`libreta` Manual

Version 0.1

Jun Zhang

Department of Chemistry

University of Illinois Urbana-Champaign

`http://www.zhjun-sci.com/software-libreta-download.php`

Contents

Chapter 1

Introduction to `libreta`

For those who want to start to use `libreta` immediately, please go to Section 3.1.

We know that users are very reluctant to read lengthy manuals thus we will keep this manual as short as possible, which will include the following contents:

1. Chapter 1: What is `libreta` and how to compile it.
2. Chapter 2: The notations and required background for using this library.
3. Chapter 3: The main part, how to use `libreta` in your own program.

1.1 What is `libreta`?

The central object of this manual, `libreta`, is an open source library for the evaluation of molecular integrals over contracted Gaussian basis set functions. With `libreta`, you can calculate the overlap, kinetic, multipole, one-electron Coulomb and two-electron Coulomb integrals as well as their first and second derivatives in a black-box and efficient way, thus you can set up your own quantum chemistry project quickly without writing these low-level subroutines by yourself.

As you may know, any quantum chemistry theories, say second-order many-body perturbation theory (MP2), involve the expression of the trail wave functions over some operators:

$$E_{\mathrm{MP2}} = \sum_{n}{}' \frac{\left\langle 0 \middle| \hat{V} \middle| n \right\rangle \left\langle n \middle| \hat{V} \middle| 0 \right\rangle}{E_0 - E_n} \tag{1.1.1}$$

However, this is only a formal theory. Only when we express it in terms of molecular integrals can we obtain a numerical value of this energy:

$$E_{\mathrm{MP2}} = \sum_{n}{}' \frac{\left\langle 0 \middle| \hat{V} \middle| n \right\rangle \left\langle n \middle| \hat{V} \middle| 0 \right\rangle}{E_0 - E_n} = \frac{1}{4} \sum_{abij} \frac{\langle ij||ab\rangle \langle ab||ij\rangle}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b} \tag{1.1.2}$$

Therefore, the evaluation of molecular integrals are a fundamental part of a quantum chemistry program. Without it no numerical results can be obtained. However, the algorithms and coding of molecular integrals are rather complex and time-consuming. The purpose of `libreta` is to release readers from these low-level works and focus on sophisticated works.

The advantages of `libreta`:

1. General. You can calculate the integrals for Cartesian or spherical Gaussians, segmented or general contracted basis sets, and arbitrary angular momentum (as long as your memory is sufficiently large).

2. Efficient. The evaluation codes have been highly optimized. Especially for one- and two-electron Coulomb integrals involving s, p, d, f, g, h basis functions.

3. Open source. You can modify the source codes to satisfy your own requirement.

4. Easy to use. The program is written in a black-box way. You do not need to know anything internal for calculating integrals.

5. Free. Perhaps the only thing you need to pay is to cite it.

1.2 How to Compile `libreta`?

1.2.1 Fast Compiling

For users who want to test the `libreta` quickly, we provide compiled static binaries for both Windows, Linux and Mac OS X platforms, which already exhibits good performance. However, to reach peak performance for your own project, say an *ab initio* quantum chemistry program, it is recommended that you compile it on your own machine.

On Linux or Mac OS X:

- Install `boost` library.

- Simply type `make` in the directory `libreta`.

On Windows:

1. Install `msys` and `mingw`, which provide a Linux-like system on your machine.

2. Add the commands to the `%PATH%` environment.

3. Install `boost` library.

4. Type `make` in the directory `libreta`.

After compiling, you will find the library in `libreta/bin/libreta.a` and header files in `libreta/bin/include/`.

1.2.2 Optimization

You can optimize the binary of `libreta` by modification of `libreta/Makefile`.

- `libreta` has been tested extensively with g++. Using other compilers, like Intel C or Microsoft Visual C++ compiler, *may* increase the efficiency. However, whether the efficiency can be improved depends on specific compilers and machine. Please do calibrations to decide which compiler you should use.

- In `libreta/Makefile`, `CXXFLAG` suggests the compiling options. Essential values are `-O3 -Os`. According to your machine, it is recommended to add `-msse4` and/or `-mavx`. Most CPUs released after 2011 support these options.

1.3 Inside the Black-box

`libreta` is a black-box library. For interested readers, please refer to the following monograph and papers for understanding "inside the box":

- Helgaker, T.; Jørgensen, J.; Olsen, J. *Molecular Electron-Structure Theory.* John Wiley & Sons Ltd., 2000.
- Zhang, J. LIBRETA: Computerized Optimization and Code Synthesis for Electron Repulsion Integral Evaluation. *J. Chem. Theory Comput.* **2018**, DOI: 10.1021/acs.jctc.7b00788.

1.4 Contact and Citation

For any bug reports, suggestions, questions or special requirement, please feel free to contact the author, Dr. Jun Zhang:

`mailto: zhjun@illinois.edu`

If you feel that `libreta` has helped you in your research, I will appreciate you cite this:

- Zhang, J. LIBRETA: Computerized Optimization and Code Synthesis for Electron Repulsion Integral Evaluation. *J. Chem. Theory Comput.* **2018**, DOI: 10.1021/acs.jctc.7b00788.

Chapter 2

Notations

2.1 Gaussian Basis Set

2.1.1 Contraction

The most popular basis function in quantum chemistry is the so-called Gaussian-type orbital (GTO). A *primitive Cartesian GTO* is defined as:

$$\chi^{\text{Prim}}(\mathbf{r}) \equiv N_{\mathbf{a}} (x - A_x)^{a_x} (y - A_y)^{a_y} (z - A_z)^{a_z} \exp\left(-\alpha (\mathbf{r} - \mathbf{A})^2\right) \tag{2.1.1}$$

Here, $\mathbf{A} \equiv \{A_x, A_y, A_z\}$ is the *center*, $\alpha > 0$ is the *exponent*, $N_{\mathbf{a}}$ is a normalization constant. The *angular momentum components* a_x, a_y, a_z are nonnegative integers and the sum of them is the *angular momentum*: $L_a = a_x + a_y + a_z$.

In practice, it is the *contracted Gaussian* that used as a basis function:

$$\chi^{\text{CRT}}(\mathbf{r}) \equiv (x - A_x)^{a_x} (y - A_y)^{a_y} (z - A_z)^{a_z} \sum_{\mu=1}^{K} D_\mu N_{\mathbf{a}\mu} \exp\left(-\alpha_\mu (\mathbf{r} - \mathbf{A})^2\right) \tag{2.1.2}$$

where K is called *contracted degree* and D_μ's are called contracted coefficients. From now on all basis functions we refer to mean contracted Gaussians. Of course, a primitive Gaussian can be viewed as a contracted one with $K = 1$.

There are two contraction patterns: *segmented* and *general* contraction. In segmented contraction, each primitive GTO is contracted only once; in general contraction, a primitive GTO can contribute to several basis functions. We refer the B basis functions contracted from the same set of K primitive ones as a *block*.

For example, there are two p GTOs here for the same atom (in Gaussian94 basis set format):

```
P   7   1.00
    315.9000000    0.392656E-02
     74.4200000    0.298811E-01
     23.4800000    0.127212E+00
      8.4880000    0.320943E+00
      3.2170000    0.455429E+00
      1.2290000    0.268563E+00
      0.2964000    0.188336E-01
P   7   1.00
    315.9000000   -0.858302E-03
     74.4200000   -0.630328E-02
     23.4800000   -0.288255E-01
      8.4880000   -0.694560E-01
      3.2170000   -0.119493E+00
      1.2290000   -0.199581E-01
      0.2964000    0.510268E+00
```

Here, the 2 GTOs are contracted from 7 primitive ones with different contraction coefficients. Thus they form a block with $L = 1$, $K = 7$, $B = 2$.

2.1.2 Angular Momentum

The angular momentum may be the most important character of a Gaussian. Given an L, a GTO can be of either *Cartesian* or *spherical* form.

