels(nuc_num_pert)
        integer(kind=C_QINT), intent(in) :: nuc_pert_orders(nuc_num_pert)
        type(C_PTR), value, intent(in) :: user_ctx
        integer(kind=C_QINT), value, intent(in) :: size_pert
        real(C_QREAL), intent(inout) :: val_nuc(size_pert)
        type(NucHamiltonFun_f), pointer :: nuc_hamilton_fun  !context of callback subroutin
        ! gets the Fortran callback subroutine
        call c_f_pointer(user_ctx, nuc_hamilton_fun)
        ! invokes Fortran callback subroutine to calculate the nuclear contributions
        call nuc_hamilton_fun%get_nuc_contrib(nuc_num_pert,              &
                                              nuc_pert_labels,           &
                                              nuc_pert_orders,           &
#if defined(OPENRSP_F_USER_CONTEXT)
                                              nuc_hamilton_fun%len_ctx,  &
                                              nuc_hamilton_fun%user_ctx, &
#endif
                                              size_pert,                 &
                                              val_nuc)
        ! cleans up
        nullify(nuc_hamilton_fun)
        return
    end subroutine RSPNucHamiltonGetContributions_f

    !% \brief cleans the context of callback subroutine of nuclear Hamiltonian
    !  \author Bin Gao
    !  \date 2015-06-23
    !% \param[NucHamiltonFun_f:type]{inout} nuc_hamilton_fun the context of callback subrou
    subroutine RSPNucHamiltonDestroy_f(nuc_hamilton_fun)
        type(NucHamiltonFun_f), intent(inout) :: nuc_hamilton_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        nuc_hamilton_fun%len_ctx = 0
        deallocate(nuc_hamilton_fun%user_ctx)
#endif
        nullify(nuc_hamilton_fun%get_nuc_contrib)
    end subroutine RSPNucHamiltonDestroy_f

end module RSPNucHamilton_f

#undef OPENRSP_API_SRC
```

175 ⟨*RSPSolver.F90* 175⟩≡
⟨*OpenRSPLicenseFortran* 116a⟩

```
!!  2014-08-06, Bin Gao
!!  * first version

! basic data types
#include "api/qcmatrix_c_type.h"

#define OPENRSP_API_SRC "src/fortran/RSPSolver.F90"

module RSPSolver_f

    use, intrinsic :: iso_c_binding
    use qcmatrix_f, only: QINT,QREAL,QcMat,QcMat_C_F_POINTER,QcMat_C_NULL_PTR

    implicit none

    integer(kind=4), private, parameter :: STDOUT = 6

    ! user specified callback subroutine
    abstract interface
        subroutine SolverRun_f(num_freq_sums, &
                               freq_sums,     &
                               size_pert,     &
                               RHS_mat,       &
#if defined(OPENRSP_F_USER_CONTEXT)
                               len_ctx,       &
                               user_ctx,      &
#endif
                               rsp_param)
            use qcmatrix_f, only: QINT,QREAL,QcMat
            integer(kind=QINT), intent(in) :: num_freq_sums
            real(kind=QREAL), intent(in) :: freq_sums(2*num_freq_sums)
            integer(kind=QINT), intent(in) :: size_pert
            type(QcMat), intent(in) :: RHS_mat(num_freq_sums*size_pert)
#if defined(OPENRSP_F_USER_CONTEXT)
            integer(kind=QINT), intent(in) :: len_ctx
            character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
            type(QcMat), intent(inout) :: rsp_param(num_freq_sums*size_pert)
        end subroutine SolverRun_f
    end interface

    ! context of callback subroutine of response equation solver
    type, public :: SolverFun_f
        private
#if defined(OPENRSP_F_USER_CONTEXT)
        ! user-defined callback function context
        integer(kind=QINT) :: len_ctx = 0
        character(len=1), allocatable :: user_ctx(:)
#endif
        ! callback function
        procedure(SolverRun_f), nopass, pointer :: get_linear_rsp_solution
```

```
    end type SolverFun_f

    public :: RSPSolverCreate_f
    public :: RSPSolverGetLinearRSPSolution_f
    public :: RSPSolverDestroy_f

    contains

    !% \brief creates the context of callback subroutine of response equation solver
    !  \author Bin Gao
    !  \date 2014-08-06
    !  \param[SolverFun_f:type]{inout} solver_fun the context of callback subroutine
    !  \param[character]{in} user_ctx user-defined callback function context
    !  \param[subroutine]{in} get_linear_rsp_solution user specified function of
    !%     response equation solver
    subroutine RSPSolverCreate_f(solver_fun, &
#if defined(OPENRSP_F_USER_CONTEXT)
                                 user_ctx,   &
#endif
                                 get_linear_rsp_solution)
        type(SolverFun_f), intent(inout) :: solver_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        character(len=1), intent(in) :: user_ctx(:)
#endif
        interface
            subroutine get_linear_rsp_solution(num_freq_sums, &
                                               freq_sums,     &
                                               size_pert,     &
                                               RHS_mat,       &
#if defined(OPENRSP_F_USER_CONTEXT)
                                               len_ctx,       &
                                               user_ctx,      &
#endif
                                               rsp_param)
                use qcmatrix_f, only: QINT,QREAL,QcMat
                integer(kind=QINT), intent(in) :: num_freq_sums
                real(kind=QREAL), intent(in) :: freq_sums(2*num_freq_sums)
                integer(kind=QINT), intent(in) :: size_pert
                type(QcMat), intent(in) :: RHS_mat(num_freq_sums*size_pert)
#if defined(OPENRSP_F_USER_CONTEXT)
                integer(kind=QINT), intent(in) :: len_ctx
                character(len=1), intent(in) :: user_ctx(len_ctx)
#endif
                type(QcMat), intent(inout) :: rsp_param(num_freq_sums*size_pert)
            end subroutine get_linear_rsp_solution
        end interface
#if defined(OPENRSP_F_USER_CONTEXT)
        integer(kind=4) ierr  !error information
        solver_fun%len_ctx = size(user_ctx)
        allocate(solver_fun%user_ctx(solver_fun%len_ctx), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPSolverCreate_f>> length", solver_fun%len_ctx
```

```
            stop "RSPSolverCreate_f>> failed to allocate memory for user_ctx"
        end if
        solver_fun%user_ctx = user_ctx
#endif
        solver_fun%get_linear_rsp_solution => get_linear_rsp_solution
    end subroutine RSPSolverCreate_f

    !% \brief calls Fortran callback subroutine to get solution of response equation
    !  \author Bin Gao
    !  \date 2014-08-06
    !  \param[integer]{in} num_freq_sums number of complex frequency sums
    !      on the left hand side of the linear response equation
    !  \param[real]{in} freq_sums the complex frequency sums on the left hand side
    !  \param[integer]{in} size_pert size of perturbations acting on the
    !      time-dependent self-consistent-field (TDSCF) equation
    !  \param[C_PTR:type]{in} RHS_mat RHS matrices, size is \var{num_freq_sums}*\var{size_p
    !  \param[C_PTR:type]{in} user_ctx user-defined callback function context
    !  \param[C_PTR:type]{out} rsp_param solved response parameters,
    !%     size is \var{num_freq_sums}*\var{size_pert}
    subroutine RSPSolverGetLinearRSPSolution_f(num_freq_sums, &
                                               freq_sums,     &
                                               size_pert,     &
                                               RHS_mat,       &
                                               user_ctx,      &
                                               rsp_param)     &
        bind(C, name="RSPSolverGetLinearRSPSolution_f")
        integer(kind=C_QINT), value, intent(in) :: num_freq_sums
        real(kind=C_QREAL), intent(in) :: freq_sums(2*num_freq_sums)
        integer(kind=C_QINT), value, intent(in) :: size_pert
        type(C_PTR), intent(in) :: RHS_mat(num_freq_sums*size_pert)
        type(C_PTR), value, intent(in) :: user_ctx
        type(C_PTR), intent(inout) :: rsp_param(num_freq_sums*size_pert)
        type(SolverFun_f), pointer :: solver_fun    !context of callback subroutine
        integer(kind=QINT) size_solution            !size of solution of response equation
        type(QcMat), allocatable :: f_RHS_mat(:)    !RHS matrices
        type(QcMat), allocatable :: f_rsp_param(:)  !response parameters
        integer(kind=4) ierr                        !error information
        ! converts C pointer to Fortran QcMat type
        size_solution = num_freq_sums*size_pert
        allocate(f_RHS_mat(size_solution), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPSolverGetLinearRSPSolution_f>> size_solution", &
                                   size_solution
            stop "RSPSolverGetLinearRSPSolution_f>> failed to allocate memory for f_RHS_mat
        end if
        ierr = QcMat_C_F_POINTER(A=f_RHS_mat, c_A=RHS_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        allocate(f_rsp_param(size_solution), stat=ierr)
        if (ierr/=0) then
            write(STDOUT,"(A,I8)") "RSPSolverGetLinearRSPSolution_f>> size_solution", &
                                   size_solution
            stop "RSPSolverGetLinearRSPSolution_f>> failed to allocate memory for f_rsp_par
```

```
        end if
        ierr = QcMat_C_F_POINTER(A=f_rsp_param, c_A=rsp_param)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        ! gets the Fortran callback subroutine
        call c_f_pointer(user_ctx, solver_fun)
        ! invokes Fortran callback subroutine to solve the response equation
        call solver_fun%get_linear_rsp_solution(num_freq_sums,       &
                                                freq_sums,           &
                                                size_pert,           &
                                                f_RHS_mat,           &
#if defined(OPENRSP_F_USER_CONTEXT)
                                                solver_fun%len_ctx,  &
                                                solver_fun%user_ctx, &
#endif
                                                f_rsp_param)
        ! cleans up
        nullify(solver_fun)
        ierr = QcMat_C_NULL_PTR(A=f_rsp_param)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        ierr = QcMat_C_NULL_PTR(A=f_RHS_mat)
        call QErrorCheckCode(STDOUT, ierr, __LINE__, OPENRSP_API_SRC)
        deallocate(f_rsp_param)
        deallocate(f_RHS_mat)
        return
    end subroutine RSPSolverGetLinearRSPSolution_f

