The LIBINT Programmer's Manual

LIBINT Version 1.2

Edward F. Valeev

Center for Computational Molecular Science and Technology,
Georgia Institute of Technology, Atlanta, Georgia 30332-0400

Created on: June 27, 2024

1 Introduction

LIBINT is a collection of functions to compute two-body integrals over Gaussian functions which appear in electronic and molecular structure theories. LIBINT Version 1.2[1] has three components which compute different types of integrals:

- libint computes the Coulomb integrals, which in electronic structure theory are called electron repulsion integrals (ERIs). This is by far the most common type of integrals in molecular structure theory.
- libderiv computes first and second derivatives of ERIs with respect to the coordinates of the basis function origin. This type of integrals are also very common in the electronic structure theory, where they appear in analytic gradient expressions.
- libr12 computes types integrals that appear in Kutzelnigg's linear R12 theories for electronic structure.[2, 3] All linear R12 methods, such as MP2-R12, contain terms in the wave function that are linear in the interelectronic distances r_{ij} (hence the name). Appearance of several types of *two*-body integrals is due to the use of the approximate resolution of the identity to reduce three- and four-body integrals to products of simpler integrals.

The components come as separate library archives, named libint.a, libderiv.a, and libr12.a, with header files named libint.h, libderiv.h, and libr12.h, respectively. Note that both libderiv and libr12 depend on functions in libint. In that sense libint is the central component of LIBINT, thus we will libint most often as an example in this manual.

LIBINT uses recursive schemes that originate in seminal Obara-Saika method[4] and Head-Gordon and Pople's variation thereof.[5] The idea of LIBINT is to optimize *computer implementation* of such methods by implementing an optimizing compiler to generate automatically highly-specialized code that runs well on superscalar architectures. The advantages of the optimizing compiler approach are:

- it allows to achieve high-performance for the *one-quartet-at-a-time* method of computing integrals. Thus LIBINT avoids vectorization as an approach to achieving high efficiency, since vectorization increases memory footprint and complicates programming. If the use of vector machines increases again, LIBINT will be vectorized, however currently there are no firm plans to do that.
- new recurrence relations are rather easy to implement in efficient code. libr12 is a good example of that.

For more details on priciples of LIBINT you should consult Justin Fermann's thesis.[6]

2 Notation

Following Obara and Saika,[4] we write an *unnormalized primitive Cartesian* Gaussian function centered at $\mathbf{A}$ as

$$\begin{aligned}\phi(\mathbf{r};\zeta,\mathbf{n},\mathbf{A}) &= (x-A_x)^{n_x}(y-A_y)^{n_y}(z-A_z)^{n_z} \\ &\quad \times \exp[-\zeta(\mathbf{r}-\mathbf{A})^2] \ , \end{aligned} \tag{1}$$

where $\mathbf{r}$ is the coordinate vector of the electron, ζ is the orbital exponent, and $\mathbf{n}$ is a set of non-negative integers. The sum of n_x, n_y, and n_z will be denoted $\lambda(\mathbf{n})$ and be referred to as the angular momentum or orbital quantum number of the Gaussian function. Hereafter $\mathbf{n}$ will be termed the angular momentum index. Henceforth, n_i will refer to the i-th component of $\mathbf{n}$, where $i \in \{x,y,z\}$. Basic vector addition rules will apply to these vector-like triads of numbers, e.g. $\mathbf{n}+\mathbf{1}_x \equiv \{n_x+1, n_y, n_z\}$.

A set of $(\lambda(\mathbf{n})+1)(\lambda(\mathbf{n})+2)/2$ functions with the same $\lambda(\mathbf{n})$, ζ, and centered at the common center but with different $\mathbf{n}$ form a *Cartesian shell*, or just a *shell*. For example, an s shell ($\lambda=0$) has one function, a p shell ($\lambda=1$) – 3 functions, etc. The order of functions in shells that `LIBINT` uses is as follows:

$$\begin{aligned} p &: p_x, p_y, p_z \\ d &: d_{xx}, d_{xy}, d_{xz}, d_{yy}, d_{yz}, d_{zz} \\ f &: f_{xxx}, f_{xxy}, f_{xxz}, f_{xyy}, f_{xyz}, f_{xzz}, f_{yyy}, f_{yyz}, f_{yzz}, f_{zzz} \\ \text{etc.} & \end{aligned}$$