The entire set of GTOs having different combinations of L_x, L_y, L_z satisfying $L_x + L_y + L_z = L$ is call a *Cartesian L-shell*. It can be shown that the number of GTOs in a Cartesian L-shell is:

$$N_L^{\mathrm{CRT}} = \frac{(L+1)(L+2)}{2} \tag{2.1.3}$$

In `libreta`, the order of Cartesian angular momenta is *lexical ordering*. For example, for $L = 2$, we have:

0. $L_x = 2, L_y = 0, L_z = 0$
1. $L_x = 1, L_y = 1, L_z = 0$
2. $L_x = 1, L_y = 0, L_z = 1$
3. $L_x = 0, L_y = 2, L_z = 0$
4. $L_x = 0, L_y = 1, L_z = 1$
5. $L_x = 0, L_y = 0, L_z = 2$

Note: here we follow the C++ convention that an array starts from 0.

A spherical GTO is a linear combination of several CRT GTOs:

$$\chi^{\mathrm{Sph}}(\mathbf{r}) \equiv S_{LM}(\mathbf{r}) \sum_{\mu=1}^{K} D_\mu N_{\mathbf{a}\mu} \exp\left(-\alpha_\mu (\mathbf{r} - \mathbf{A})^2\right) \tag{2.1.4}$$

It has only one component: M, ranging from $-L$ to $+L$. Thus the number of Gaussians in a *spherical L-shell* is:

$$N_L^{\mathrm{Sph}} = 2L + 1 \tag{2.1.5}$$

In `libreta`, the order of spherical angular momenta is *decending*. For example, for $L = 2$, we have:

0. $M = +2$
1. $M = +1$
2. $M = 0$
3. $M = -1$
4. $M = -2$

The definitions of S_{LM} in different quantum chemistry programs are slightly different, but they all satisfy the Laplace's equation:

$$\nabla^2 S_{LM}(\mathbf{r}) = 0 \tag{2.1.6}$$

If your own program uses a special form of spherical GTO, you can use `libreta` to calculate the integrals in Cartesian form, then use your own codes to transform them into your own spherical form.

The explicit expression of S_{LM} used in `libreta` is:

$$S_{LM}(\mathbf{r}) = N_{LM} \sum_{t=0}^{\lfloor (L-|M|)/2 \rfloor} \sum_{u=0}^{t} \sum_{v=v_m}^{\lfloor |m|/2-v_m \rfloor + v_m} C_{tuv}^{LM} M_{tuv}^{LM} x^{2t+|M|-2(u+v)} y^{2(u+v)} z^{L-2t-|M|} \tag{2.1.7}$$

$$N_{LM} = \frac{1}{2^{|M|} L!} \sqrt{\frac{(L+|M|)!(L-|M|)!}{2^{\delta_{m0}-1}}} \tag{2.1.8}$$

$$C_{tuv}^{LM} = (-1)^{t+v-v_m} \left(\frac{1}{4}\right)^t \binom{L}{t} \binom{L-t}{|M|+t} \binom{t}{u} \binom{|M|}{2v} \tag{2.1.9}$$

$$M_{tuv}^{LM} = \sqrt{\frac{(2(2t+|M|-2(u+v))-1)!!(4(u+v)-1)!!(2(L-2t-|M|)-1)!!}{(2L-1)!!}} \tag{2.1.10}$$

$$v_m = \begin{cases} 1/2 & m < 0 \\ 0 & m \geq 0 \end{cases} \tag{2.1.11}$$

Although (2.1.7) looks formidable, the coefficients form a very sparse matrix and the numeric values are very small. Some examples of S_{LM} can be found in Table 2.1.1. You can convince yourself that the polynomials in Table 2.1.1 can be obtained from (2.1.7).

Table 2.1.1: S_{LM} ($L = 0$ to 3) defined in `libreta`.

M	0	1	2	3
3				$\sqrt{\frac{5}{8}}x^3 - \sqrt{\frac{9}{8}}xy^2$
2			$\frac{\sqrt{3}}{2}(x^2 - y^2)$	$\frac{\sqrt{3}}{2}(x^2z - y^2z)$
1		x	xz	$-\sqrt{\frac{3}{8}}x^3 - \sqrt{\frac{3}{40}}xy^2 + \sqrt{\frac{6}{5}}xz^2$
0	1	z	$-\frac{1}{2}x^2 - \frac{1}{2}y^2 + z^2$	$-\sqrt{\frac{9}{20}}x^2z - \sqrt{\frac{9}{20}}y^2z + z^3$
-1		y	yz	$-\sqrt{\frac{3}{40}}x^2y - \sqrt{\frac{3}{8}}y^3 + \sqrt{\frac{6}{5}}yz^2$
-2			xy	xyz
-3				$\sqrt{\frac{9}{8}}x^2y - \sqrt{\frac{5}{8}}y^3$

2.2 Supported Molecular Integrals

In this Section, we will introduce the molecular integrals supported in `libreta`. In future versions of `libreta` more types will be added. We introduce four Cartesian contracted Gaussians: $\chi_a(\mathbf{r}), \chi_b(\mathbf{r}), \chi_c(\mathbf{r})$ and $\chi_d(\mathbf{r})$ (We drop the "CRT" superscripts).

2.2.1 "Zero"-electron Integrals

Actually a "Zero"-electron integral is not an integral, it operates on a single Gaussian.

Gaussian value of $\chi_a(\mathbf{r})$ at a point $\mathbf{P}$ is defined as:

$$\chi_a(\mathbf{P}) \tag{2.2.1}$$

2.2.2 One-electron Integrals

Many types of one-electron integrals exists in quantum chemistry community and the following are the most typical ones.

Overlap integral of $\chi_a(\mathbf{r})$ and $\chi_b(\mathbf{r})$:

$$S_{ab} = \int \chi_a(\mathbf{r})\chi_b(\mathbf{r})\mathrm{d}\mathbf{r} \tag{2.2.2}$$

Differential integral of $\chi_a(\mathbf{r})$ and $\chi_b(\mathbf{r})$ of rank $d = d_x + d_y + d_z$ $(d_x, d_y, d_z \geq 0)$:

$$D_{ab}(d_x, d_y, d_z) = \int \chi_a(\mathbf{r}) \left(\frac{\partial}{\partial x}\right)^{d_x} \left(\frac{\partial}{\partial y}\right)^{d_y} \left(\frac{\partial}{\partial z}\right)^{d_z} \chi_b(\mathbf{r})\mathrm{d}\mathbf{r} \tag{2.2.3}$$

Multipole integral of $\chi_a(\mathbf{r})$ and $\chi_b(\mathbf{r})$ on center $\mathbf{C}$ of rank $m = m_x + m_y + m_z$ $(m_x, m_y, m_z \geq 0)$:

$$M_{ab}(\mathbf{C}, m_x, m_y, m_z) = \int \chi_a(\mathbf{r})(x - C_x)^{m_x}(y - C_y)^{m_y}(z - C_z)^{m_z}\chi_b(\mathbf{r})\mathrm{d}\mathbf{r} \tag{2.2.4}$$

Note that some integrals does not appear directly in `libreta` but can still be obtained. For example, the kinetic integral can be obtained from differential integrals:

$$\int \chi_a(\mathbf{r}) \left(-\frac{1}{2}\nabla^2\right) \chi_b(\mathbf{r})\mathrm{d}\mathbf{r} = -\frac{1}{2}\left(D_{ab}(2,0,0) + D_{ab}(0,2,0) + D_{ab}(0,0,2)\right) \tag{2.2.5}$$

One-electron Coulomb integral of $\chi_a(\mathbf{r})$ and $\chi_b(\mathbf{r})$ on center $\mathbf{C}$:

$$V_{ab}(\mathbf{C}) = \int \chi_a(\mathbf{r}) \sum_C \frac{Q_C}{r_C}\chi_b(\mathbf{r})\mathrm{d}\mathbf{r} \tag{2.2.6}$$

2.2.3 Two-electron Integrals

These are the most expensive parts in almost every quantum chemistry program. We have and still been doing our best to optimize this part.