    !% \brief cleans the context of callback subroutine of response equation solver
    !  \author Bin Gao
    !  \date 2014-08-06
    !% \param[SolverFun_f:type]{inout} solver_fun the context of callback subroutine
    subroutine RSPSolverDestroy_f(solver_fun)
        type(SolverFun_f), intent(inout) :: solver_fun
#if defined(OPENRSP_F_USER_CONTEXT)
        solver_fun%len_ctx = 0
        deallocate(solver_fun%user_ctx)
#endif
        nullify(solver_fun%get_linear_rsp_solution)
    end subroutine RSPSolverDestroy_f

end module RSPSolver_f

#undef OPENRSP_API_SRC
```

Furthermoe, we implement the following Fortran adapters:

179 ⟨*OpenRSPFortranAdapter.c* 179⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

   2014-07-31, Bin Gao
   * first version
*/

#include "OpenRSP.h"
```

```
QErrorCode OpenRSPCreateFortranAdapter(void **open_rsp)
{
    OpenRSP *c_open_rsp;
    QErrorCode ierr;
    c_open_rsp = (OpenRSP *)malloc(sizeof(OpenRSP));
    if (c_open_rsp==NULL) {
        QErrorExit(FILE_AND_LINE, "failed to allocate memory for c_open_rsp");
    }
    ierr = OpenRSPCreate(c_open_rsp);
    *open_rsp = (void *)(c_open_rsp);
    return ierr;
}

//QErrorCode f_api_OpenRSPSetElecEOM(void **open_rsp,
//                                   const QInt elec_EOM_type)
//{
//    OpenRSP *c_open_rsp;
//    ElecEOMType c_elec_EOM_type;
//    QErrorCode ierr;
//    /* should be consistent with what defined in src/f03/openrsp_f.F90 */
//    switch (elec_EOM_type) {
//    case 0:
//        c_elec_EOM_type = ELEC_AO_D_MATRIX;
//        break;
//    case 1:
//        c_elec_EOM_type = ELEC_MO_C_MATRIX;
//        break;
//    case 2:
//        c_elec_EOM_type = ELEC_COUPLED_CLUSTER;
//        break;
//    default:
//        return QFAILURE;
//    }
//    c_open_rsp = (OpenRSP *)(*open_rsp);
//    ierr = OpenRSPSetElecEOM(c_open_rsp, c_elec_EOM_type);
//    return ierr;
//}

QErrorCode OpenRSPDestroyFortranAdapter(void **open_rsp)
{
    OpenRSP *c_open_rsp;
    QErrorCode ierr;
    c_open_rsp = (OpenRSP *)(*open_rsp);
    ierr = OpenRSPDestroy(c_open_rsp);
    *open_rsp = NULL;
    open_rsp = NULL;
    return ierr;
}
```

3.13 Unit Testing

3.13.1 Testing C APIs

We first implement perturbations and its callback function for unit testing:

181a ⟨*OpenRSPTestPerturbations.h* 181a⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

  This is the header file of perturbations.

  2014-07-31, Bin Gao:
  * first version
*/

#include "OpenRSP.h"

#define NUM_ALL_PERT 3
extern const QcPertInt PERT_GEOMETRIC;
extern const QcPertInt PERT_DIPOLE;
extern const QcPertInt PERT_MAGNETIC;
extern const QInt MAX_ORDER_GEOMETRIC;
extern const QInt MAX_ORDER_DIPOLE;
extern const QInt MAX_ORDER_MAGNETIC;
```

181b ⟨*OpenRSPTestPerturbations.c* 181b⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩
*/

#include "OpenRSPTestPerturbations.h"

const QcPertInt PERT_GEOMETRIC = 1;
const QcPertInt PERT_DIPOLE = 2;
const QcPertInt PERT_MAGNETIC = 5;
const QInt MAX_ORDER_GEOMETRIC = 7;
const QInt MAX_ORDER_DIPOLE = 1;
const QInt MAX_ORDER_MAGNETIC = 7;
```

Here we have used three different perturbation labels for geometric, dipole length and magnetic perturbations.

181c ⟨*OpenRSPPertCallback.h* 181c⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

  This is the header file of perturbations' callback function.

  2014-07-31, Bin Gao:
  * first version
*/

#if !defined(OPENRSP_PERT_CALLBACK_H)
#define OPENRSP_PERT_CALLBACK_H
```

```
#include "OpenRSP.h"

extern void get_pert_concatenation(const QInt,
                                   const QcPertInt,
                                   const QInt,
                                   const QInt,
                                   const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                                   void*,
#endif
                                   QInt*);

#endif
```

182a ⟨*OpenRSPPertCallback.c* 182a⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩
*/

#include "OpenRSPPertCallback.h"

void get_pert_concatenation(const QInt pert_label,
                            const QcPertInt first_cat_comp,
                            const QInt num_cat_comps,
                            const QInt num_sub_tuples,
                            const QInt *len_sub_tuples,
#if defined(OPENRSP_C_USER_CONTEXT)
                            void *user_ctx,
#endif
                            QInt *rank_sub_comps)
{
}
```

We next implement callback functions for AO-based response theory calculations:

182b ⟨*OpenRSPAODensCallback.h* 182b⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

  This header file contains callback functions for AO-based response theory
  calculations.

  2015-10-16, Bin Gao:
  * first version
*/

#if !defined(OPENRSP_AO_DENS_CALLBACK_H)
#define OPENRSP_AO_DENS_CALLBACK_H

#include "OpenRSP.h"

#if defined(ZERO_BASED_NUMBERING)
#define IDX_BLOCK_ROW 0
#define IDX_BLOCK_COL 0
#define IDX_FIRST_ROW 0
```

```
#define IDX_FIRST_COL 0
#else
#define IDX_BLOCK_ROW 1
#define IDX_BLOCK_COL 1
#define IDX_FIRST_ROW 1
#define IDX_FIRST_COL 1
#endif

extern void get_overlap_mat(const QInt,
                            const QcPertInt*,
                            const QInt*,
                            const QInt,
                            const QcPertInt*,
                            const QInt*,
                            const QInt,
                            const QcPertInt*,
                            const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                            void*,
#endif
                            const QInt,
                            QcMat*[]);
extern void get_overlap_exp(const QInt,
                            const QcPertInt*,
                            const QInt*,
                            const QInt,
                            const QcPertInt*,
                            const QInt*,
                            const QInt,
                            const QcPertInt*,
                            const QInt*,
                            const QInt,
                            QcMat*[],
#if defined(OPENRSP_C_USER_CONTEXT)
                            void*,
#endif
                            const QInt,
                            QReal*);
extern void get_one_oper_mat(const QInt,
                             const QcPertInt*,
                             const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                             void*,
#endif
                             const QInt,
                             QcMat*[]);
extern void get_one_oper_exp(const QInt,
                             const QcPertInt*,
                             const QInt*,
                             const QInt,
                             QcMat*[],
#if defined(OPENRSP_C_USER_CONTEXT)
```

```
                                void*,
#endif
                                const QInt,
                                QReal*);
extern void get_two_oper_mat(const QInt,
                                const QcPertInt*,
                                const QInt*,
                                const QInt,
                                QcMat*[],
#if defined(OPENRSP_C_USER_CONTEXT)
                                void*,
#endif
                                const QInt,
                                QcMat*[]);
extern void get_two_oper_exp(const QInt,
                                const QcPertInt*,
                                const QInt*,
                                const QInt,
                                const QInt*,
                                QcMat*[],
                                const QInt*,
                                QcMat*[],
#if defined(OPENRSP_C_USER_CONTEXT)
                                void*,
#endif
                                const QInt,
                                QReal*);
extern void get_nuc_contrib(const QInt,
                               const QcPertInt*,
                               const QInt*,
#if defined(OPENRSP_C_USER_CONTEXT)
                               void*,
#endif
                               const QInt,
                               QReal*);
extern void get_linear_rsp_solution(const QInt,
                                       const QReal*,
                                       const QInt,
                                       QcMat*[],
#if defined(OPENRSP_C_USER_CONTEXT)
                                       void*,
#endif
                                       QcMat*[]);