In general, the following loop structure can be used to generate angular momentum indices in the canonical `LIBINT` order for all members of a shell of angular momentum `am`:

```
for(int i=0; i<=am; i++) {
  int nx = am - i;  /* exponent of x */
  for(int j=0; j<=i; j++) {
    int ny = i-j;   /* exponent of y */
    int nz = j;     /* exponent of z */
  }
}
```

The normalization constant for a primitive Gaussian $\phi(\mathbf{r};\zeta,\mathbf{n},\mathbf{A})$

$$N(\zeta,\mathbf{n}) = \left[\left(\frac{2}{\pi}\right)^{3/4} \frac{2^{(\lambda(\mathbf{n}))}\zeta^{(2\lambda(\mathbf{n})+3)/4}}{[(2n_x-1)!!(2n_y-1)!!(2n_z-1)!!]^{1/2}}\right] \tag{2}$$

A contracted Gaussian function is just a linear combination of primitive Gaussians (also termed *primitives*) centered at the same center $\mathbf{A}$ and with the same momentum indices $\mathbf{n}$ but with different exponents ζ_i:

$$\begin{aligned}\phi(\mathbf{r};\boldsymbol{\zeta},\mathbf{C},\mathbf{n},\mathbf{A}) &= (x-A_x)^{n_x}(y-A_y)^{n_y}(z-A_z)^{n_z} \\ &\quad \times \sum_{i=1}^{M} C_i \exp[-\zeta_i(\mathbf{r}-\mathbf{A})^2] \ , \end{aligned} \tag{3}$$

Contracted Gaussians form shells the same way as primitives. The contraction coefficients **C** already include normalization constants so that the resulting combination is properly normalized. Published contraction coefficients **c** are linear coefficients for normalized primitives, hence the normalization-including contraction coefficients **C** have to be computed from them as

$$C_i = c_i N(\zeta_i, \mathbf{n}) \tag{4}$$

and scaled further so that the self-overlap of the contracted function is 1:

$$\frac{\pi^{3/2}(2n_x-1)!!(2n_y-1)!!(2n_z-1)!!}{2^{\lambda(\mathbf{n})}} \sum_{i=1}^{M}\sum_{j=1}^{M} \frac{C_i C_j}{(\zeta_i+\zeta_j)^{\lambda(\mathbf{n})+3/2}} = 1 \tag{5}$$

If sets of orbital exponents are used to form contracted Gaussians of one angular momentum only then this is called a *segmented* contraction scheme. If there is a set of exponents that forms contracted Gaussians of several angular momenta then such scheme is called *general* contraction. Examples of basis sets that include general contractions include Atomic Natural Orbitals (ANO) sets. `LIBINT` was not designed to handle general contractions very well. You should use either split general contractions into segments for each angular momentum (it's done for correlation consistent basis sets) or use basis sets with segmented contractions only.

An integral of a two-electron operator $\hat{O}(\mathbf{r}_1, \mathbf{r}_2)$ over unnormalized primitive Cartesian Gaussians is written as

$$\int \phi(\mathbf{r}_1; \zeta_a, \mathbf{a}, \mathbf{A})\phi(\mathbf{r}_2; \zeta_c, \mathbf{c}, \mathbf{C})\hat{O}(\mathbf{r}_1, \mathbf{r}_2)\phi(\mathbf{r}_1; \zeta_b, \mathbf{b}, \mathbf{B})\phi(\mathbf{r}_2; \zeta_d, \mathbf{d}, \mathbf{D})d\mathbf{r}_1 d\mathbf{r}_2 \equiv (\mathbf{ab}|\hat{O}|\mathbf{cd}) \tag{6}$$

A set of integrals $\{(\mathbf{ab}|\hat{O}(\mathbf{r}_1, \mathbf{r}_2)|\mathbf{cd})\}$ over all possible combinations of functions $\mathbf{a} \in$ ShellA, $\mathbf{b} \in$ ShellB, etc. is termed a *shell*, or *quartet*, or *class* of integrals. For example, a $(ps|sd)$ class consists of $3 \times 1 \times 1 \times 6 = 18$ integrals.