Two-electron Coulomb integral of $\chi_a(\mathbf{r})$, $\chi_b(\mathbf{r})$, $\chi_c(\mathbf{r})$ and $\chi_d(\mathbf{r})$:

$$(ab|cd) = \int \chi_a(\mathbf{r}_1)\chi_b(\mathbf{r}_1)\frac{1}{r_{12}}\chi_c(\mathbf{r}_2)\chi_d(\mathbf{r}_2)\mathrm{d}\mathbf{r}_1\mathrm{d}\mathbf{r}_2 \tag{2.2.7}$$

2.2.4 Cartesian to Spherical Transformations

This transformation is performed according to (2.1.7). `libreta` provides the functions for transformation of one-, two- and four-index quantities.

2.2.5 Other Functions

`libreta` also contains some low-level functions for readers.

Boys function:

$$F_n(x) = \int t^{2n}\exp(-xt^2)\mathrm{d}t \tag{2.2.8}$$

Roots and weights of Gaussian quadrature for *Rys polynomial* $R_n^\alpha(x)$, which is defined as:

$$\int R_m^\alpha(x)R_n^\alpha(x)\exp(-\alpha x^2)\mathrm{d}x = \delta_{mn} \tag{2.2.9}$$

Chapter 3

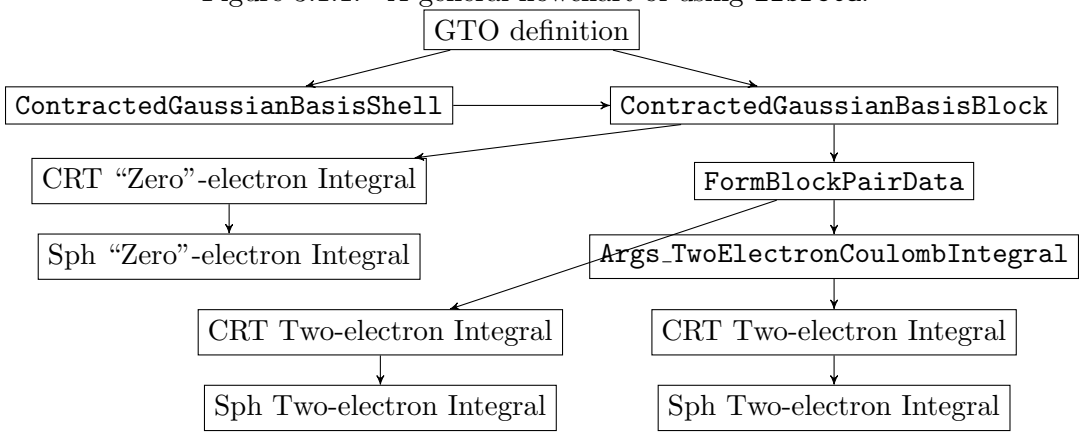
Use `libreta` in Your Program

If you do not want to read the verbose tutorial, you can read the general remarks below and then refer to read test files in `libreta/bin/test/*.cpp` and header files in `libreta/bin/include/*.h`.

3.1 Introduction

In this Chapter, we will discuss how to use `libreta` in your program with examples. Only the essential parts in `libreta` are discussed here. For all aspects, please read the header files (`libreta/bin/include/*.h`), where all the interfaces are available. If you want to study or modify the internal algorithms, you will have to read the C++ source codes (`libreta/src/*.cpp`). A general flowchart of using `libreta` is shown in Figure 3.1.1.

Figure 3.1.1: A general flowchart of using `libreta`.



All the functions in `libreta` calculate integrals in Cartesian GTO basis, with the following structure:

```
XXIntegral(args, IntegralBuffer, Results)

XXIntegralDerivFirst(args, IntegralBuffer, Results, dResults)

XXIntegralDerivSecond(args, IntegralBuffer, Results, dResults,
                      d2Results)
```

where:

- `args` are parameters for the integral (for example, GTO information);

- `IntegralBuffer` is a buffer for the function. Its minimal size can be obtained from some functions in `libreta`. Note that this buffer can be allocated at the beginning of your program and used for all integral calculations, and deallocated before your program terminates.

- `Results` stores the calculated integrals. They can be transformed to spherical GTO basis.

- `dResults` stores the first derivatives of the integrals.

- `d2Results` stores the second derivatives of the integrals.

Given four Cartesian GTO blocks a, b, c, and d, `libreta` will calculate *all* possible integrals simultaneouly. The order of the integrals must be remembered carefully.

Angular Momentum. For one-electron integrals $(a|\hat{O}|b)$, `libreta` calculates it as if $L_a \geq L_b$. For example, for $(p|\hat{O}|d)$, `libreta` will change it to $(d|\hat{O}|p)$. For two-electron integrals $(ab|cd)$, `libreta` calculates it as if $L_a \geq L_b$, $L_c \geq L_d$ and $L_a(L_a+1)/2 + L_b \geq L_c(L_c+1)/2 + L_d$. For example, for $(ds|pf)$, `libreta` will change it to $(fp|ds)$. This rearrangement is carried out by `libreta` automatically and users do not need to do this explicitly. BUT REMEMBER THAT ALL THE RESULTS ARE ORDERED ACCORDING TO THE REORDERED INTEGRALS!

General Contraction. The integrals are right-major ordered, block-by-block.

Shell Order. In each block-pair, the integrals are right-major ordered according to their angular momentum component orders.

Derivative Order. According to the translational invariance of the integrals, for integrals involving 2 (differential integrals), 3 (multipole and one-electron Coulomb integrals) and 4 (two-electron integrals) centers, we need only to calculate derivatives with respect to 1, 2, and 3 centers. The derivatives are right-major ordered according to the order $A_x, A_y, A_z, B_x, B_y, B_z, C_x, C_y, C_z$.

The first derivatevs are ordered as:

$$\frac{\partial}{\partial A_x}, \frac{\partial}{\partial A_y}, \frac{\partial}{\partial A_z}, \frac{\partial}{\partial B_x}, \frac{\partial}{\partial B_y}, \frac{\partial}{\partial B_z}, \frac{\partial}{\partial C_x}, \frac{\partial}{\partial C_y}, \frac{\partial}{\partial C_z} \tag{3.1.1}$$

and the second derivatives are ordered as:

$$\frac{\partial^2}{\partial A_x \partial A_x}, \frac{\partial^2}{\partial A_x \partial A_y}, \frac{\partial^2}{\partial A_x \partial A_z}, \frac{\partial^2}{\partial A_x \partial B_x}, \frac{\partial^2}{\partial A_x \partial B_y}, \frac{\partial^2}{\partial A_x \partial B_z}, \frac{\partial^2}{\partial A_x \partial C_x}, \frac{\partial^2}{\partial A_x \partial C_y}, \frac{\partial^2}{\partial A_x \partial C_z},$$
$$\frac{\partial^2}{\partial A_y \partial A_y}, \frac{\partial^2}{\partial A_y \partial A_z}, \frac{\partial^2}{\partial A_y \partial B_x}, \frac{\partial^2}{\partial A_y \partial B_y}, \frac{\partial^2}{\partial A_y \partial B_z}, \frac{\partial^2}{\partial A_y \partial C_x}, \frac{\partial^2}{\partial A_y \partial C_y}, \frac{\partial^2}{\partial A_y \partial C_z},$$
$$\frac{\partial^2}{\partial A_z \partial A_z}, \frac{\partial^2}{\partial A_z \partial B_x}, \frac{\partial^2}{\partial A_z \partial B_y}, \frac{\partial^2}{\partial A_z \partial B_z}, \frac{\partial^2}{\partial A_z \partial C_x}, \frac{\partial^2}{\partial A_z \partial C_y}, \frac{\partial^2}{\partial A_z \partial C_z},$$
$$\frac{\partial^2}{\partial B_x \partial B_x}, \frac{\partial^2}{\partial B_x \partial B_y}, \frac{\partial^2}{\partial B_x \partial B_z}, \frac{\partial^2}{\partial B_x \partial C_x}, \frac{\partial^2}{\partial B_x \partial C_y}, \frac{\partial^2}{\partial B_x \partial C_z},$$
$$\frac{\partial^2}{\partial B_y \partial B_y}, \frac{\partial^2}{\partial B_y \partial B_z}, \frac{\partial^2}{\partial B_y \partial C_x}, \frac{\partial^2}{\partial B_y \partial C_y}, \frac{\partial^2}{\partial B_y \partial C_z},$$
$$\frac{\partial^2}{\partial B_z \partial B_z}, \frac{\partial^2}{\partial B_z \partial C_x}, \frac{\partial^2}{\partial B_z \partial C_y}, \frac{\partial^2}{\partial B_z \partial C_z},$$
$$\frac{\partial^2}{\partial C_x \partial C_x}, \frac{\partial^2}{\partial C_x \partial C_y}, \frac{\partial^2}{\partial C_x \partial C_z},$$
$$\frac{\partial^2}{\partial C_y \partial C_y}, \frac{\partial^2}{\partial C_y \partial C_z},$$
$$\frac{\partial^2}{\partial C_z \partial C_z} \tag{3.1.2}$$