#endif
```

Here we mimic callback functions by saving different AO-based matrices calculated from Dalton program. These matrices are save in the directory `tests/ao_dens_ground_state_hf`, `tests/ao_dens_alpha_hf`.

184 ⟨*OpenRSPAODensCallback.c* 184⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩
*/
```

```
#include "OpenRSPAODensCallback.h"

void get_overlap_mat(const QInt bra_num_pert,
                     const QcPertInt *bra_pert_labels,
                     const QInt *bra_pert_orders,
                     const QInt ket_num_pert,
                     const QcPertInt *ket_pert_labels,
                     const QInt *ket_pert_orders,
                     const QInt oper_num_pert,
                     const QcPertInt *oper_pert_labels,
                     const QInt *oper_pert_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                     void *user_ctx,
#endif
                     const QInt num_int,
                     QcMat *val_int[])
{
#include "OpenRSPTestPerturbations.h"
#include "ao_dens_ground_state_hf/num_atomic_orbitals.h"
#include "ao_dens_ground_state_hf/overlap_integrals.h"
#if defined(OPENRSP_C_USER_CONTEXT)
    char *overlap_context;
#endif
    QInt idx_block_row[1] = {IDX_BLOCK_ROW};
    QInt idx_block_col[1] = {IDX_BLOCK_COL};
    QcDataType data_type[1] = {QREALMAT};
    QBool assembled;
    QInt imat;
    QErrorCode ierr;
#if defined(OPENRSP_C_USER_CONTEXT)
    overlap_context = (char *)user_ctx;
    if (strcmp(overlap_context, "OVERLAP")!=0) {
        printf("get_overlap_mat>> incorrect operator at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
#endif
    /* overlap integrals */
    if (oper_num_pert==0 && bra_num_pert==0 && ket_num_pert==0) {
        /* checks if the matrix is assembled or not */
        ierr = QcMatIsAssembled(val_int[0], &assembled);
        if (ierr!=QSUCCESS) {
            printf("get_overlap_mat>> error happened at %s: %s\n",
                   FILE_AND_LINE,
                   "calling QcMatIsAssembled");
            exit(ierr);
        }
        if (assembled==QFALSE) {
            ierr = QcMatBlockCreate(val_int[0], 1);
            if (ierr!=QSUCCESS) {
                printf("get_overlap_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
```

```
                    "calling QcMatBlockCreate");
                exit(ierr);
            }
            ierr = QcMatSetSymType(val_int[0], QSYMMAT);
            if (ierr!=QSUCCESS) {
                printf("get_overlap_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatSetSymType");
                exit(ierr);
            }
            ierr = QcMatSetDataType(val_int[0],
                                    1,
                                    idx_block_row,
                                    idx_block_col,
                                    data_type);
            if (ierr!=QSUCCESS) {
                printf("get_overlap_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatSetDataType");
                exit(ierr);
            }
            ierr = QcMatSetDimMat(val_int[0], NUM_AO, NUM_AO);
            if (ierr!=QSUCCESS) {
                printf("get_overlap_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatSetDimMat");
                exit(ierr);
            }
            ierr = QcMatAssemble(val_int[0]);
            if (ierr!=QSUCCESS) {
                printf("get_overlap_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatAssemble");
                exit(ierr);
            }
        }
        ierr = QcMatSetValues(val_int[0],
                              IDX_BLOCK_ROW,
                              IDX_BLOCK_COL,
                              IDX_FIRST_ROW,
                              NUM_AO,
                              IDX_FIRST_COL,
                              NUM_AO,
                              values_overlap,
                              NULL);
        if (ierr!=QSUCCESS) {
            printf("get_overlap_mat>> error happened at %s: %s\n",
                   FILE_AND_LINE,
                   "calling QcMatSetValues");
            exit(ierr);
        }
    }
```

```
    else if (oper_num_pert==1 && bra_num_pert==0 && ket_num_pert==0) {
        if (oper_pert_labels[0]==PERT_GEOMETRIC) {
            printf("get_overlap_mat>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
        /* zero integrals */
        else if (oper_pert_labels[0]==PERT_DIPOLE) {
            for (imat=0; imat<num_int; imat++) {
                /* checks if the matrix is assembled or not */
                ierr = QcMatIsAssembled(val_int[imat], &assembled);
                if (ierr!=QSUCCESS) {
                    printf("get_overlap_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatIsAssembled");
                    exit(ierr);
                }
                if (assembled==QFALSE) {
                    ierr = QcMatBlockCreate(val_int[imat], 1);
                    if (ierr!=QSUCCESS) {
                        printf("get_overlap_mat>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatBlockCreate");
                        exit(ierr);
                    }
                    ierr = QcMatSetSymType(val_int[imat], QSYMMAT);
                    if (ierr!=QSUCCESS) {
                        printf("get_overlap_mat>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatSetSymType");
                        exit(ierr);
                    }
                    ierr = QcMatSetDataType(val_int[imat],
                                            1,
                                            idx_block_row,
                                            idx_block_col,
                                            data_type);
                    if (ierr!=QSUCCESS) {
                        printf("get_overlap_mat>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatSetDataType");
                        exit(ierr);
                    }
                    ierr = QcMatSetDimMat(val_int[imat], NUM_AO, NUM_AO);
                    if (ierr!=QSUCCESS) {
                        printf("get_overlap_mat>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatSetDimMat");
                        exit(ierr);
                    }
                    ierr = QcMatAssemble(val_int[imat]);
                    if (ierr!=QSUCCESS) {
```

```
                    printf("get_overlap_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatAssemble");
                    exit(ierr);
                }
            }
            ierr = QcMatZeroEntries(val_int[imat]);
            if (ierr!=QSUCCESS) {
                printf("get_overlap_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatZeroEntries");
                exit(ierr);
            }
        }
    }
    else if (oper_pert_labels[0]==PERT_MAGNETIC) {
        printf("get_overlap_mat>> not implemented at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
    else {
        printf("get_overlap_mat>> unknown perturbations at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
}
else if (oper_num_pert==0 && bra_num_pert==1 && ket_num_pert==0) {
    if (bra_pert_labels[0]==PERT_GEOMETRIC) {
        printf("get_overlap_mat>> not implemented at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
    /* zero integrals */
    else if (bra_pert_labels[0]==PERT_DIPOLE) {
        for (imat=0; imat<num_int; imat++) {
            /* checks if the matrix is assembled or not */
            ierr = QcMatIsAssembled(val_int[imat], &assembled);
            if (ierr!=QSUCCESS) {
                printf("get_overlap_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatIsAssembled");
                exit(ierr);
            }
            if (assembled==QFALSE) {
                ierr = QcMatBlockCreate(val_int[imat], 1);
                if (ierr!=QSUCCESS) {
                    printf("get_overlap_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatBlockCreate");
                    exit(ierr);
                }
                ierr = QcMatSetSymType(val_int[imat], QSYMMAT);
```

```
                if (ierr!=QSUCCESS) {
                    printf("get_overlap_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetSymType");
                    exit(ierr);
                }
                ierr = QcMatSetDataType(val_int[imat],
                                        1,
                                        idx_block_row,
                                        idx_block_col,
                                        data_type);
                if (ierr!=QSUCCESS) {
                    printf("get_overlap_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetDataType");
                    exit(ierr);
                }
                ierr = QcMatSetDimMat(val_int[imat], NUM_AO, NUM_AO);
                if (ierr!=QSUCCESS) {
                    printf("get_overlap_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetDimMat");
                    exit(ierr);
                }
                ierr = QcMatAssemble(val_int[imat]);
                if (ierr!=QSUCCESS) {
                    printf("get_overlap_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatAssemble");
                    exit(ierr);
                }
            }
            ierr = QcMatZeroEntries(val_int[imat]);
            if (ierr!=QSUCCESS) {
                printf("get_overlap_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatZeroEntries");
                exit(ierr);
            }
        }
    }
    else if (bra_pert_labels[0]==PERT_MAGNETIC) {
        printf("get_overlap_mat>> not implemented at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
    else {
        printf("get_overlap_mat>> unknown perturbations at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
}
```

```
else if (oper_num_pert==0 && bra_num_pert==0 && ket_num_pert==1) {
    if (ket_pert_labels[0]==PERT_GEOMETRIC) {
        printf("get_overlap_mat>> not implemented at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
    /* zero integrals */
    else if (ket_pert_labels[0]==PERT_DIPOLE) {
        for (imat=0; imat<num_int; imat++) {
            /* checks if the matrix is assembled or not */
            ierr = QcMatIsAssembled(val_int[imat], &assembled);
            if (ierr!=QSUCCESS) {
                printf("get_overlap_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatIsAssembled");
                exit(ierr);
            }
            if (assembled==QFALSE) {
                ierr = QcMatBlockCreate(val_int[imat], 1);
                if (ierr!=QSUCCESS) {
                    printf("get_overlap_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatBlockCreate");
                    exit(ierr);
                }
                ierr = QcMatSetSymType(val_int[imat], QSYMMAT);
                if (ierr!=QSUCCESS) {
                    printf("get_overlap_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetSymType");
                    exit(ierr);
                }
                ierr = QcMatSetDataType(val_int[imat],
                                        1,
                                        idx_block_row,
                                        idx_block_col,
                                        data_type);
                if (ierr!=QSUCCESS) {
                    printf("get_overlap_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetDataType");
                    exit(ierr);
                }
                ierr = QcMatSetDimMat(val_int[imat], NUM_AO, NUM_AO);
                if (ierr!=QSUCCESS) {
                    printf("get_overlap_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetDimMat");
                    exit(ierr);
                }
                ierr = QcMatAssemble(val_int[imat]);
                if (ierr!=QSUCCESS) {
```

```
                        printf("get_overlap_mat>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatAssemble");
                        exit(ierr);
                    }
                }
                ierr = QcMatZeroEntries(val_int[imat]);
                if (ierr!=QSUCCESS) {
                    printf("get_overlap_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatZeroEntries");
                    exit(ierr);
                }
            }
        }
        else if (ket_pert_labels[0]==PERT_MAGNETIC) {
            printf("get_overlap_mat>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
        else {
            printf("get_overlap_mat>> unknown perturbations at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
    }
    else {