The following definitions have been used throughout this work:

$$\zeta = \zeta_a + \zeta_b \tag{7}$$
$$\eta = \zeta_c + \zeta_d \tag{8}$$
$$\rho = \frac{\zeta\eta}{\zeta+\eta} \tag{9}$$
$$\mathbf{P} = \frac{\zeta_a\mathbf{A} + \zeta_b\mathbf{B}}{\zeta} \tag{10}$$
$$\mathbf{Q} = \frac{\zeta_c\mathbf{C} + \zeta_d\mathbf{D}}{\eta} \tag{11}$$
$$\mathbf{W} = \frac{\zeta\mathbf{P} + \eta\mathbf{Q}}{\zeta+\eta} \tag{12}$$

Incomplete gamma function is defined as

$$F_m(T) = \int_0^1 dt\ t^{2m}\ \exp(-Tt^2) \tag{13}$$

Evaluation of integrals over functions of non-zero angular momentum starts with the *auxiliary* integrals over primitive s-functions defined as

$$(\mathbf{00}|\mathbf{00})^{(m)} = 2F_m(\rho|\mathbf{PQ}|^2)\sqrt{\frac{\rho}{\pi}}S_{12}S_{34} \tag{14}$$

where $\mathbf{PQ} = \mathbf{P} - \mathbf{Q}$ and primitive overlaps S_{12} and S_{34} are computed as

$$S_{12} = \left(\frac{\pi}{\zeta}\right)^{3/2}\exp\left(-\frac{\zeta_a\zeta_b}{\zeta}|\mathbf{AB}|^2\right) \tag{15}$$

$$S_{34} = \left(\frac{\pi}{\eta}\right)^{3/2}\exp\left(-\frac{\zeta_c\zeta_d}{\eta}|\mathbf{CD}|^2\right) \tag{16}$$

In the evaluation of integrals over contracted functions it is convenient to use auxiliary integrals over primitives which include contraction and normalization factors of the target quartet $(\mathbf{ab}|\mathbf{cd})$:

$$(\mathbf{00}|\mathbf{00})^{(m)} = 2F_m(\rho|\mathbf{PQ}|^2)\sqrt{\frac{\rho}{\pi}}S_{12}S_{34}C_1C_2C_3C_4 \tag{17}$$

where the coefficients C_a, C_b, C_c, and C_d are normalization-including contraction coefficients (Eqs. (4) and (5)) for the first basis function out of each respective shell in the target quartet.

3 Overview of LIBINT's API

Prototypes for externaly accessible functions of LIBINT's components are contained in header files `libint.h libderiv.h` and `libr12.h`. Although LIBINT's machine generated source is written in C++, functions and data structures of the external interface are linked according to C convention, which simplifies its use in C and FORTRAN programs.

So let's look at header file `libint.h`. Inside the standard header wrappers, library static parameters are `defined`:

```
#define REALTYPE double
#define LIBINT_MAX_AM 8
#define LIBINT_OPT_AM 5
```

These parameters depend on how library was configured before compilation (see compilation manual). The first macro is the basic datatype for real numbers that LIBINT uses to compute integrals. It can be `double` or `long double`. With some compilers, e.g. IBM Visual Age C++, the latter datatype allows higher precision calculations. Macro `LIBINT_MAX_AM` specifies the maximum angular momentum + 1 of basis functions for which electron repulsion integrals can be computed. Hence in this example up to k functions ($L_{\max} = 7$) can be handled.

Before any component of LIBINT can be used some static data has to be initialized via a corresponding function call. That function for `libint` is

```
void init_libint_base();
```

After `init_libint_base()` has been called one has to initialize one or several corresponding integrals evaluator objects. Objects are "constructed' and "destructed" by calling the following functions

```
int  init_libint(Libint_t *, int max_am, int max_num_prim_comb);
void free_libint(Libint_t *);
```

The first argument to either function is the pointer to the object. Second and third arguments to `init_libint()` are the maximum angular momentum of the basis functions to be handled by this object and the maximum number of combinations of primitives per shell quartet that this object will handle. The latter quantity can be safely computed as a fourth power of the maximum number of primitives per shell in the basis set. `init_libint()` returns the number of `REALTYPE`-sized words of memory that was allocated for the object. The amount of memory depends heavily on `max_am` and somewhat on `max_num_prim_comb`. Memory tracking is not done by `LIBINT` internally and is left to the user's code. In order to compute how much memory an evaluator object will require one can call the following function:

```
int  libint_storage_required(int max_am, int max_num_prim_comb);
```

The return value is the number of `REALTYPE`-sized words of memory that a `Libint_t` object will require for the given values of `max_am` and `max_num_prim_comb`.

Note that integrals evaluator objects themselves are completely thread-safe and can be used in multiple thread environments. However, `init_libint_base()` is not reentrant, hence proper locking must be ensured. However, it needs to be called only once in the process, after that all threads can use `libint`.