Other derivatives can be calculated according to translational invariance like

$$\left(\frac{\partial}{\partial A_x} + \frac{\partial}{\partial B_x} + \frac{\partial}{\partial C_x} + \frac{\partial}{\partial D_x}\right)(\mathbf{ab}|\mathbf{cd}) = 0 \tag{3.1.3}$$

For example,

$$\frac{\partial^2}{\partial A_x \partial D_y}(\mathbf{ab}|\mathbf{cd}) = -\left(\frac{\partial^2}{\partial A_x \partial A_y} + \frac{\partial^2}{\partial A_x \partial B_y} + \frac{\partial^2}{\partial A_x \partial C_y}\right)(\mathbf{ab}|\mathbf{cd}) \tag{3.1.4}$$

$$\frac{\partial^2}{\partial D_x^2}(\mathbf{ab}|\mathbf{cd}) = -\left(\frac{\partial^2}{\partial A_x \partial D_x} + \frac{\partial^2}{\partial B_x \partial D_x} + \frac{\partial^2}{\partial C_x \partial D_x}\right)(\mathbf{ab}|\mathbf{cd}) \tag{3.1.5}$$

3.2 Define a GTO Block

Related test files:

- `libreta/bin/test/gtos.cpp`

Related header files:

- `libreta/bin/include/Vec3D.h`
- `libreta/bin/include/ContractedGaussianBasisShell.h`
- `libreta/bin/include/ContractedGaussianBasisBlock.h`

3.2.1 Cartesian GTO Shell

The first step to use `libreta` is to define a Cartesian GTO shell. The basis set information can be obtained from, say, the EMSL Basis Set Library:

`https://bse.pnl.gov/bse/portal`

`libreta` can define a GTO simply by parsing a `std::string` of GTO in standard Gaussian format. For example, a d GTO:

```
D 3 1.0
24.0400000   0.0069760
 7.5710000   0.0316690
 2.7320000   0.1040060
```

For the first line:

1. `D` The angular momentum. For $L \leq 6$, you can use either integers `0,1,2,3,4,5,6` or letters `s,p,d,f,g,h,i` (case-insensitive). For $L \geq 7$, you can only use integers.
2. `3` The contraction degree.
3. `1.0` Scaling factor. However, it is NOT used in the current version of `libreta`.

For the following lines,

- `101.8000000` and `0.0008910` is the exponent and contraction coefficient, respectively. Both can also be written in an exponential form. For example, `0.0008910` can also be written as `8.91E-004` or `8.91e-004`. However, for some reasons, `8.91D-004` or `8.91d-004` are not recognizable in the current version.

The center of a Cartesian GTO shell is given by the `Vec3D` class. It is very easy to use and almost self-explained. Please refer to `libreta/bin/include/Vec3D.h`.

Now, we can define such a GTO shell by `ContractedGaussianBasisShell`:

```
void ContractedGaussianBasisShell::ContractedGaussianBasisShell(
    const string& basisinfo,
    const Vec3D& tcenter);
```

- `basisinfo` A `std::string` of the basis set information in Gaussian format.
- `tcenter` A `Vec3D` object of the center.

```
#include "ContractedGaussianBasisShell.h"
#include "CartesianAngularMomentum.h"
#include "Vec3D.h"
#include <string>
#include <cstdio>

using namespace libreta;
using namespace std;

int main(int argc, char** argv)
{
    const Vec3D v(2, 0, 0);
    const string basisinfo("D␣3␣1.0\n24.0400000␣␣0.0069760\n7.5710000␣␣0.0316690\n2.7320000␣␣0.1040060\n");

    // Construct a GTO shell.
    const ContractedGaussianBasisShell cgbs(basisinfo, v);

    // Get and output information of a GTO shell.
    const Vec3D& center(cgbs.GetCenter());
    const vector<double>& exponents(cgbs.GetExponents());
    const vector<double>& coefficients(cgbs.GetCoefficients());
    const vector<double>& mLfactors(cgbs.GetmLFactors());
    printf("GTO␣shell␣information:\n");
    printf("Center:␣%10.8f␣%10.8f␣%10.8f\n", center.x, center.y, center.z);
    printf("Angular␣momentum:␣␣␣%d\n", cgbs.GetL());
    printf("Contraction␣degree:␣%d\n", cgbs.GetK());
    printf("%20s␣␣%20s\n", "alpha", "coefficients");
    for(int k = 0; k < cgbs.GetK(); ++k)
    {
        printf("%20.8f␣␣%20.8f\n", exponents[k], coefficients[k]);
    }
```

```
    CartesianAngularMomentum crtL(cgbs.GetL());
    int nCRTShellL = crtL.GetNLLz();
    printf("mL factors:\n");
    for(int mL = 0; mL < nCRTShellL; ++mL, ++crtL)
    {
        printf("(%s)  %20.8f\n", crtL.tostr().c_str(), mLfactors[mL]);
    }

    return 0;
}
```

Now, `cgbs` is the corresponding Cartesian GTO shell. The GTO will be automatically normalized. Compile this file and run, you will get the following output:

```
GTO shell information:
Center: 2.00000000 0.00000000 0.00000000
Angular momentum:   2
Contraction degree: 3
              alpha          coefficients
         24.04000000           40.24718864
          7.57100000           24.19058090
          2.73200000           13.34729861
mL factors:
(  2,   0,   0)            0.57735027
(  1,   1,   0)            1.00000000
(  1,   0,   1)            1.00000000
(  0,   2,   0)            0.57735027
(  0,   1,   1)            1.00000000
(  0,   0,   2)            0.57735027
```

Now we explain the functions and output:

1. `cgbs.GetL()` An integer. Get the angular momentum.
2. `cgbs.GetK()` An integer. Get the contraction degree.
3. `cgbs.GetCenter()` A constant `Vec3D` reference. Get the center.
4. `cgbs.GetExponents()`. A constant `std::vector` reference. Size: K. Get the exponents.
5. `cgbs.GetCoefficients()`. A constant `std::vector` reference. Size: K. Get the contraction coefficients. However, they are not the original coefficients but scaled by normalization. For $L \geq 2$, they still require a scaling factor, which is given in `mLfactors`.
6. `cgbs.GetmLFactors()`. A constant `std::vector` reference. Size: $(L+1)(L+2)/2$. Get the geometrical normalization factors. The GTOs of different angular momentum components require an additional factor to be normalized. For s and p, these factors are always 1. For $L \geq 2$, the factors differ for each angular momentum and are lexically ordered.