        printf("get_overlap_mat>> not implemented at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
}

void get_overlap_exp(const QInt bra_num_pert,
                     const QcPertInt *bra_pert_labels,
                     const QInt *bra_pert_orders,
                     const QInt ket_num_pert,
                     const QcPertInt *ket_pert_labels,
                     const QInt *ket_pert_orders,
                     const QInt oper_num_pert,
                     const QcPertInt *oper_pert_labels,
                     const QInt *oper_pert_orders,
                     const QInt num_dens,
                     QcMat *ao_dens[],
#if defined(OPENRSP_C_USER_CONTEXT)
                     void *user_ctx,
#endif
                     const QInt num_exp,
                     QReal *val_exp)
{
#include "OpenRSPTestPerturbations.h"
#include "ao_dens_ground_state_hf/num_atomic_orbitals.h"
```

```
#include "ao_dens_ground_state_hf/overlap_integrals.h"
#if defined(OPENRSP_C_USER_CONTEXT)
    char *overlap_context;
#endif
    QInt idx_block_row[1] = {IDX_BLOCK_ROW};
    QInt idx_block_col[1] = {IDX_BLOCK_COL};
    QcDataType data_type[1] = {QREALMAT};
    QcMat val_int[1];
    QInt idens;
    QErrorCode ierr;
#if defined(OPENRSP_C_USER_CONTEXT)
    overlap_context = (char *)user_ctx;
    if (strcmp(overlap_context, "OVERLAP")!=0) {
        printf("get_overlap_mat>> incorrect operator at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
#endif
    /* overlap integrals */
    if (oper_num_pert==0 && bra_num_pert==0 && ket_num_pert==0) {
        ierr = QcMatCreate(&val_int[0]);
        if (ierr!=QSUCCESS) {
            printf("get_overlap_exp>> error happened at %s: %s\n",
                   FILE_AND_LINE,
                   "calling QcMatCreate");
            exit(ierr);
        }
        ierr = QcMatBlockCreate(&val_int[0], 1);
        if (ierr!=QSUCCESS) {
            printf("get_overlap_exp>> error happened at %s: %s\n",
                   FILE_AND_LINE,
                   "calling QcMatBlockCreate");
            exit(ierr);
        }
        ierr = QcMatSetSymType(&val_int[0], QSYMMAT);
        if (ierr!=QSUCCESS) {
            printf("get_overlap_exp>> error happened at %s: %s\n",
                   FILE_AND_LINE,
                   "calling QcMatSetSymType");
            exit(ierr);
        }
        ierr = QcMatSetDataType(&val_int[0],
                                1,
                                idx_block_row,
                                idx_block_col,
                                data_type);
        if (ierr!=QSUCCESS) {
            printf("get_overlap_exp>> error happened at %s: %s\n",
                   FILE_AND_LINE,
                   "calling QcMatSetDataType");
            exit(ierr);
        }
```

```
    ierr = QcMatSetDimMat(&val_int[0], NUM_AO, NUM_AO);
    if (ierr!=QSUCCESS) {
        printf("get_overlap_exp>> error happened at %s: %s\n",
               FILE_AND_LINE,
               "calling QcMatSetDimMat");
        exit(ierr);
    }
    ierr = QcMatAssemble(&val_int[0]);
    if (ierr!=QSUCCESS) {
        printf("get_overlap_exp>> error happened at %s: %s\n",
               FILE_AND_LINE,
               "calling QcMatAssemble");
        exit(ierr);
    }
    ierr = QcMatSetValues(&val_int[0],
                          IDX_BLOCK_ROW,
                          IDX_BLOCK_COL,
                          IDX_FIRST_ROW,
                          NUM_AO,
                          IDX_FIRST_COL,
                          NUM_AO,
                          values_overlap,
                          NULL);
    if (ierr!=QSUCCESS) {
        printf("get_overlap_exp>> error happened at %s: %s\n",
               FILE_AND_LINE,
               "calling QcMatSetValues");
        exit(ierr);
    }
    for (idens=0; idens<num_dens; idens++) {
        ierr = QcMatGetMatProdTrace(&val_int[0],
                                    ao_dens[idens],
                                    MAT_NO_OPERATION,
                                    1,
                                    &val_exp[2*idens]);
        if (ierr!=QSUCCESS) {
            printf("get_overlap_exp>> error happened at %s: %s\n",
                   FILE_AND_LINE,
                   "calling QcMatGetMatProdTrace");
            exit(ierr);
        }
    }
    ierr = QcMatDestroy(&val_int[0]);
    if (ierr!=QSUCCESS) {
        printf("get_overlap_exp>> error happened at %s: %s\n",
               FILE_AND_LINE,
               "calling QcMatDestroy");
        exit(ierr);
    }
}
else if (oper_num_pert==1 && bra_num_pert==0 && ket_num_pert==0) {
    if (oper_pert_labels[0]==PERT_GEOMETRIC) {
```

```
            printf("get_overlap_exp>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
        /* zero integrals */
        else if (oper_pert_labels[0]==PERT_DIPOLE) {
            for (idens=0; idens<2*num_exp; idens++) {
                val_exp[idens] = 0;
            }
        }
        else if (oper_pert_labels[0]==PERT_MAGNETIC) {
            printf("get_overlap_exp>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
        else {
            printf("get_overlap_exp>> unknown perturbations at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
    }
    else if (oper_num_pert==0 && bra_num_pert==1 && ket_num_pert==0) {
        if (bra_pert_labels[0]==PERT_GEOMETRIC) {
            printf("get_overlap_exp>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
        /* zero integrals */
        else if (bra_pert_labels[0]==PERT_DIPOLE) {
            for (idens=0; idens<2*num_exp; idens++) {
                val_exp[idens] = 0;
            }
        }
        else if (bra_pert_labels[0]==PERT_MAGNETIC) {
            printf("get_overlap_exp>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
        else {
            printf("get_overlap_exp>> unknown perturbations at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
    }
    else if (oper_num_pert==0 && bra_num_pert==0 && ket_num_pert==1) {
        if (ket_pert_labels[0]==PERT_GEOMETRIC) {
            printf("get_overlap_exp>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
        /* zero integrals */
        else if (ket_pert_labels[0]==PERT_DIPOLE) {
```

```
            for (idens=0; idens<2*num_exp; idens++) {
                val_exp[idens] = 0;
            }
        }
        else if (ket_pert_labels[0]==PERT_MAGNETIC) {
            printf("get_overlap_exp>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
        else {
            printf("get_overlap_exp>> unknown perturbations at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
    }
    else if (oper_num_pert==0 && bra_num_pert==1 && ket_num_pert==1) {
        /* zero integrals */
        if (bra_pert_labels[0]==PERT_DIPOLE || ket_pert_labels[0]==PERT_DIPOLE) {
            for (idens=0; idens<2*num_exp; idens++) {
                val_exp[idens] = 0;
            }
        }
        else {
            printf("get_overlap_exp>> unknown perturbations at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
    }
    else {
         printf("get_overlap_exp>> not implemented at %s\n",
                FILE_AND_LINE);
         exit(QFAILURE);
    }
}

void get_one_oper_mat(const QInt oper_num_pert,
                      const QcPertInt *oper_pert_labels,
                      const QInt *oper_pert_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                      void *user_ctx,
#endif
                      const QInt num_int,
                      QcMat *val_int[])
{
#include "OpenRSPTestPerturbations.h"
#include "ao_dens_ground_state_hf/num_atomic_orbitals.h"
#include "ao_dens_alpha_hf/dipole_length_integrals.h"
#if defined(OPENRSP_C_USER_CONTEXT)
    char *oneham_context = "ONEHAM";
    char *ext_field_context = "EXT_FIELD";
    char *one_oper_context;
#endif
```

```
    QInt idx_block_row[1] = {IDX_BLOCK_ROW};
    QInt idx_block_col[1] = {IDX_BLOCK_COL};
    QcDataType data_type[1] = {QREALMAT};
    QBool assembled;
    QInt imat;
    QErrorCode ierr;
#if defined(OPENRSP_C_USER_CONTEXT)
    one_oper_context = (char *)user_ctx;
    if (strcmp(one_oper_context, oneham_context)==0) {
        /* electric fields (zero integrals) */
        if (oper_num_pert==1 && oper_pert_labels[0]==PERT_DIPOLE) {
            for (imat=0; imat<num_int; imat++) {
                /* checks if the matrix is assembled or not */
                ierr = QcMatIsAssembled(val_int[imat], &assembled);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatIsAssembled");
                    exit(ierr);
                }
                if (assembled==QFALSE) {
                    ierr = QcMatBlockCreate(val_int[imat], 1);
                    if (ierr!=QSUCCESS) {
                        printf("get_one_oper_mat>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatBlockCreate");
                        exit(ierr);
                    }
                    ierr = QcMatSetSymType(val_int[imat], QSYMMAT);
                    if (ierr!=QSUCCESS) {
                        printf("get_one_oper_mat>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatSetSymType");
                        exit(ierr);
                    }
                    ierr = QcMatSetDataType(val_int[imat],
                                            1,
                                            idx_block_row,
                                            idx_block_col,
                                            data_type);
                    if (ierr!=QSUCCESS) {
                        printf("get_one_oper_mat>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatSetDataType");
                        exit(ierr);
                    }
                    ierr = QcMatSetDimMat(val_int[imat], NUM_AO, NUM_AO);
                    if (ierr!=QSUCCESS) {
                        printf("get_one_oper_mat>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatSetDimMat");
                        exit(ierr);
```

```
                    }
                    ierr = QcMatAssemble(val_int[imat]);
                    if (ierr!=QSUCCESS) {
                        printf("get_one_oper_mat>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatAssemble");
                        exit(ierr);
                    }
                }
                ierr = QcMatZeroEntries(val_int[imat]);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatZeroEntries");
                    exit(ierr);
                }
            }
        }
        else {
            printf("get_one_oper_mat>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
    }
    else if (strcmp(one_oper_context, ext_field_context)==0) {
        /* electric fields */
        if (oper_num_pert==1 && oper_pert_labels[0]==PERT_DIPOLE) {
            /* checks if the matrix is assembled or not */
            for (imat=0; imat<num_int; imat++) {
                ierr = QcMatIsAssembled(val_int[imat], &assembled);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatIsAssembled");
                    exit(ierr);
                }
                if (assembled==QFALSE) {
                    ierr = QcMatBlockCreate(val_int[imat], 1);
                    if (ierr!=QSUCCESS) {
                        printf("get_one_oper_mat>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatBlockCreate");
                        exit(ierr);
                    }
                    ierr = QcMatSetSymType(val_int[imat], QSYMMAT);
                    if (ierr!=QSUCCESS) {
                        printf("get_one_oper_mat>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatSetSymType");
                        exit(ierr);
                    }
                    ierr = QcMatSetDataType(val_int[imat],
```

```
                                    1,
                                    idx_block_row,
                                    idx_block_col,
                                    data_type);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetDataType");
                    exit(ierr);
                }
                ierr = QcMatSetDimMat(val_int[imat], NUM_AO, NUM_AO);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetDimMat");
                    exit(ierr);
                }
                ierr = QcMatAssemble(val_int[imat]);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatAssemble");
                    exit(ierr);
                }
            }
        }
        /* dipole length integrals */
        if (oper_pert_orders[0]==1) {
            for (imat=0; imat<3; imat++) {
                ierr = QcMatSetValues(val_int[imat],