After a `Libint_t` object has been initialized, we are ready to compute ERIs. In order to do that user must provide shell quartet data to the evaluator object and call an appropriate method to compute the integrals. `LIBINT`'s philosophy is to provide the leanest possible code. Thus it does not provide any functionality related to computing recurrence relation prerequisites, such as geometric quantities and incomplete gamma function values defined in the previous section. It is fully user's responsibility to compute the necessary data and feed it to the evaluator object. So let's look at the definition of `Libint_t`:

```
typedef struct {
  REALTYPE *int_stack;
  prim_data *PrimQuartet;
  REALTYPE AB[3];
  REALTYPE CD[3];
  REALTYPE *vrr_classes[15][15];
  REALTYPE *vrr_stack;
  } Libint_t;
```

The most important 3 members of the type are `PrimQuartet`, `AB`, and `CD`. All three of these members have to be set properly before a shell quartet can be computed. `PrimQuartet` is the array of data for each combination of primitives that contribute to this shell quartet. The datatype for `PrimQuartet` is described below. `AB` and `CD` store quantities **AB** and **CD** for this shell quartet. The rest of the data in `Libint_t` object is not meant for external use.

While `Libint_t.AB` and `Libint_.CD` are trivial to compute, the primitive quartet data is more involved. Let's look at definition of `prim_data`:

```
typedef struct pdata{
  REALTYPE F[29];
  REALTYPE U[6][3];
  REALTYPE twozeta_a;
  REALTYPE twozeta_b;
  REALTYPE twozeta_c;
  REALTYPE twozeta_d;
  REALTYPE oo2z;
  REALTYPE oo2n;
  REALTYPE oo2zn;
  REALTYPE poz;
  REALTYPE pon;
  REALTYPE oo2p;
  REALTYPE ss_r12_ss;
  } prim_data;
```

Let's look at what quantities each component of `prim_data` holds:

- `F` – values of auxiliary primitive integrals $(\mathbf{00}|\mathbf{00})^{(m)}$ (Eq. (17)) for $0 \le m \le \lambda(\mathbf{a}) + \lambda(\mathbf{b}) + \lambda(\mathbf{c}) + \lambda(\mathbf{d}) + C$, where $C = 0$ when computing ERIs, $C = 1$ when computing first derivative ERIs and integrals for linear R12 methods, and $C = 2$ when computing second derivative ERIs.
- `U` – geometric quantities **PA** (`U[0]`), **QC** (`U[2]`), **WP** (`U[4]`), and **WQ** (`U[5]`). If `libderiv` is being used then the following quatities are stored in `U[1]` and `U[3]`: **PB** and **QD**. If `libr12` is being used then the following quantities are stored in `U[1]` and `U[3]`: **QA** and **PC**.
- `twozeta_a` – $2\zeta_a$ (only used by `libderiv` and `libr12`)
- `twozeta_b` – $2\zeta_b$ (only used by `libderiv` and `libr12`)
- `twozeta_c` – $2\zeta_c$ (only used by `libderiv` and `libr12`)
- `twozeta_d` – $2\zeta_d$ (only used by `libderiv` and `libr12`)
- `oo2z` – $\frac{1}{2\zeta}$
- `oo2n` – $\frac{1}{2\eta}$

- `oo2zn` – $\frac{1}{2(\zeta+\eta)}$
- `poz` – $\frac{\rho}{\zeta}$
- `pon` – $\frac{\rho}{\eta}$
- `oo2p` – $\frac{1}{2\rho}$
- `ss_r12_ss` – $(\mathbf{00}|r_{12}|\mathbf{00}) = \frac{1}{\rho}(\mathbf{00}|\mathbf{00})^{(0)} + |\mathbf{PQ}|^2((\mathbf{00}|\mathbf{00})^{(0)} - (\mathbf{00}|\mathbf{00})^{(1)})$ (only used by `libr12`)

Most of these quantities are simple to evaluate. Evaluation of the incomplete gamma function `prim_data.F` is more involved. One should consult external sources for information on how to compute it efficiently.[4, 7]

Once the quartet data has been computed for every unique combination of primitives and put into `Libint_t.PrimQuartet`, ERIs can be computed. Appropriate functions are accessed via a four-dimensional array of pointers called `build_eri`:

```
extern REALTYPE *(*build_eri[8][8][8][8])(Libint_t *, int);
```

where the first argument is the integrals evaluator object, the second is the number of primitive quartet combinations that were stored in the previous step in `Libint_t.PrimQuartet`, and the array indices refer to the angular momenta of respective shells. For example, a function which evaluates a quartet of $(ps|ds)$ integrals is referred to as `build_eri[1][0][2][0](inteval1,num_prim_comb)`. The functions return pointer to the array that holds target integrals. The integrals are stored in "row major" order.[8] For example, if the number of functions in each shell is n_a, n_b, n_c, and n_d, respectively, then the integral $(ab|cd)$ is found at position $abcd = ((an_b + b)n_c + c)n_d + d$.