Explicitly, `cgbs` represents 6 GTOs in the following order:

$$0.57735027 \times (x - 2.00000000)^2 F \tag{3.2.1}$$

$$1.00000000 \times (x - 2.00000000)(y - 0.00000000) F \tag{3.2.2}$$

$$1.00000000 \times (x - 2.00000000)(z - 0.00000000) F \tag{3.2.3}$$

$$0.57735027 \times (y - 0.00000000)^2 F \tag{3.2.4}$$

$$1.00000000 \times (y - 0.00000000)(z - 0.00000000) F \tag{3.2.5}$$

$$0.57735027 \times (z - 0.00000000)^2 F \tag{3.2.6}$$

where

$$\begin{aligned} F =& 40.24718864 \exp(-24.04000000 r_A^2) + 24.19058090 \exp(-7.57100000 r_A^2) \\ &+ 13.34729861 \exp(-2.73200000 r_A^2)) \end{aligned} \tag{3.2.7}$$

$$r_A^2 = (x - 2.00000000)^2 + (y - 0.00000000)^2 + (z - 0.00000000)^2 \tag{3.2.8}$$

3.2.2 Cartesian GTO Block: Segmented Contraction

To perform the evaluation of molecular integrals, you need to transform `ContractedGaussianBasisShell` to `ContractedGaussianBasisBlock`.

```
void ContractedGaussianBasisBlock::ContractedGaussianBasisBlock(
    const Vec3D& tcenter,
    int tL,
    int tK,
    int tB,
    const vector<double>& texponents,
    const vector<double>& tcoefficients,
    const vector<double>& tmLfactors);
```

- `tcenter` A `Vec3D` object of the center.
- `tL` The angular momentum.
- `tK` The contraction degree.
- `tB` The block size.
- `texponents` A `std::vector` containing K exponents.
- `tcoefficients` A `std::vector` containing KB scaled contraction coefficients.
- `tmLfactors` A `std::vector` containing $(L+1)(L+2)/2$ geometrical normalization factors.

For segmented basis functions, there is only one step:

```
#include "ContractedGaussianBasisBlock.h"
#include "ContractedGaussianBasisShell.h"
#include "CartesianAngularMomentum.h"
#include "Vec3D.h"
#include <string>
#include <cstdio>

using namespace libreta;
using namespace std;

int main(int argc, char** argv)
{
    const Vec3D v(2, 0, 0);
    const string basisinfo("D␣3␣1.0\n24.0400000␣␣0.0069760\n7.5710000␣␣0.0316690\n2.7320000␣␣0.1040060\n");
    // Construct a GTO shell.
        const ContractedGaussianBasisShell cgbs(basisinfo, v);
    const Vec3D& tcenter(cgbs.GetCenter());
    const int tL = cgbs.GetL();
    const int tK = cgbs.GetK();
    const int tB = 1;
    const vector<double>& texponents(cgbs.GetExponents());
    const vector<double>& tcoefficients(cgbs.GetCoefficients());
    const vector<double>& tmLfactors(cgbs.GetmLFactors());
    // Construct a GTO block.
    const ContractedGaussianBasisBlock block(tcenter, tL, tK, tB, texponents, tcoefficients, tmLfactors);

    // Get and output information of a GTO block.
    const Vec3D& center(block.GetCenter());
    const vector<double>& exponents(block.GetExponents());
    const vector<double>& coefficients(block.GetCoefficients());
    const vector<double>& mLfactors(block.GetmLFactors());
    printf("GTO␣block␣information:\n");
    printf("Center:␣%10.8f␣%10.8f␣%10.8f\n", center.x, center.y, center.z);
    printf("Angular␣momentum:␣␣␣%d\n", block.GetL());
    printf("Contraction␣degree:␣%d\n", block.GetK());
    printf("Block␣size:␣␣␣␣␣␣␣␣%d\n", block.GetB());
    printf("%20s␣␣%20s\n", "alpha", "coefficients");
    for(int k = 0; k < block.GetK(); ++k)
    {
        printf("%20.8f␣␣%20.8f\n", exponents[k], coefficients[k]);
    }
    CartesianAngularMomentum crtL(block.GetL());
    int nCRTShellL = crtL.GetNLLz();
    printf("mL␣factors:\n");
    for(int mL = 0; mL < nCRTShellL; ++mL, ++crtL)
    {
        printf("(%s)␣␣%20.8f\n", crtL.tostr().c_str(), mLfactors[mL]);
    }

    return 0;
}
```

Now, `block` is the GTO block that can be used in the following calculations.

3.2.3 Cartesian GTO Block: General Contraction

For general contracted GTOs, we can use similar approaches. For example, for the following two GTOs:

```
P 3 1.0
 24.0400000    +0.0069760
  7.5710000    +0.0316690
  2.7320000    +0.1040060
P 3 1.0
 24.0400000    +0.0069760
  7.5710000    +0.0316690
  2.7320000    -0.1040060
```

in this case, all GTOs have $K = 3$ and $B = 2$. We just need to store all their contracted coefficents sequentially for `ContractedGaussianBasisBlock`. See the following example:

```
#include "ContractedGaussianBasisBlock.h"
#include "ContractedGaussianBasisShell.h"
#include "CartesianAngularMomentum.h"
#include "Vec3D.h"
#include <string>
#include <cstdio>

using namespace libreta;
using namespace std;

int main(int argc, char** argv)
{
    const Vec3D v(2, 0, 0);
    const string basisinfo1("D 3 1.0\n24.0400000  0.0069760\n7.5710000  0.0316690\n2.7320000  0.1040060\n");
    const string basisinfo2("D 3 1.0\n24.0400000  0.0069760\n7.5710000  0.0316690\n2.7320000 -0.1040060\n");
    // Construct two GTO shells.
    const ContractedGaussianBasisShell cgbs1(basisinfo1, v);
    const ContractedGaussianBasisShell cgbs2(basisinfo2, v);

    const Vec3D& tcenter(cgbs1.GetCenter());
    const int tL = cgbs1.GetL();
    const int tK = cgbs1.GetK();
    const int tB = 2;
    const vector<double>& texponents(cgbs1.GetExponents());
    vector<double> tcoefficients;
    const vector<double>& coefficients1(cgbs1.GetCoefficients());
    const vector<double>& coefficients2(cgbs2.GetCoefficients());
    // for the 1st GTO
    tcoefficients.insert(tcoefficients.end(), coefficients1.begin(), coefficients1.end());
    // for the 2nd GTO
    tcoefficients.insert(tcoefficients.end(), coefficients2.begin(), coefficients2.end());
    const vector<double>& tmLfactors(cgbs1.GetmLFactors());
    // Construct a GTO block.
    const ContractedGaussianBasisBlock block(tcenter, tL, tK, tB, texponents, tcoefficients, tmLfactors);

    // Get and output information of a GTO block.
    const Vec3D& center(block.GetCenter());
    const vector<double>& exponents(block.GetExponents());
    const vector<double>& coefficients(block.GetCoefficients());
    const vector<double>& mLfactors(block.GetmLFactors());
    printf("GTO block information:\n");
    printf("Center: %10.8f %10.8f %10.8f\n", center.x, center.y, center.z);
    printf("Angular momentum:   %d\n", block.GetL());
    printf("Contraction degree: %d\n", block.GetK());
    printf("Block size:         %d\n", block.GetB());
    printf("%20s  %20s\n", "alpha", "coefficients");
    for(int b = 0, idxD = 0; b < block.GetB(); ++b)
    {
        printf("Block %d:\n", b);
        for(int k = 0; k < block.GetK(); ++k, ++idxD)
        {
            printf("%20.8f  %20.8f\n", exponents[k], coefficients[idxD]);
        }
    }
    CartesianAngularMomentum crtL(block.GetL());
    int nCRTShellL = crtL.GetNLLz();
    printf("mL factors:\n");
    for(int mL = 0; mL < nCRTShellL; ++mL, ++crtL)
    {
        printf("(%s)  %20.8f\n", crtL.tostr().c_str(), mLfactors[mL]);
    }

    return 0;
}
```