                                      IDX_BLOCK_ROW,
                                      IDX_BLOCK_COL,
                                      IDX_FIRST_ROW,
                                      NUM_AO,
                                      IDX_FIRST_COL,
                                      NUM_AO,
                                      &values_diplen[imat*SIZE_AO_MAT],
                                      NULL);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetValues");
                    exit(ierr);
                }
            }
        }
        /* zero integrals */
        else {
            for (imat=0; imat<3; imat++) {
                ierr = QcMatZeroEntries(val_int[imat]);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_mat>> error happened at %s: %s\n",
```

```
                                FILE_AND_LINE,
                                "calling QcMatZeroEntries");
                        exit(ierr);
                    }
                }
            }
        }
        else {
            printf("get_one_oper_mat>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
    }
    else {
        printf("get_one_oper_mat>> unknown one-electron operator at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
#else
    /* electric fields */
    if (oper_num_pert==1 && oper_pert_labels[0]==PERT_DIPOLE) {
        /* checks if the matrix is assembled or not */
        for (imat=0; imat<num_int; imat++) {
            ierr = QcMatIsAssembled(val_int[imat], &assembled);
            if (ierr!=QSUCCESS) {
                printf("get_one_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatIsAssembled");
                exit(ierr);
            }
            if (assembled==QFALSE) {
                ierr = QcMatBlockCreate(val_int[imat], 1);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatBlockCreate");
                    exit(ierr);
                }
                ierr = QcMatSetSymType(val_int[imat], QSYMMAT);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetSymType");
                    exit(ierr);
                }
                ierr = QcMatSetDataType(val_int[imat],
                                        num_blocks,
                                        idx_block_row,
                                        idx_block_col,
                                        data_type);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_mat>> error happened at %s: %s\n",
```

```
                    FILE_AND_LINE,
                    "calling QcMatSetDataType");
                exit(ierr);
            }
            ierr = QcMatSetDimMat(val_int[imat], NUM_AO, NUM_AO);
            if (ierr!=QSUCCESS) {
                printf("get_one_oper_mat>> error happened at %s: %s\n",
                    FILE_AND_LINE,
                    "calling QcMatSetDimMat");
                exit(ierr);
            }
            ierr = QcMatAssemble(val_int[imat]);
            if (ierr!=QSUCCESS) {
                printf("get_one_oper_mat>> error happened at %s: %s\n",
                    FILE_AND_LINE,
                    "calling QcMatAssemble");
                exit(ierr);
            }
        }
    }
    /* dipole length integrals */
    if (oper_pert_orders[0]==1) {
        for (imat=0; imat<3; imat++) {
            ierr = QcMatSetValues(val_int[imat],
                                  IDX_BLOCK_ROW,
                                  IDX_BLOCK_COL,
                                  IDX_FIRST_ROW,
                                  NUM_AO,
                                  IDX_FIRST_COL,
                                  NUM_AO,
                                  values_diplen[imat*SIZE_AO_MAT],
                                  NULL);
            if (ierr!=QSUCCESS) {
                printf("get_one_oper_mat>> error happened at %s: %s\n",
                    FILE_AND_LINE,
                    "calling QcMatSetValues");
                exit(ierr);
            }
        }
    }
    /* zero integrals */
    else {
        for (imat=0; imat<3; imat++) {
            ierr = QcMatZeroEntries(val_int[imat]);
            if (ierr!=QSUCCESS) {
                printf("get_one_oper_mat>> error happened at %s: %s\n",
                    FILE_AND_LINE,
                    "calling QcMatZeroEntries");
                exit(ierr);
            }
        }
    }
```

```
    }
    else {
        printf("get_one_oper_mat>> not implemented at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
#endif
}

void get_one_oper_exp(const QInt oper_num_pert,
                      const QcPertInt *oper_pert_labels,
                      const QInt *oper_pert_orders,
                      const QInt num_dmat,
                      QcMat *dens_mat[],
#if defined(OPENRSP_C_USER_CONTEXT)
                      void *user_ctx,
#endif
                      const QInt num_exp,
                      QReal *val_exp)
{
#include "OpenRSPTestPerturbations.h"
#include "ao_dens_ground_state_hf/num_atomic_orbitals.h"
#include "ao_dens_alpha_hf/dipole_length_integrals.h"
#if defined(OPENRSP_C_USER_CONTEXT)
    char *oneham_context = "ONEHAM";
    char *ext_field_context = "EXT_FIELD";
    char *one_oper_context;
#endif
    QInt idx_block_row[1] = {IDX_BLOCK_ROW};
    QInt idx_block_col[1] = {IDX_BLOCK_COL};
    QcDataType data_type[1] = {QREALMAT};
    QcMat val_int[1];
    QInt offset_exp;
    QInt imat;
    QInt idens;
    QErrorCode ierr;
#if defined(OPENRSP_C_USER_CONTEXT)
    one_oper_context = (char *)user_ctx;
    if (strcmp(one_oper_context, oneham_context)==0) {
        /* electric fields (zero integrals) */
        if (oper_num_pert==1 && oper_pert_labels[0]==PERT_DIPOLE) {
            for (idens=0; idens<2*num_exp; idens++) {
                val_exp[idens] = 0;
            }
        }
        else {
            printf("get_one_oper_exp>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
    }
    else if (strcmp(one_oper_context, ext_field_context)==0) {
```

```
    /* electric fields */
    if (oper_num_pert==1 && oper_pert_labels[0]==PERT_DIPOLE) {
        /* dipole length integrals */
        if (oper_pert_orders[0]==1) {
            offset_exp = 0;
            for (imat=0; imat<3; imat++) {
                ierr = QcMatCreate(&val_int[0]);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatCreate");
                    exit(ierr);
                }
                ierr = QcMatBlockCreate(&val_int[0], 1);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatBlockCreate");
                    exit(ierr);
                }
                ierr = QcMatSetSymType(&val_int[0], QSYMMAT);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetSymType");
                    exit(ierr);
                }
                ierr = QcMatSetDataType(&val_int[0],
                                        1,
                                        idx_block_row,
                                        idx_block_col,
                                        data_type);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetDataType");
                    exit(ierr);
                }
                ierr = QcMatSetDimMat(&val_int[0], NUM_AO, NUM_AO);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetDimMat");
                    exit(ierr);
                }
                ierr = QcMatAssemble(&val_int[0]);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatAssemble");
                    exit(ierr);
                }
```

```
                ierr = QcMatSetValues(&val_int[0],
                                      IDX_BLOCK_ROW,
                                      IDX_BLOCK_COL,
                                      IDX_FIRST_ROW,
                                      NUM_AO,
                                      IDX_FIRST_COL,
                                      NUM_AO,
                                      &values_diplen[imat*SIZE_AO_MAT],
                                      NULL);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetValues");
                    exit(ierr);
                }
                for (idens=0; idens<num_dmat; idens++) {
                    ierr = QcMatGetMatProdTrace(&val_int[0],
                                                dens_mat[idens],
                                                MAT_NO_OPERATION,
                                                1,
                                                &val_exp[offset_exp+2*idens]);
                    if (ierr!=QSUCCESS) {
                        printf("get_one_oper_exp>> error happened at %s: %s\n",
                               FILE_AND_LINE,
                               "calling QcMatGetMatProdTrace");
                        exit(ierr);
                    }
                }
                ierr = QcMatDestroy(&val_int[0]);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatDestroy");
                    exit(ierr);
                }
                offset_exp += 2*num_dmat;
            }
        }
        /* zero integrals */
        else {
            for (idens=0; idens<2*num_exp; idens++) {
                val_exp[idens] = 0;
            }
        }
    }
    else {
        printf("get_one_oper_exp>> not implemented at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
}
else {
```

```
        printf("get_one_oper_exp>> unknown one-electron operator at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
#else
    /* electric fields */
    if (oper_num_pert==1 && oper_pert_labels[0]==PERT_DIPOLE) {
        /* dipole length integrals */
        if (oper_pert_orders[0]==1) {
            offset_exp = 0;
            for (imat=0; imat<3; imat++) {
                ierr = QcMatCreate(&val_int[0]);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatCreate");
                    exit(ierr);
                }
                ierr = QcMatBlockCreate(&val_int[0], 1);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatBlockCreate");
                    exit(ierr);
                }
                ierr = QcMatSetSymType(&val_int[0], QSYMMAT);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetSymType");
                    exit(ierr);
                }
                ierr = QcMatSetDataType(&val_int[0],
                                        1,
                                        idx_block_row,
                                        idx_block_col,
                                        data_type);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetDataType");
                    exit(ierr);
                }
                ierr = QcMatSetDimMat(&val_int[0], NUM_AO, NUM_AO);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatSetDimMat");
                    exit(ierr);
                }
                ierr = QcMatAssemble(&val_int[0]);
                if (ierr!=QSUCCESS) {
```

```
                printf("get_one_oper_exp>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatAssemble");
                exit(ierr);
            }
            ierr = QcMatSetValues(&val_int[0],
                                  IDX_BLOCK_ROW,
                                  IDX_BLOCK_COL,
                                  IDX_FIRST_ROW,
                                  NUM_AO,
                                  IDX_FIRST_COL,
                                  NUM_AO,
                                  values_diplen[imat*SIZE_AO_MAT],
                                  NULL);
            if (ierr!=QSUCCESS) {
                printf("get_one_oper_exp>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatSetValues");
                exit(ierr);
            }
            for (idens=0; idens<num_dmat; idens++) {
                ierr = QcMatGetMatProdTrace(&val_int[0],
                                            dens_mat[idens],
                                            MAT_NO_OPERATION,
                                            1,
                                            &val_exp[offset_exp+2*idens]);
                if (ierr!=QSUCCESS) {
                    printf("get_one_oper_exp>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatGetMatProdTrace");
                    exit(ierr);
                }
            }
            ierr = QcMatDestroy(val_int[0]);
            if (ierr!=QSUCCESS) {
                printf("get_one_oper_exp>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatDestroy");
                exit(ierr);
            }
            offset_exp += 2*num_dmat;
        }
    }
    /* zero integrals */
    else {
        for (idens=0; idens<2*num_exp; idens++) {