Note that currently `LIBINT` has a very important restriction on the angular momentum ordering of the functions in shell quartets that it can handle. `LIBINT` can evaluate a shell quartet $(\mathbf{ab}|\mathbf{cd})$ if $\lambda(\mathbf{a}) \geq \lambda(\mathbf{b})$, $\lambda(\mathbf{c}) \geq \lambda(\mathbf{d})$, and $\lambda(\mathbf{c}) + \lambda(\mathbf{d}) \geq \lambda(\mathbf{a}) + \lambda(\mathbf{b})$. If one needs to compute a quartet that doesn't conform the rule, e.g. of type $(pf|sd)$, permutational symmetry of integrals can be utilized to compute such quartet:[1]

$$(pq|rs) = (pq|sr) = (qp|rs) = (qp|sr) = (rs|pq) = (rs|qp) = (sr|pq) = (sr|qp) \tag{18}$$

In the case of $(pf|sd)$ shell quartet, one computes quartet $(ds|fp)$ instead, and then permutes function indices back to obtain the desired $(pf|sd)$.

The final integrals that `LIBINT` computes are not fully normalized yet. The reason is that the auxiliary integrals $(\mathbf{00}|\mathbf{00})^{(m)}$ include normalization factors of the first function of each shell. For example, in a $(ds|fp)$ quartet computed by `LIBINT` only integrals $(d_{xx}s|f_{xxx}p_x)$, $(d_{yy}s|f_{xxx}p_x)$, $(d_{xx}s|f_{yyy}p_x)$, etc., will be properly normalized. In order to compute integrals

[1] Note that some of the integrals that `libr12` computes possess different permutational symmetries than ERIs. One can still compute all desired integrals in that case.

in terms of functions which are all normalized to unity one has to multiply each integral by a "renormalization" prefactor:

$$(ab|cd) \equiv \frac{N(\zeta_a,\mathbf{a})N(\zeta_b,\mathbf{b})N(\zeta_c,\mathbf{c})N(\zeta_d,\mathbf{d})}{N(\zeta_a,\begin{pmatrix}\lambda(\mathbf{a})\\0\\0\end{pmatrix})N(\zeta_b,\begin{pmatrix}\lambda(\mathbf{b})\\0\\0\end{pmatrix})N(\zeta_c,\begin{pmatrix}\lambda(\mathbf{c})\\0\\0\end{pmatrix})N(\zeta_d,\begin{pmatrix}\lambda(\mathbf{d})\\0\\0\end{pmatrix})}(ab|cd) \quad (19)$$

3.1 Notes on using `libderiv`

Component `libderiv` is used to evaluate derivatives of ERIs with respect to basis function positions. Using `libderiv` is mostly similar to how `libint` is used. Here we only concentrate on significant differences which have not been noted before or on aspects of use specific to `libderiv`.

One quartet of ERIs (**ab**|**cd**) has total of 12 first derivatives

$$\frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial A_i}, \frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial B_i}, \frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial C_i}, \frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial D_i} : \quad i \in \{x,y,z\}$$

and $12*12 = 144$ second derivatives, although $12*13/2 = 78$ derivatives are unique because of permutation symmetry with respect to the order of taking the derivative:

$$\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial A_j}, \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_i \partial B_j}, \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial C_i \partial C_j}, \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial D_i \partial D_j} : \quad i \le j \in \{x,y,z\}$$
$$\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial B_j}, \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial C_j}, \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial D_j},$$
$$\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_i \partial C_j}, \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_i \partial D_j}, \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial C_i \partial D_j} : \quad i,j \in \{x,y,z\}$$