The output of this program is:

```
GTO block information:
Center: 2.00000000 0.00000000 0.00000000
Angular momentum:   2
Contraction degree: 3
Block size:         2
               alpha          coefficients
Block 0:
         24.04000000           40.24718864
```

```
          7.57100000           24.19058090
          2.73200000           13.34729861
Block 1:
         24.04000000           59.48829444
          7.57100000           35.75545144
          2.73200000          -19.72828554
mL factors:
(  2,   0,   0)            0.57735027
(  1,   1,   0)            1.00000000
(  1,   0,   1)            1.00000000
(  0,   2,   0)            0.57735027
(  0,   1,   1)            1.00000000
(  0,   0,   2)            0.57735027
```

Explicitly, `block` represents $2 \times 6 = 12$ GTOs in the following order:

$$0.57735027 \times (x - 2.00000000)^2 F_1 \tag{3.2.9}$$

$$1.00000000 \times (x - 2.00000000)(y - 0.00000000) F_1 \tag{3.2.10}$$

$$1.00000000 \times (x - 2.00000000)(z - 0.00000000) F_1 \tag{3.2.11}$$

$$0.57735027 \times (y - 0.00000000)^2 F_1 \tag{3.2.12}$$

$$1.00000000 \times (y - 0.00000000)(z - 0.00000000) F_1 \tag{3.2.13}$$

$$0.57735027 \times (z - 0.00000000)^2 F_1 \tag{3.2.14}$$

$$0.57735027 \times (x - 2.00000000)^2 F_2 \tag{3.2.15}$$

$$1.00000000 \times (x - 2.00000000)(y - 0.00000000) F_2 \tag{3.2.16}$$

$$1.00000000 \times (x - 2.00000000)(z - 0.00000000) F_2 \tag{3.2.17}$$

$$0.57735027 \times (y - 0.00000000)^2 F_2 \tag{3.2.18}$$

$$1.00000000 \times (y - 0.00000000)(z - 0.00000000) F_2 \tag{3.2.19}$$

$$0.57735027 \times (z - 0.00000000)^2 F_2 \tag{3.2.20}$$

where

$$\begin{aligned} F_1 =& 40.24718864 \exp(-24.04000000 r_A^2) + 24.19058090 \exp(-7.57100000 r_A^2) \\ &+ 13.34729861 \exp(-2.73200000 r_A^2)) \end{aligned} \tag{3.2.21}$$

$$\begin{aligned} F_2 =& 59.48829444 \exp(-24.04000000 r_A^2) + 35.75545144 \exp(-7.57100000 r_A^2) \\ &- 19.72828554 \exp(-2.73200000 r_A^2)) \end{aligned} \tag{3.2.22}$$

$$r_A^2 = (x - 2.00000000)^2 + (y - 0.00000000)^2 + (z - 0.00000000)^2 \tag{3.2.23}$$

3.2.4 Cartesian GTO Block: Unnormalized GTOs

You may note that you can construct a Cartesian GTO Block without using `ContractedGaussianBasisShell` at all, but directly assigning the centers, exponents, coefficients, etc. here for a GTO. You are right. Actually, `ContractedGaussianBasisShell` is only a wrapper for those who want to set up the program rapidly. Actually, when you use `ContractedGaussianBasisBlock` directly, you can define an *un*normalized primitive GTO. However, even in this case, you must ensure that `mLfactors` contains N_L^{CRT} 1's.

```
#include "ContractedGaussianBasisBlock.h"
#include "ContractedGaussianBasisShell.h"
#include "CartesianAngularMomentum.h"
#include "Vec3D.h"
#include <string>
#include <cstdio>

using namespace libreta;
using namespace std;

int main(int argc, char** argv)
{
    const Vec3D v(2, 0, 0);
    const int L = 3;
    const int K = 1;
    const int B = 1;
    vector<double> exponents; expnents.push_back(24.0400000);
    vector<double> coefficents; coefficents.push_back(1.);
    vector<double> mLfactors(10, 1.); // CRT f-shell size: (3+1)(3+2)/2=10
    const ContractedGaussianBasisBlock block(v, L, K, B, exponents, coefficients, mLFactors);
    return 0;
}
```

For example, explicitly, the Cartesian f GTO with angular momentum $(1, 1, 1)$ represented by this `block` is:

$$(x - 2.00000000)(y - 0.00000000)(z - 0.00000000)\exp(-24.04000000r_A^2) \tag{3.2.24}$$

$$r_A^2 = (x - 2.00000000)^2 + (y - 0.00000000)^2 + (z - 0.00000000)^2 \tag{3.2.25}$$

For contracted GTOs, however, it is *absurd* to define an unnormalized one.

3.3 "Zero"-electron Integrals

Related test files:

- `libreta/bin/test/int0e.cpp`

Related header files:

- `libreta/bin/include/GaussianValue.h`
- `libreta/bin/include/ContractedGaussianBasisBlock.h`

We will start with *Gaussian value* (3.3.1). For details you can read the header file `libreta/bin/include/GaussianValue.h`.

$$\chi_a(\mathbf{P}) \tag{3.3.1}$$

For a GTO block of angular momentum L and block size B, and to calculate their values at a point $\mathbf{P}$, one should call

```
void GaussianValue::ValueBlock(
    const ContractedGaussianBasisBlock& blockA,
    const Vec3D& centerP,
    double* IntegralBuffer,
    double* crtint_contrA);
```

- `blockA` The Cartesian GTO block to be calculated.
- `centerP` The point at which the value is to be calculated.
- `IntegralBuffer` A buffer. Minimal size: can be obtained by `GaussianValue::GetGaussianValueIntegralBufferSize(L)`.
- `crtint_contrA` The results. Minimal size: $B(L + 1)(L + 2)/2$. The results are ordered block by block. In each block, the results are lexically ordered.

An example is given below. See also `libreta/bin/test/int0e.cpp`.

```
#include "GaussianValue.h"
#include "ContractedGaussianBasisBlock.h"
#include "ContractedGaussianBasisShell.h"
#include "CartesianAngularMomentum.h"
#include "Vec3D.h"
#include <string>
#include <cstdlib>
#include <cstdio>

using namespace libreta;
using namespace std;

int main(int argc, char** argv)
{
    // Define a GTO block
    const Vec3D v(2., 0., 0.);
    const string basisinfo1("D␣3␣1.0\n24.0400000␣␣0.0069760\n7.5710000␣␣0.0316690\n2.7320000␣␣0.1040060\n");
    const string basisinfo2("D␣3␣1.0\n24.0400000␣␣0.0069760\n7.5710000␣␣0.0316690\n2.7320000␣-0.1040060\n");
    const ContractedGaussianBasisShell cgbs1(basisinfo1, v);
    const ContractedGaussianBasisShell cgbs2(basisinfo2, v);
    const Vec3D& center(cgbs1.GetCenter());
    const int L = cgbs1.GetL();
    const int K = cgbs1.GetK();
    const int B = 2;
    const vector<double>& exponents(cgbs1.GetExponents());
    vector<double> coefficients;
    const vector<double>& coefficients1(cgbs1.GetCoefficients());
    const vector<double>& coefficients2(cgbs2.GetCoefficients());
    coefficients.insert(coefficients.end(), coefficients1.begin(), coefficients1.end()); // for the 1st GTO
    coefficients.insert(coefficients.end(), coefficients2.begin(), coefficients2.end()); // for the 2nd GTO
    const vector<double>& mLfactors(cgbs1.GetmLFactors());
    const ContractedGaussianBasisBlock blockA(center, L, K, B, exponents, coefficients, mLfactors);

    // Calculate the results.
    const Vec3D centerP(1.7, 0.1, 0.2);
    const int BufferSize = GaussianValue::GetGaussianValueIntegralBufferSize(L);
    double* IntegralBuffer = (double*)malloc(sizeof(double)*BufferSize);
    const int nCRTShellL = CartesianAngularMomentum::GetNLLz(L);
    double crtint_contrA[B*nCRTShellL]; // Minimal size
    GaussianValue::ValueBlock(blockA, centerP, IntegralBuffer, crtint_contrA);
    free(IntegralBuffer);

    // Print results.
    printf("GTO␣value␣at␣(%.8f,␣%.8f,␣%.8f):\n", centerP.x, centerP.y, centerP.z);
    for(int iB = 0, idxD = 0; iB < B; ++iB)
    {
        printf("GTO␣Block␣%d:\n", iB);
        CartesianAngularMomentum crtL(blockA.GetL());
        for(int mL = 0; mL < nCRTShellL; ++mL, ++crtL, ++idxD)
        {
            printf("(%s):␣%20.8f\n", crtL.tostr().c_str(), crtint_contrA[idxD]);
        }
    }

    return 0;
}
```