            val_exp[idens] = 0;
        }
    }
else {
    printf("get_one_oper_exp>> not implemented at %s\n",
           FILE_AND_LINE);
```

```
        exit(QFAILURE);
    }
#endif
}

void get_two_oper_mat(const QInt oper_num_pert,
                      const QcPertInt *oper_pert_labels,
                      const QInt *oper_pert_orders,
                      const QInt num_dmat,
                      QcMat *dens_mat[],
#if defined(OPENRSP_C_USER_CONTEXT)
                      void *user_ctx,
#endif
                      const QInt num_int,
                      QcMat *val_int[])
{
#include "OpenRSPTestPerturbations.h"
#include "ao_dens_ground_state_hf/num_atomic_orbitals.h"
#include "ao_dens_alpha_hf/two_electron_integrals.h"
#if defined(OPENRSP_C_USER_CONTEXT)
    char *two_oper_context;
#endif
    QInt idx_block_row[1] = {IDX_BLOCK_ROW};
    QInt idx_block_col[1] = {IDX_BLOCK_COL};
    QcDataType data_type[1] = {QREALMAT};
    QBool assembled;
    QInt imat;
    QErrorCode ierr;
    static QInt id_gmat = -1;
#if defined(OPENRSP_C_USER_CONTEXT)
    two_oper_context = (char *)user_ctx;
    if (strcmp(two_oper_context, "NONLAO")!=0) {
        printf("get_two_oper_mat>> not implemented at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
    else {
        /* checks if the matrix is assembled or not */
        for (imat=0; imat<num_int; imat++) {
            ierr = QcMatIsAssembled(val_int[imat], &assembled);
            if (ierr!=QSUCCESS) {
                printf("get_two_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatIsAssembled");
                exit(ierr);
            }
            if (assembled==QFALSE) {
                ierr = QcMatBlockCreate(val_int[imat], 1);
                if (ierr!=QSUCCESS) {
                    printf("get_two_oper_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatBlockCreate");
```

```
                exit(ierr);
            }
            ierr = QcMatSetSymType(val_int[imat], QSYMMAT);
            if (ierr!=QSUCCESS) {
                printf("get_two_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatSetSymType");
                exit(ierr);
            }
            ierr = QcMatSetDataType(val_int[imat],
                                    1,
                                    idx_block_row,
                                    idx_block_col,
                                    data_type);
            if (ierr!=QSUCCESS) {
                printf("get_two_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatSetDataType");
                exit(ierr);
            }
            ierr = QcMatSetDimMat(val_int[imat], NUM_AO, NUM_AO);
            if (ierr!=QSUCCESS) {
                printf("get_two_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatSetDimMat");
                exit(ierr);
            }
            ierr = QcMatAssemble(val_int[imat]);
            if (ierr!=QSUCCESS) {
                printf("get_two_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatAssemble");
                exit(ierr);
            }
        }
    }
    /* unperturbed two-electron integrals */
    if (oper_num_pert==0) {
        id_gmat++;
        if (id_gmat>5) {
            printf("get_two_oper_mat>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
        ierr = QcMatSetValues(val_int[0],
                              IDX_BLOCK_ROW,
                              IDX_BLOCK_COL,
                              IDX_FIRST_ROW,
                              NUM_AO,
                              IDX_FIRST_COL,
                              NUM_AO,
                              &values_gmat[id_gmat*SIZE_AO_MAT],
```

```
                                NULL);
            if (ierr!=QSUCCESS) {
                printf("get_two_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatSetValues");
                exit(ierr);
            }
        }
        /* electric fields (zero integrals) */
        else if (oper_num_pert==1 && oper_pert_labels[0]==PERT_DIPOLE) {
            for (imat=0; imat<num_int; imat++) {
                ierr = QcMatZeroEntries(val_int[imat]);
                if (ierr!=QSUCCESS) {
                    printf("get_two_oper_mat>> error happened at %s: %s\n",
                           FILE_AND_LINE,
                           "calling QcMatZeroEntries");
                    exit(ierr);
                }
            }
        }
        else {
            printf("get_two_oper_mat>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
    }
#else
    /* checks if the matrix is assembled or not */
    for (imat=0; imat<num_int; imat++) {
        ierr = QcMatIsAssembled(val_int[imat], &assembled);
        if (ierr!=QSUCCESS) {
            printf("get_two_oper_mat>> error happened at %s: %s\n",
                   FILE_AND_LINE,
                   "calling QcMatIsAssembled");
            exit(ierr);
        }
        if (assembled==QFALSE) {
            ierr = QcMatBlockCreate(val_int[imat], 1);
            if (ierr!=QSUCCESS) {
                printf("get_two_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatBlockCreate");
                exit(ierr);
            }
            ierr = QcMatSetSymType(val_int[imat], QSYMMAT);
            if (ierr!=QSUCCESS) {
                printf("get_two_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatSetSymType");
                exit(ierr);
            }
            ierr = QcMatSetDataType(val_int[imat],
```

```
                                num_blocks,
                                idx_block_row,
                                idx_block_col,
                                data_type);
            if (ierr!=QSUCCESS) {
                printf("get_two_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatSetDataType");
                exit(ierr);
            }
            ierr = QcMatSetDimMat(val_int[imat], NUM_AO, NUM_AO);
            if (ierr!=QSUCCESS) {
                printf("get_two_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatSetDimMat");
                exit(ierr);
            }
            ierr = QcMatAssemble(val_int[imat]);
            if (ierr!=QSUCCESS) {
                printf("get_two_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatAssemble");
                exit(ierr);
            }
        }
    }
    /* unperturbed two-electron integrals */
    if (oper_num_pert==0) {
        id_gmat++;
        if (id_gmat>5) {
            printf("get_two_oper_mat>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
        ierr = QcMatSetValues(val_int[0],
                              IDX_BLOCK_ROW,
                              IDX_BLOCK_COL,
                              IDX_FIRST_ROW,
                              NUM_AO,
                              IDX_FIRST_COL,
                              NUM_AO,
                              values_gmat[id_gmat*SIZE_AO_MAT],
                              NULL);
        if (ierr!=QSUCCESS) {
            printf("get_two_oper_mat>> error happened at %s: %s\n",
                   FILE_AND_LINE,
                   "calling QcMatSetValues");
            exit(ierr);
        }
    }
    /* electric fields (zero integrals) */
    else if (oper_num_pert==1 && oper_pert_labels[0]==PERT_DIPOLE) {
```

```
        for (imat=0; imat<num_int; imat++) {
            ierr = QcMatZeroEntries(val_int[imat]);
            if (ierr!=QSUCCESS) {
                printf("get_two_oper_mat>> error happened at %s: %s\n",
                       FILE_AND_LINE,
                       "calling QcMatZeroEntries");
                exit(ierr);
            }
        }
    }
    else {
        printf("get_two_oper_mat>> not implemented at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
#endif
}

void get_two_oper_exp(const QInt oper_num_pert,
                      const QcPertInt *oper_pert_labels,
                      const QInt *oper_pert_orders,
                      const QInt dmat_len_tuple,
                      const QInt *num_LHS_dmat,
                      QcMat *LHS_dens_mat[],
                      const QInt *num_RHS_dmat,
                      QcMat *RHS_dens_mat[],
#if defined(OPENRSP_C_USER_CONTEXT)
                      void *user_ctx,
#endif
                      const QInt num_exp,
                      QReal *val_exp)
{
#include "OpenRSPTestPerturbations.h"
#if defined(OPENRSP_C_USER_CONTEXT)
    char *two_oper_context;
#endif
    QInt ival;
    QErrorCode ierr;
#if defined(OPENRSP_C_USER_CONTEXT)
    two_oper_context = (char *)user_ctx;
    if (strcmp(two_oper_context, "NONLAO")!=0) {
        printf("get_two_oper_exp>> not implemented at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
    else {
        /* electric fields */
        if (oper_num_pert==1 && oper_pert_labels[0]==PERT_DIPOLE) {
            for (ival=0; ival<2*num_exp; ival++) {
                val_exp[ival] = 0;
            }
        }
```

```
        else {
            printf("get_two_oper_exp>> not implemented at %s\n",
                   FILE_AND_LINE);
            exit(QFAILURE);
        }
    }
#else
    /* electric fields */
    if (oper_num_pert==1 && oper_pert_labels[0]==PERT_DIPOLE) {
        for (ival=0; ival<2*num_exp; ival++) {
            val_exp[ival] = 0;
        }
    }
    else {
        printf("get_two_oper_exp>> not implemented at %s\n",
               FILE_AND_LINE);
        exit(QFAILURE);
    }
#endif
}

void get_nuc_contrib(const QInt nuc_num_pert,
                     const QcPertInt *nuc_pert_labels,
                     const QInt *nuc_pert_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                     void *user_ctx,
#endif
                     const QInt size_pert,
                     QReal *val_nuc)
{
}

void get_linear_rsp_solution(const QInt size_pert,
                             const QReal *freq_sums,
                             const QInt num_freq_sums,
                             QcMat *RHS_mat[],
#if defined(OPENRSP_C_USER_CONTEXT)
                             void *user_ctx,
#endif
                             QcMat *rsp_param[])
{
#include "ao_dens_ground_state_hf/num_atomic_orbitals.h"
#include "ao_dens_alpha_hf/response_parameters.h"
    QInt idx_block_row[1] = {IDX_BLOCK_ROW};
    QInt idx_block_col[1] = {IDX_BLOCK_COL};
    QcDataType data_type[1] = {QREALMAT};
    QBool assembled;
    QErrorCode ierr;
    static QInt id_rsp_param = -1;
    id_rsp_param++;
    if (id_rsp_param>2) {
        printf("get_linear_rsp_solution>> not implemented at %s\n",
```

```
            FILE_AND_LINE);
    exit(QFAILURE);
}
/* checks if the matrix is assembled or not */
ierr = QcMatIsAssembled(rsp_param[0], &assembled);
if (ierr!=QSUCCESS) {
    printf("get_linear_rsp_solution>> error happened at %s: %s\n",
           FILE_AND_LINE,
           "calling QcMatIsAssembled");
    exit(ierr);
}
if (assembled==QFALSE) {
    ierr = QcMatBlockCreate(rsp_param[0], 1);
    if (ierr!=QSUCCESS) {
        printf("get_linear_rsp_solution>> error happened at %s: %s\n",
               FILE_AND_LINE,
               "calling QcMatBlockCreate");
        exit(ierr);
    }
    ierr = QcMatSetDataType(rsp_param[0],
                            1,
                            idx_block_row,
                            idx_block_col,
                            data_type);
    if (ierr!=QSUCCESS) {
        printf("get_linear_rsp_solution>> error happened at %s: %s\n",
               FILE_AND_LINE,
               "calling QcMatSetDataType");
        exit(ierr);
    }
    ierr = QcMatSetDimMat(rsp_param[0], NUM_AO, NUM_AO);
    if (ierr!=QSUCCESS) {
        printf("get_linear_rsp_solution>> error happened at %s: %s\n",
               FILE_AND_LINE,
               "calling QcMatSetDimMat");
        exit(ierr);
    }
    ierr = QcMatAssemble(rsp_param[0]);
    if (ierr!=QSUCCESS) {
        printf("get_linear_rsp_solution>> error happened at %s: %s\n",
               FILE_AND_LINE,
               "calling QcMatAssemble");
        exit(ierr);
    }
}
ierr = QcMatSetValues(rsp_param[0],
                      IDX_BLOCK_ROW,
                      IDX_BLOCK_COL,
                      IDX_FIRST_ROW,
                      NUM_AO,
                      IDX_FIRST_COL,
                      NUM_AO,
```

```
                            &alpha_rsp_param[id_rsp_param*SIZE_AO_MAT],
                            NULL);
        if (ierr!=QSUCCESS) {
            printf("get_linear_rsp_solution>> error happened at %s: %s\n",
                   FILE_AND_LINE,
                   "calling QcMatSetValues");
            exit(ierr);
        }
    }
```