Translational invariance of ERIs can be used to eliminate any 3 of 12 first derivatives

$$\frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial B_i} = -\frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial A_i} - \frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial C_i} - \frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial D_i} \quad i \in \{x,y,z\} \quad (20)$$

and 33 of 78 second derivatives

$$\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial B_j} = -\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial A_j} - \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial C_j} - \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial D_j} \quad i,j \in \{x,y,z\} \tag{21}$$

$$\begin{aligned}\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_i \partial B_j} = {} & \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial A_j} + \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial C_j} + \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial D_j} \\ & \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial C_j} + \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial C_i \partial C_j} + \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial C_i \partial D_j} \\ & \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_j \partial D_i} + \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial C_j \partial D_i} + \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial D_i \partial D_j} \quad i \le j \in \{x,y,z\}\end{aligned} \tag{22}$$

$$\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_i \partial C_j} = -\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial C_j} - \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial C_i \partial C_j} - \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial C_j \partial D_i} \quad i,j \in \{x,y,z\} \tag{23}$$

$$\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_i \partial D_j} = -\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial D_j} - \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial C_i \partial D_j} - \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial D_i \partial D_j} \quad i,j \in \{x,y,z\} \tag{24}$$

$$\tag{25}$$

While `libint` computes one target quartet at a time, `libderiv` evaluates all of its possible unique derivatives. There are 2 types of "compute" functions in `libderiv` (see `libderiv.h`):

```
extern void (*build_deriv1_eri[5][5][5][5])(Libderiv_t *, int);
extern void (*build_deriv12_eri[4][4][4][4])(Libderiv_t *, int);
```

The former refers to functions which compute only first derivative ERIs, and the second refers to functions which compute both first and second derivative ERIs. The dimensions of each array are determined by the following 2 configure-time macros:

```
#define LIBDERIV_MAX_AM1 5
#define LIBDERIV_MAX_AM12 4
```

Note that "compute" functions in `libint`, `build_eri`, simply return a pointer to the target quartet, whereas `libderiv`'s functions return target data through integrals evaluator object, `Libderiv_t`. Such objects are initialized using one of the following functions:

```
int  init_libderiv1(Libderiv_t *, int max_am, int max_num_prim_quartets,
                    int max_cart_class_size);
int  init_libderiv12(Libderiv_t *, int max_am, int max_num_prim_quartets,
                     int max_cart_class_size);
```

These functions initialize objects for use with `build_deriv1_eri` and `build_deriv12_eri` compute functions, respectively. It is illegal to use a `Libderiv_t` object initialized by `init_libderiv1()` with `build_deriv12_eri` compute functions. Memory requirements for initializing these two types of objects are evaluated using

```
int  libderiv1_storage_required(int max_am, int max_num_prim_quartets,
                                int max_cart_class_size);
int  libderiv12_storage_required(int max_am, int max_num_prim_quartets,
                                 int max_cart_class_size);
```

Structure of `Libderiv_t` is essentially similar to `Libint_t`:

```
typedef struct {
  double *int_stack;
  prim_data *PrimQuartet;
  double *zero_stack;
  double *ABCD[12+144];
  double AB[3];
  double CD[3];
  double *deriv_classes[9][9][12];
  double *deriv2_classes[9][9][144];
  double *dvrr_classes[9][9];
  double *dvrr_stack;
  } Libderiv_t;
```

User passes quartet data to `libderiv` through `PrimQuartet`, `AB`, and `CD`. Data is returned through member `ABCD`. The dimension of `ABCD` is explicitly written as 12+144 which refer to the number of all (including nonunique) first and second derivatives of ERIs. If a derivative index runs For example, `ABCD[4]` and `ABCD[11]` point to derivative quartets $\frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial B_y}$ and $\frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial D_z}$, respectively. Similarly, `ABCD[13]` and `ABCD[27]` refer to $\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_x \partial A_y}$ and $\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_y \partial B_x}$, respectively.