The program output is:

```
GTO value at (1.70000000, 0.10000000, 0.20000000):
GTO Block 0:
(  2,   0,   0):           0.98086489
(  1,   1,   0):          -0.56630261
(  1,   0,   1):          -1.13260521
(  0,   2,   0):           0.10898499
(  0,   1,   1):           0.37753507
(  0,   0,   2):           0.43593995
GTO Block 1:
(  2,   0,   0):           0.05119423
(  1,   1,   0):          -0.02955700
(  1,   0,   1):          -0.05911401
(  0,   2,   0):           0.00568825
(  0,   1,   1):           0.01970467
(  0,   0,   2):           0.02275299
```

3.4 One-electron Integrals

Related test files:

- `libreta/bin/test/int1ederiv.cpp`
- `libreta/bin/test/int1e.cpp`

Related header files:

- `libreta/bin/include/GaussianOneElectronCoulombIntegral.h`

- `libreta/bin/include/GaussianPolynomialIntegral.h`
- `libreta/bin/include/ContractedGaussianBasisBlock.h`

3.4.1 Block-pair Data

Now we will calculate one-electron integrals $(a|\hat{O}|b)$. Assume two Cartesian GTO blocks a and b are represented by two `ContractedGaussianBasisBlock` objects, we will calculate a `BlockPairData` object by `ContractedGaussianBasisBlock::FormBlockPairData`. The `BlockPairData` can be used for the evaluation of all one-electron as well as two-electron integrals, and its definition can be found in `libreta /bin/include/ContractedGaussianBasisBlock.h`. The function you need to call is:

```
void ContractedGaussianBasisBlock::FormBlockPairData(
    const ContractedGaussianBasisBlock& cBlockgsA,
    const ContractedGaussianBasisBlock& cBlockgsB,
    BlockPairDat& pBlockAB);
```

- `cBlockgsA` and `cBlockgsB`: The two Cartesian GTO blocks to be calculated.
- `pBlockAB`: Block pair data.

The `BlockPairData` is a very large object:

```
class BlockPairData {
public:
    BlockPairData(void)  { /* Nothing to do. */ }
    ~BlockPairData(void)  { /* Nothing to do. */ }

    int LA;
    int LB;
    int LAB;          // LAB = LA+LB
    int KA;
    int KB;
    int KAB;          // KAB = KA*KB
    int nBlockA;
    int nBlockB;
    int nBlockAB;     // nBlockAB = nBlockA*nBlockB
    int nCRTShellA;
    int nCRTShellB;
    int nCRTShellAB; // nCRTShellAB = nCRTShellA*nCRTShellB
    int nSphShellA;
    int nSphShellB;
    int nSphShellAB; // nSphShellAB = nSphShellA*nSphShellB
    int idxLAB;       // idxLAB = LA*(LA+1)/2+LB

    Vec3D centerA;
    Vec3D centerB;
    Vec3D AB;

    const vector<double>* as;
    const vector<double>* bs;
    vector<double> a_2s;
    vector<double> b_2s;
    vector<double> ps;
    vector<double> p_2s;
    vector<double> rec_p_2s;
    vector<double> MABs;
    vector<double> MAB_d_ps;
    vector<Vec3D> centerPs;
    vector<Vec3D> PAs;
    vector<Vec3D> PBs;
    vector<double> coefficients;
    vector<double> mLfactors;
};
```

You do not need to know what each member means if you do not want to modify the source codes (If you want to know, see `/bin/include/ContractedGaussianBasisBlock.h`). However, a VERY IMPORTANT thing you have to keep in mind is, when $L_a < L_b$, `libreta` will switch the order of two blocks! For example, $(p|d)$ will be rearranged to $(d|p)$. See also Section 3.1.

Explicitly, when block a and block b have $L_a = 1, B_a = 3, K_a = 5$ and $L_b = 0, B_b = 7, K_b = 10$, respectively, then for the corresponding `BlockPairData`:

```
BlockPairData.LA = 1
BlockPairData.nBlockA = 2
BlockPairData.KA = 5
BlockPairData.LB = 0
```

```
BlockPairData.nBlockB = 7
BlockPairData.KB = 10
...
```

However, if block a and block b have $L_a = 2, B_a = 2, K_a = 5$ and $L_b = 3, B_b = 1, K_b = 1$, respectively, then for the corresponding `BlockPairData`:

```
BlockPairData.LA = 3
BlockPairData.nBlockA = 1
BlockPairData.KA = 1
BlockPairData.LB = 2
BlockPairData.nBlockB = 2
BlockPairData.KB = 5
...
```

Therefore, in `BlockPairData`, `xxA` are always the parameters corresponding to the block with higher angular momentum.

3.4.2 One-electron Coulomb Integrals

First, we will show how to calculate one-electron Coulomb integrals 3.4.1 with `libreta`.

$$V_{ab}(\mathbf{C}) = \int \chi_a(\mathbf{r}) \sum_C \frac{Q_C}{r_C} \chi_b(\mathbf{r}) \mathrm{d}\mathbf{r} \tag{3.4.1}$$

The main function is:

```
void GaussianOneElectronCoulombIntegral::OneElectronCoulombIntegralBlock(
    const BlockPairData& pBlockAB,
    const vector<Vec3D>& centerCs,
    const double* chargeQs,
    double* IntegralBuffer,
    double* crtint_contrA_contrB);
```

- `pBlockAB`: Block pair data.
- `centerCs`: A `std::vector` containing the nuclear positions.
- `chargeQs`: An array containing the nuclear charges. Its size should be consistent with that of `centerCs`.
- `IntegralBuffer`: A buffer. Minimal size: can be obtained by `GaussianOneElectronCoulombIntegral::GetOneElectronCoulombIntegralBufferSize(LA, LB, BA, BB)`.
- `crtint_contrA_contrB` The results. Minimal size: $B_a B_b (L_a + 1)(L_a + 2)/2(L_b + 1)(L_b + 2)/2$.