The atomic-orbital density matrix-based response theory is tested by:

213a ⟨*OpenRSPAODensTest.h* 213a⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

   This is the header file of unit testing of the AO density matrix-based
   response theory.

   2014-07-31, Bin Gao
   * first version
*/
#if !defined(OPENRSP_AO_DENS_TEST_H)
#define OPENRSP_AO_DENS_TEST_H

#include "OpenRSPPertCallback.h"
#include "OpenRSPAODensCallback.h"

extern QErrorCode OpenRSPAODensTest(OpenRSP*,FILE*);

#endif
```

213b ⟨*OpenRSPAODensTest.c* 213b⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

   This file tests the AO density matrix-based response theory.

   2014-07-31, Bin Gao
   * first version
*/

#include "OpenRSPAODensTest.h"

QErrorCode OpenRSPAODensTest(OpenRSP *open_rsp, FILE *fp_log)
{
#include "OpenRSPTestPerturbations.h"
#if defined(OPENRSP_C_USER_CONTEXT)
    /* user defined context of linear response equation solver */
    char *solver_context = "SOLVER";
#endif
    /* overlap integrals with London atomic orbitals */
    QInt overlap_num_pert = 2;
    QcPertInt overlap_pert_labels[2] = {PERT_GEOMETRIC,PERT_MAGNETIC};
```

```
    QInt overlap_pert_orders[2] = {MAX_ORDER_GEOMETRIC,MAX_ORDER_MAGNETIC};
#if defined(OPENRSP_C_USER_CONTEXT)
    char *overlap_context = "OVERLAP";
#endif
    /* one-electron Hamiltonian */
    QInt oneham_num_pert = 2;
    QcPertInt oneham_pert_labels[2] = {PERT_GEOMETRIC,PERT_MAGNETIC};
    QInt oneham_pert_orders[2] = {MAX_ORDER_GEOMETRIC,MAX_ORDER_MAGNETIC};
#if defined(OPENRSP_C_USER_CONTEXT)
    char *oneham_context = "ONEHAM";
#endif
    /* external field */
    QInt ext_field_num_pert = 3;
    QcPertInt ext_field_pert_labels[3] = {PERT_GEOMETRIC,PERT_DIPOLE,PERT_MAGNETIC};
    QInt ext_field_pert_orders[3] = {MAX_ORDER_GEOMETRIC,MAX_ORDER_DIPOLE,MAX_ORDER_MAGNETI
#if defined(OPENRSP_C_USER_CONTEXT)
    char *ext_field_context = "EXT_FIELD";
#endif
    /* two-electron Hamiltonian */
    QInt twoel_num_pert = 2;
    QcPertInt twoel_pert_labels[2] = {PERT_GEOMETRIC,PERT_MAGNETIC};
    QInt twoel_pert_orders[2] = {MAX_ORDER_GEOMETRIC,MAX_ORDER_MAGNETIC};
#if defined(OPENRSP_C_USER_CONTEXT)
    char *twoel_context = "NONLAO";
#endif
    /* referenced state */
#include "ao_dens_ground_state_hf/num_atomic_orbitals.h"
#include "ao_dens_ground_state_hf/fock_matrix.h"
#include "ao_dens_ground_state_hf/density_matrix.h"
#include "ao_dens_ground_state_hf/overlap_integrals.h"
    QInt idx_block_row[1] = {IDX_BLOCK_ROW};
    QInt idx_block_col[1] = {IDX_BLOCK_COL};
    QcDataType data_type[1] = {QREALMAT};
    QcMat F_unpert;
    QcMat D_unpert;
    QcMat S_unpert;
    /* polarizability */
    QInt ALPHA_NUM_PROPS = 1;
    QInt ALPHA_LEN_TUPLE[1] = {2};
    QcPertInt ALPHA_PERT_TUPLE[2] = {PERT_DIPOLE,PERT_DIPOLE};
    QInt ALPHA_NUM_FREQ_CONFIGS[1] = {1};
    QReal ALPHA_PERT_FREQS[4] = {-0.072,0.0,0.072,0.0};
    QInt ALPHA_KN_RULES[1] = {0};
    /* response functions */
    QInt size_rsp_funs;
    QReal rsp_funs[18];
    QInt ipert,jpert,ival;
    /* error information */
    QErrorCode ierr;

    ///* sets the equation of motion of electrons */
    //ierr = OpenRSPSetElecEOM(open_rsp, ELEC_AO_D_MATRIX);
```

```
    //QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPSetElecEOM");
    //fprintf(fp_log, "OpenRSPAODensTest>> OpenRSPSetElecEOM() passed\n");

    /* sets the context of linear response equation solver */
    ierr = OpenRSPSetLinearRSPSolver(open_rsp,
#if defined(OPENRSP_C_USER_CONTEXT)
                                     (void *)solver_context,
#endif
                                     &get_linear_rsp_solution);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPSetLinearRSPSolver");
    fprintf(fp_log, "OpenRSPAODensTest>> OpenRSPSetLinearRSPSolver() passed\n");

    /* sets the context of perturbation dependent basis sets */
    ierr = OpenRSPSetOverlap(open_rsp,
                             overlap_num_pert,
                             overlap_pert_labels,
                             overlap_pert_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                             (void *)overlap_context,
#endif
                             &get_overlap_mat,
                             &get_overlap_exp);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPSetOverlap");
    fprintf(fp_log, "OpenRSPAODensTest>> OpenRSPSetOverlap() passed\n");

    /* adds one-electron Hamiltonian */
    ierr = OpenRSPAddOneOper(open_rsp,
                             oneham_num_pert,
                             oneham_pert_labels,
                             oneham_pert_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                             (void *)oneham_context,
#endif
                             &get_one_oper_mat,
                             &get_one_oper_exp);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPAddOneOper(h)");
    fprintf(fp_log, "OpenRSPAODensTest>> OpenRSPAddOneOper(h) passed\n");

    /* adds external field */
    ierr = OpenRSPAddOneOper(open_rsp,
                             ext_field_num_pert,
                             ext_field_pert_labels,
                             ext_field_pert_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                             (void *)ext_field_context,
#endif
                             &get_one_oper_mat,
                             &get_one_oper_exp);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPAddOneOper(V)");
    fprintf(fp_log, "OpenRSPAODensTest>> OpenRSPAddOneOper(V) passed\n");

    /* adds two-electron Hamiltonian */
```

```
    ierr = OpenRSPAddTwoOper(open_rsp,
                             twoel_num_pert,
                             twoel_pert_labels,
                             twoel_pert_orders,
#if defined(OPENRSP_C_USER_CONTEXT)
                             (void *)twoel_context,
#endif
                             &get_two_oper_mat,
                             &get_two_oper_exp);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPAddTwoOper()");
    fprintf(fp_log, "OpenRSPAODensTest>> OpenRSPAddTwoOper() passed\n");

    /* assembles the context of response theory calculations */
    ierr = OpenRSPAssemble(open_rsp);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPAssemble");
    fprintf(fp_log, "OpenRSPAODensTest>> OpenRSPAssemble() passed\n");

    /* writes the context of response theory calculations */
    ierr = OpenRSPWrite(open_rsp, "OpenRSPAODensTest.log");
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPWrite");
    fprintf(fp_log, "OpenRSPAODensTest>> OpenRSPWrite() passed\n");

    /* sets the unperturbed Fock matrix */
    ierr = QcMatCreate(&F_unpert);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatCreate(F)");
    ierr = QcMatBlockCreate(&F_unpert, 1);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatBlockCreate(F)");
    ierr = QcMatSetSymType(&F_unpert, QSYMMAT);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatSetSymType(F)");
    ierr = QcMatSetDataType(&F_unpert, 1, idx_block_row, idx_block_col, data_type);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatSetDataType(F)");
    ierr = QcMatSetDimMat(&F_unpert, NUM_AO, NUM_AO);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatSetDimMat(F)");
    ierr = QcMatAssemble(&F_unpert);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatAssemble(F)");
    ierr = QcMatSetValues(&F_unpert,
                          IDX_BLOCK_ROW,
                          IDX_BLOCK_COL,
                          IDX_FIRST_ROW,
                          NUM_AO,
                          IDX_FIRST_COL,
                          NUM_AO,
                          values_fock,
                          NULL);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatSetValues(F)");
    /* sets the unperturbed density matrix */
    ierr = QcMatCreate(&D_unpert);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatCreate(D)");
    ierr = QcMatBlockCreate(&D_unpert, 1);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatBlockCreate(D)");
    ierr = QcMatSetSymType(&D_unpert, QSYMMAT);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatSetSymType(D)");
```

```
ierr = QcMatSetDataType(&D_unpert, 1, idx_block_row, idx_block_col, data_type);
QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatSetDataType(D)");
ierr = QcMatSetDimMat(&D_unpert, NUM_AO, NUM_AO);
QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatSetDimMat(D)");
ierr = QcMatAssemble(&D_unpert);
QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatAssemble(D)");
ierr = QcMatSetValues(&D_unpert,
                      IDX_BLOCK_ROW,
                      IDX_BLOCK_COL,
                      IDX_FIRST_ROW,
                      NUM_AO,
                      IDX_FIRST_COL,
                      NUM_AO,
                      values_density,
                      NULL);
QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatSetValues(D)");
/* sets the unperturbed overlap integrals */
ierr = QcMatCreate(&S_unpert);
QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatCreate(S)");
ierr = QcMatBlockCreate(&S_unpert, 1);
QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatBlockCreate(S)");
ierr = QcMatSetSymType(&S_unpert, QSYMMAT);
QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatSetSymType(S)");
ierr = QcMatSetDataType(&S_unpert, 1, idx_block_row, idx_block_col, data_type);
QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatSetDataType(S)");
ierr = QcMatSetDimMat(&S_unpert, NUM_AO, NUM_AO);
QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatSetDimMat(S)");
ierr = QcMatAssemble(&S_unpert);
QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatAssemble(S)");
ierr = QcMatSetValues(&S_unpert,
                      IDX_BLOCK_ROW,
                      IDX_BLOCK_COL,
                      IDX_FIRST_ROW,
                      NUM_AO,
                      IDX_FIRST_COL,
                      NUM_AO,
                      values_overlap,
                      NULL);
QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatSetValues(S)");

/* gets the polarizability */
size_rsp_funs = 9;
ierr = OpenRSPGetRSPFun(open_rsp,
                        &F_unpert,
                        &D_unpert,
                        &S_unpert,
                        ALPHA_NUM_PROPS,
                        ALPHA_LEN_TUPLE,
                        ALPHA_PERT_TUPLE,
                        ALPHA_NUM_FREQ_CONFIGS,
                        ALPHA_PERT_FREQS,
                        ALPHA_KN_RULES,
```

```
                            size_rsp_funs,
                            rsp_funs);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPGetRSPFun");
    fprintf(fp_log, "OpenRSPAODensTest>> OpenRSPGetRSPFun() passed\n");
    for (ipert=0,ival=0; ipert<3; ipert++) {
        for (jpert=0; jpert<3; jpert++) {
            fprintf(fp_log, " (%f,%f)", rsp_funs[ival], rsp_funs[ival+1]);
            ival += 2;
        }
        fprintf(fp_log, "\n");
    }

    /* cleans */
    ierr = QcMatDestroy(&F_unpert);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatDestroy(F)");
    ierr = QcMatDestroy(&D_unpert);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatDestroy(D)");
    ierr = QcMatDestroy(&S_unpert);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling QcMatDestroy(S)");

    return QSUCCESS;
}
```

218a ⟨*OpenRSPTest.h* 218a⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

  This is the header file of OpenRSP unit testing.