Due to the translation invariance relations and the permutational symmetry of the second derivative integrals, some derivative quartets are not computed and thus only some elements of this array are initialized. Eqs. (20-24) specify how to evaluate elements which are not computed. Thus `build_deriv1_eri()` and `build_deriv12_eri()` functions produce 9 and $9 + 45 = 54$ unique derivative quartets, respectively. The unique quartets and corresponding

elements of `Libderiv_t.ABCD` are listed here:

$\frac{\partial(\mathbf{ab}\|\mathbf{cd})}{\partial A_x}$	0	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_y \partial A_y}$	25	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_x \partial D_x}$	93
$\frac{\partial(\mathbf{ab}\|\mathbf{cd})}{\partial A_y}$	1	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_y \partial A_z}$	26	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_x \partial D_y}$	94
$\frac{\partial(\mathbf{ab}\|\mathbf{cd})}{\partial A_z}$	2	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_y \partial C_x}$	30	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_x \partial D_z}$	95
$\frac{\partial(\mathbf{ab}\|\mathbf{cd})}{\partial C_x}$	6	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_y \partial C_y}$	31	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_y \partial C_y}$	103
$\frac{\partial(\mathbf{ab}\|\mathbf{cd})}{\partial C_y}$	7	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_y \partial C_z}$	32	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_y \partial C_z}$	104
$\frac{\partial(\mathbf{ab}\|\mathbf{cd})}{\partial C_z}$	8	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_y \partial D_x}$	33	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_y \partial D_x}$	105
$\frac{\partial(\mathbf{ab}\|\mathbf{cd})}{\partial D_x}$	9	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_y \partial D_y}$	34	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_y \partial D_y}$	106
$\frac{\partial(\mathbf{ab}\|\mathbf{cd})}{\partial D_y}$	10	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_y \partial D_z}$	35	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_y \partial D_z}$	107
$\frac{\partial(\mathbf{ab}\|\mathbf{cd})}{\partial D_z}$	11	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_z \partial A_z}$	38	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_z \partial C_z}$	116
$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_x \partial A_x}$	12	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_z \partial C_x}$	42	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_z \partial D_x}$	117
$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_x \partial A_y}$	13	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_z \partial C_y}$	43	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_z \partial D_y}$	118
$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_x \partial A_z}$	14	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_z \partial C_z}$	44	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_z \partial D_z}$	119
$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_x \partial C_x}$	18	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_z \partial D_x}$	45	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial D_x \partial D_x}$	129
$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_x \partial C_y}$	19	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_z \partial D_y}$	46	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial D_x \partial D_y}$	130
$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_x \partial C_z}$	20	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_z \partial D_z}$	47	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial D_x \partial D_z}$	131
$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_x \partial D_x}$	21	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_x \partial C_x}$	90	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial D_y \partial D_y}$	142
$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_x \partial D_y}$	22	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_x \partial C_y}$	91	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial D_y \partial D_z}$	143
$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial A_x \partial D_z}$	23	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial C_x \partial C_z}$	92	$\frac{\partial^2(\mathbf{ab}\|\mathbf{cd})}{\partial D_z \partial D_z}$	155

Each derivative quartet is identical in structure to a nondifferentiated quartet, i.e. individual integrals are arranged in a row major order. Normalization convention for the derivative integrals is the same as for the regular ERIs.

3.2 Notes on using `libr12`

Component `libr12` is used to evaluate integrals used in linear R12 theories[2, 3, 9, 10]. over operators $\frac{1}{r_{12}}$, r_{12}, $[r_{12}, \hat{T}_1]$, and $[r_{12}, \hat{T}_2]$. Using `libr12` is mostly similar to how `libint` is used. Here we only concentrate on significant differences which have not been noted before or on aspects of use specific to `libr12`.

There are two types of compute functions in `libr12` (see `libr12.h`):

```
extern void (*build_r12_gr[7][7][7][7])(Libr12_t *, int);
extern void (*build_r12_grt[7][7][7][7])(Libr12_t *, int);
```

The former computes integrals of operators $\frac{1}{r_{12}}$ ("g") and r_{12} ("r") only,[2] whereas in addition the latter computes also integrals of operators $[r_{12}, \hat{T}_1]$ and $[r_{12}, \hat{T}_2]$ ("t").[3] The size of each dimension of these function pointer arrays is determined by

```
#define LIBR12_MAX_AM 7
```

which corresponds to the maximum angular momentum of basis functions which `libr12` can handle, incremented by one.