  2014-07-31, Bin Gao:
  * first version
*/

#if !defined(OPENRSP_TEST_H)
#define OPENRSP_TEST_H

#include <string.h>
#include "OpenRSP.h"

#include "OpenRSPAODensTest.h"

#endif
```

218b ⟨*OpenRSPTest.c* 218b⟩≡

```
/*
  ⟨OpenRSPLicense 14a⟩

   This file tests the APIs of OpenRSP library.

   2014-07-31, Bin Gao
   * first version
*/
```

```
#include "OpenRSPTest.h"

/* <macrodef name='OPENRSP_TEST_EXECUTABLE'>
     Build test suite as excutables.
   </macrodef> */
#if defined(OPENRSP_TEST_EXECUTABLE)
QErrorCode main()
{
    FILE *fp_log=stdout;  /* file pointer */
#else
QErrorCode test_c_OpenRSP(FILE *fp_log)
{
#endif
#include "OpenRSPTestPerturbations.h"
    OpenRSP open_rsp;                                    /* context of response theory calcula
    const QcPertInt ALL_PERT_LABELS[NUM_ALL_PERT] = {      /* labels of all perturbations *
        PERT_GEOMETRIC,PERT_DIPOLE,PERT_MAGNETIC};
    const QInt ALL_PERT_MAX_ORDERS[NUM_ALL_PERT] = {  /* maximum allowed orders of all pert
        MAX_ORDER_GEOMETRIC,MAX_ORDER_DIPOLE,MAX_ORDER_MAGNETIC};
    const QInt ALL_PERT_SIZES[] = {                      /* sizes of all perturbations up to t
        12,78,364,1365,4368,12376,31824,                 /* geometric derivatives (4 atoms) */
        3,                                               /* electric dipole */
        3,6,10,15,21,28,36};                             /* magnetic derivatives */
#if defined(OPENRSP_C_USER_CONTEXT)
    char *pert_context = "NRNZGEO";                     /* user defined context for perturbati
#endif
    QErrorCode ierr;                                     /* error information */
    const QInt NUM_ATOMS = 4;

    /* creates the context of response theory calculations */
    ierr = OpenRSPCreate(&open_rsp);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPCreate");
    fprintf(fp_log, "test_c_OpenRSP>> OpenRSPCreate() passed\n");

    /* sets information of all perturbations */
    ierr = OpenRSPSetPerturbations(&open_rsp,
                                   NUM_ALL_PERT,
                                   ALL_PERT_LABELS,
                                   ALL_PERT_MAX_ORDERS,
                                   ALL_PERT_SIZES,
#if defined(OPENRSP_C_USER_CONTEXT)
                                   (void *)pert_context,
#endif
                                   &get_pert_concatenation);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPSetPerturbations");
    fprintf(fp_log, "test_c_OpenRSP>> OpenRSPSetPerturbations() passed\n");

    /* sets the nuclear Hamiltonian */
    ierr = OpenRSPSetNucHamilton(&open_rsp,
                                 NUM_ALL_PERT,
                                 ALL_PERT_LABELS,
```

```
                                    ALL_PERT_MAX_ORDERS,
#if defined(OPENRSP_C_USER_CONTEXT)
                                    (void *)pert_context,
#endif
                                    &get_nuc_contrib,
                                    NUM_ATOMS);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPSetNucHamilton");
    fprintf(fp_log, "test_c_OpenRSP>> OpenRSPSetNucHamilton() passed\n");

    /* tests the density matrix-based response theory */
    ierr = OpenRSPAODensTest(&open_rsp, fp_log);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPAODensTest");
    fprintf(fp_log, "test_c_OpenRSP>> density matrix-based response theory passed\n");

    /* destroys the context of response theory calculations */
    ierr = OpenRSPDestroy(&open_rsp);
    QErrorCheckCode(ierr, FILE_AND_LINE, "calling OpenRSPDestroy");
    fprintf(fp_log, "test_c_OpenRSP>> OpenRSPDestroy() passed\n");

    return QSUCCESS;
}
```

3.13.2 Testing Fortran APIs

220 ⟨*OpenRSPTest.F90* 220⟩≡

```
!! <QcLang='Fortran'>
⟨OpenRSPLicenseFortran 116a⟩
!!
!! This file tests the Fortran APIs of OpenRSP library.
!!
!! 2014-08-03, Bin Gao
!! * first version
```

Chapter 4

Maintenance

4.1 Support and Citation

If there is any question regarding the use of OpenRSP , please contact the authors as given in the file `AUTHORS.rst`.

If you have used OpenRSP and found it is useful, please consider to cite OpenRSP as described in the file `openrsp.bib`.

4.2 List of Chunks

⟨*OpenRSPAODensCallback.c* 184⟩
⟨*OpenRSPAODensCallback.h* 182b⟩
⟨*OpenRSPAODensTest.c* 213b⟩
⟨*OpenRSPAODensTest.h* 213a⟩
⟨*OpenRSPAPIs* 14c⟩
⟨*OpenRSP.F90* 116b⟩
⟨*OpenRSPFortranAdapter.c* 179⟩
⟨*OpenRSPLicense* 14a⟩
⟨*OpenRSPLicenseFortran* 116a⟩
⟨*OpenRSPPertCallback.c* 182a⟩
⟨*OpenRSPPertCallback.h* 181c⟩
⟨*OpenRSPStruct* 14b⟩
⟨*OpenRSPTest.F90* 220⟩
⟨*OpenRSPTestPerturbations.c* 181b⟩
⟨*OpenRSPTestPerturbations.h* 181a⟩
⟨*OpenRSPTest.c* 218b⟩
⟨*OpenRSPTest.h* 218a⟩
⟨*OpenRSP.c* 17a⟩
⟨*OpenRSP.h* 13⟩
⟨*RSNucHamiltonStruct* 99a⟩
⟨*RSPNucHamiltonAPIs* 99b⟩
⟨*RSPNucHamilton.F90* 172⟩
⟨*RSPNucHamilton.c* 99c⟩
⟨*RSPNucHamilton.h* 98⟩
⟨*RSPOneOperAPIs* 58b⟩
⟨*RSPOneOper.F90* 152⟩
⟨*RSPOneOperStruct* 58a⟩
⟨*RSPOneOper.c* 59⟩
⟨*RSPOneOper.h* 57⟩
⟨*RSPOverlapAPIs* 45a⟩

⟨*RSPOverlap.F90* 144⟩
⟨*RSPOverlapStruct* 44⟩
⟨*RSPOverlap.c* 45b⟩
⟨*RSPOverlap.h* 43⟩
⟨*RSPPertBasicTypes* 22b⟩
⟨*RSPPertBasicTypes.F90* 141a⟩
⟨*RSPPertCallback* 24a⟩
⟨*RSPPertStruct* 24b⟩
⟨*RSPPerturbation.F90* 141b⟩
⟨*RSPPerturbation.c* 23c⟩
⟨*RSPPerturbation.h* 22a⟩
⟨*RSPSolverAPIs* 108b⟩
⟨*RSPSolver.F90* 175⟩
⟨*RSPSolverStruct* 108a⟩
⟨*RSPSolver.c* 108c⟩
⟨*RSPSolver.h* 107b⟩
⟨*RSPTwoOperAPIs* 71b⟩
⟨*RSPTwoOperStruct* 71a⟩
⟨*RSPTwoOper.c* 72⟩
⟨*RSPTwoOper.h* 70⟩
⟨*RSPTwoper.F90* 158⟩
⟨*RSPXCFunAPIs* 85b⟩
⟨*RSPXCFun.F90* 165⟩
⟨*RSPXCFunStruct* 85a⟩
⟨*RSPXCFun.c* 86⟩
⟨*RSPXCFun.h* 84⟩
⟨*RSPertAPIs* 26a⟩

Bibliography

[1] Andreas J. Thorvaldsen, Kenneth Ruud, Kasper Kristensen, Poul Jørgensen, and Sonia Coriani. A density matrix-based quasienergy formulation of the Kohn–Sham density functional response theory using perturbation- and time-dependent basis sets. *J. Chem. Phys.*, 129(21):214108, 2008.

[2] Radovan Bast, Ulf Ekström, Bin Gao, Trygve Helgaker, Kenneth Ruud, and Andreas J. Thorvaldsen. The ab initio calculation of molecular electric, magnetic and geometric properties. *Phys. Chem. Chem. Phys.*, 13(7):2627–2651, 2011.

[3] Radovan Bast, Andreas J. Thorvaldsen, Magnus Ringholm, and Kenneth Ruud. Atomic orbital-based cubic response theory for one-, two-, and four-component relativistic self-consistent field models. *Chem. Phys.*, 356(1–3):177–186, 2009.

[4] Magnus Ringholm, Dan Jonsson, and Kenneth Ruud. A General, Recursive, and Open-Ended Response Code. *J. Comput. Chem.*, 35(8):622–633, 2014.

[5] Daniel H. Friese, Maarten T. P. Beerepoot, Magnus Ringholm, and Kenneth Ruud. Open-Ended Recursive Approach for the Calculation of Multiphoton Absorption Matrix Elements. *J. Chem. Theory Comput.*, 11(3):1129–1144, 2015.

Index