Evaluator object type `Libr12_t` is defined as

```
#define NUM_TE_TYPES 4

typedef struct {
  REALTYPE *int_stack;
  prim_data *PrimQuartet;
  contr_data ShellQuartet;
  REALTYPE *te_ptr[NUM_TE_TYPES];
  REALTYPE *t1vrr_classes[13][13];
  REALTYPE *t2vrr_classes[13][13];
  REALTYPE *rvrr_classes[13][13];
  REALTYPE *gvrr_classes[14][14];
  REALTYPE *r12vrr_stack;

  } Libr12_t;
```

The usual array of data structures `PrimQuartet` is there along with a new data structure `ShellQuartet` for shell quartet data into which `Libint_t`'s **AB** and **CD** have migrated:

```
typedef struct {
  REALTYPE AB[3];
  REALTYPE CD[3];
  REALTYPE AC[3];
  REALTYPE ABdotAC, CDdotCA;
  } contr_data;
```

Members of the data structure correspond to the following quantities: **AB**, **CD**, **AC**, $\mathbf{AB} \cdot \mathbf{AC}$, and $\mathbf{CD} \cdot \mathbf{CA}$. Before computing a set of shell quartet, one initializes `PrimQuartet` with the primitive quartet data and `ShellQuartet` with the shell quartet data. Pointers to

[2] As of this writing, these functions have not been implemented yet.

[3] Note that in the literature the sum of reversed commutators is usually used, i.e. $[\hat{T}_1 + \hat{T}_2, r_{12}] = -[r_{12}, \hat{T}_1] - [r_{12}, \hat{T}_2]$.

computed shell quartets are returned in `Libr12_t.te_ptr`. `te_ptr[0]` refers to the quartet of ERIs, `te_ptr[1]` – to the quartet of integrals of the r_{12} operator, `te_ptr[2]` – to the quartet of integrals of the $[r_{12}, \hat{T}_1]$ operator, `te_ptr[3]` – to the quartet of integrals of the $[r_{12}, \hat{T}_2]$ operator. The integrals follow the aforementioned normalization convention of `LIBINT`.

One must remember that the commutator integrals have different permutational symmetry than ERIs and integrals of the r_{12} operator, namely:

$$\begin{aligned} (pq|[r_{12},\hat{T}_1]|rs) &= (pq|[r_{12},\hat{T}_1]|sr) = -(qp|[r_{12},\hat{T}_1]|rs) = -(qp|[r_{12},\hat{T}_1]|sr) = \\ = (rs|[r_{12},\hat{T}_2]|pq) &= (sr|[r_{12},\hat{T}_2]|pq) = -(rs|[r_{12},\hat{T}_2]|qp) = -(sr|[r_{12},\hat{T}_2]|qp) \end{aligned} \quad (26)$$

One must keep them in mind when computing such integrals with `libr12`because of the required preordering of shells in the shell quartet according to the canonical `LIBINT` order (see above). To obtain the desired integrals shells need to be reordered back, which is slightly more involved for the commutator integrals than for ERIs. Nevertheless, the reordering is always possible because integrals of both $[r_{12}, \hat{T}_1]$ and $[r_{12}, \hat{T}_2]$ operators are computed simultaneously.

4 Example: using `libint`

References

[1] J. T. Fermann and E. F. Valeev. Libint: Machine-generated library for efficient evaluation of molecular integrals over Gaussians, 2003. Freely available at `http://www.ccmst.gatech.edu/evaleev/libint/` or one of the authors.

[2] W. Kutzelnigg. r_{12}-dependent terms in the wave function as closed sums of partial wave amplitudes for large l. *Theor. Chim. Acta*, 68:445, 1985.

[3] W. Kutzelnigg and W. Klopper. Wave functions with terms linear in the interelectronic coordinates to take care of the correlation cusp. I. General theory. *J. Chem. Phys.*, 94:1985, 1991.

[4] S. Obara and A. Saika. Efficient recursive computations of molecular integrals over Cartesian Gaussian functions. *J. Chem. Phys.*, 84:3963, 1986.

[5] M. Head-Gordon and J. A. Pople. A method for 2-electron Gaussian integral and integral derivative evaluation using recurrence relations. *J. Chem. Phys.*, 89:5777, 1988.

[6] J. T. Fermann. *Efficient implementation of vertical recursion relations for the generation of electron repulsion intgerals.* PhD thesis, University of Georgia, 1996.

[7] P. M. W. Gill and J. A. Pople. The prism algorithm for 2-electron integrals. *Int. J. Quantum Chem.*, 40:753, 1991.

[8] D. E. Knuth. *The Art of Computer Programming.* Addison-Wesley, Reading, MA, third edition, 1997.

[9] W. Klopper and R. Röhse. Computation of some new two-electron Gaussian integrals. *Theor. Chim. Acta*, 83:441, 1992.

[10] E. F. Valeev and H. F. Schaefer. Evaluation of two-electron integrals for explicit r_{12} theories. *J. Chem. Phys.*, 113:3990, 2000.