The following code will show how to do such a calculation. See also `libreta/bin/test/int1e.cpp`.

```
#include "GaussianOneElectronCoulombIntegral.h"
#include "GaussianPolynomialIntegral.h"
#include "ContractedGaussianBasisBlock.h"
#include "ContractedGaussianBasisShell.h"
#include "CartesianAngularMomentum.h"
#include "Vec3D.h"
#include <string>
#include <cstdlib>
#include <cstdio>

using namespace libreta;
using namespace std;

int main(int argc, char** argv)
{
    // Define GTO block1
    const Vec3D v1(0, 0, +1);
    const string basisinfo11("D␣3␣1.0\n24.0400000␣␣0.0069760\n7.5710000␣␣0.0316690\n2.7320000␣␣0.1040060\n")
        ;
    const string basisinfo12("D␣3␣1.0\n24.0400000␣␣0.0069760\n7.5710000␣␣0.0316690\n2.7320000␣-0.1040060\n")
        ;
    const ContractedGaussianBasisShell cgbs11(basisinfo11, v1);
    const ContractedGaussianBasisShell cgbs12(basisinfo12, v1);
    const Vec3D& center1(cgbs11.GetCenter());
    const int L1 = cgbs11.GetL();
    const int K1 = cgbs11.GetK();
```

```
    const int B1 = 2;
    const vector<double>& exponents1(cgbs11.GetExponents());
    vector<double> coefficients1;
    const vector<double>& coefficients11(cgbs11.GetCoefficients());
    const vector<double>& coefficients12(cgbs12.GetCoefficients());
    coefficients1.insert(coefficients1.end(), coefficients11.begin(), coefficients11.end());
    coefficients1.insert(coefficients1.end(), coefficients12.begin(), coefficients12.end());
    const vector<double>& mLfactors1(cgbs11.GetmLFactors());
    const ContractedGaussianBasisBlock block1(center1, L1, K1, B1, exponents1, coefficients1, mLfactors1);

    // Define GTO block2
    const Vec3D v2(0, 0, -1);
    const string basisinfo21("P 2 1.0\n5.3710000  0.0348780\n2.4310000  0.4400060\n");
    const string basisinfo22("P 2 1.0\n5.3710000  0.0435430\n2.4310000 -0.7870060\n");
    const ContractedGaussianBasisShell cgbs21(basisinfo21, v2);
    const ContractedGaussianBasisShell cgbs22(basisinfo22, v2);
    const Vec3D& center2(cgbs21.GetCenter());
    const int L2 = cgbs21.GetL();
    const int K2 = cgbs21.GetK();
    const int B2 = 2;
    const vector<double>& exponents2(cgbs21.GetExponents());
    vector<double> coefficients2;
    const vector<double>& coefficients21(cgbs21.GetCoefficients());
    const vector<double>& coefficients22(cgbs22.GetCoefficients());
    coefficients2.insert(coefficients2.end(), coefficients21.begin(), coefficients21.end());
    coefficients2.insert(coefficients2.end(), coefficients22.begin(), coefficients22.end());
    const vector<double>& mLfactors2(cgbs21.GetmLFactors());
    const ContractedGaussianBasisBlock block2(center2, L2, K2, B2, exponents2, coefficients2, mLfactors2);

    // Form Pair Data
    BlockPairData pBlockAB;
    ContractedGaussianBasisBlock::FormBlockPairData(block1, block2, pBlockAB);

    int BufferSize;
    double* IntegralBuffer;

    // Calculate the one-electron Coulomb integrals.
    vector<Vec3D> centerCs;
    centerCs.push_back(Vec3D(0, 0, 1.));
    centerCs.push_back(Vec3D(0, 0, -1.));
    centerCs.push_back(Vec3D(0, 1., 0));
    const double chargeQs[] = {2., 3., 2.};
    BufferSize = GaussianOneElectronCoulombIntegral::GetOneElectronCoulombIntegralBufferSize(L1, L2, B1, B2)
        ;
    IntegralBuffer = (double*)malloc(sizeof(double)*BufferSize);
    double crtint_contrA_contrB[pBlockAB.nBlockAB*pBlockAB.nCRTShellAB]; // Minimal size
    GaussianOneElectronCoulombIntegral::OneElectronCoulombIntegralBlock(pBlockAB, centerCs, chargeQs,
        IntegralBuffer, crtint_contrA_contrB);
    free(IntegralBuffer);

    // Output results
    printf("One-electron Coulomb integrals:\n");
    const int LA = pBlockAB.LA;
    const int LB = pBlockAB.LB;
    const int nCRTShellA = pBlockAB.nCRTShellA;
    const int nCRTShellB = pBlockAB.nCRTShellB;
    const int BA = pBlockAB.nBlockA;
    const int BB = pBlockAB.nBlockB;
    for(int iBA = 0, idxD = 0; iBA < BA; ++iBA)
    {
        for(int iBB = 0; iBB < BB; ++iBB)
        {
            printf("Block (%d, %d)\n", iBA, iBB);
            CartesianAngularMomentum crtLA(LA);
            for(int mLA = 0; mLA < nCRTShellA; ++mLA, ++crtLA)
            {
                CartesianAngularMomentum crtLB(LB);
                for(int mLB = 0; mLB < nCRTShellB; ++mLB, ++crtLB, ++idxD)
                {
                    printf("(%s|V|%s): %20.8f\n", crtLA.tostr().c_str(), crtLB.tostr().c_str(),
                        crtint_contrA_contrB[idxD]);
                }
            }
        }
    }
    return 0;
}
```

The output is:

```
One-electron Coulomb integrals:
Block (0, 0)
(  2,   0,   0|V|  1,   0,   0):           0.00000000
(  2,   0,   0|V|  0,   1,   0):           0.00117773
(  2,   0,   0|V|  0,   0,   1):           0.05915768
...
(  0,   0,   2|V|  1,   0,   0):           0.00000000
(  0,   0,   2|V|  0,   1,   0):           0.01655489
(  0,   0,   2|V|  0,   0,   1):           0.54202992
Block (0, 1)
...
Block (1, 0)
...
Block (1, 1)
...
```

3.4.3 Differential Integrals

A differential integral is defined as:

$$D_{ab}(d_x, d_y, d_z) = \int \chi_a(\mathbf{r}) \left(\frac{\partial}{\partial x}\right)^{d_x} \left(\frac{\partial}{\partial y}\right)^{d_y} \left(\frac{\partial}{\partial z}\right)^{d_z} \chi_b(\mathbf{r}) \mathrm{d}\mathbf{r} \tag{3.4.2}$$

The rank of this integral is defined as $d_x + d_y + d_z$. Given a rank `D`, `libreta` will calculate all the integrals for $d = 0$ to `D`! Say, `D`=2, then `libreta` will calculate:

- $d = 0$: $(a|b)$ (overlap integrals)
- $d = 1$: $(a|\partial/\partial x|b), (a|\partial/\partial y|b), (a|\partial/\partial z|b)$ (linear momentum integrals)
- $d = 2$: $(a|(\partial/\partial x)^2|b), (a|\partial/\partial x\partial/\partial y|b), (a|\partial/\partial x\partial/\partial z|b), (a|(\partial/\partial y)^2|b), (a|\partial/\partial y\partial/\partial z|b), (a|(\partial/\partial z)^2|b)$

Totally $(D+1)(D+2)(D+3)/6$ kinds of integrals will be calculated simultaneously.

Differential integrals can by calculated by by calling

```
void GaussianPolynomialIntegral::DifferentialIntegralBlock(
    const BlockPairData& pBlockAB,
    int D,
    double* IntegralBuffer,
    double* crtint_contrA_contrB);
```

- `pblockAB` Block pair data.
- `D` The highest rank of differentiation you want to calculate.
- `IntegralBuffer` A buffer. Minimal size: can be obtained by `GaussianPolynomialIntegral::GetPolynomialIntegralBufferSize(LA, LB, D)`.
- `crtint_contrA_contrB` The results. Minimal size: $B_a B_b (L_a+1)(L_a+1)/2(L_b+1)(L_b+1)/2(D+1)(D+2)(D+3)/6$. For each d, the powers of $\partial/\partial x$ etc. are also lexically ordered.

The following code computes the differential integrals:

```
#include "GaussianOneElectronCoulombIntegral.h"
#include "GaussianPolynomialIntegral.h"
#include "ContractedGaussianBasisBlock.h"
#include "ContractedGaussianBasisShell.h"
#include "CartesianAngularMomentum.h"
#include "Vec3D.h"
#include <string>
#include <cstdlib>
#include <cstdio>

using namespace libreta;
using namespace std;

int main(int argc, char** argv)
{
    // Define GTO block1
    const Vec3D v1(0, 0, +1);
    const string basisinfo11("D␣3␣1.0\n24.0400000␣␣0.0069760\n7.5710000␣␣0.0316690\n2.7320000␣␣0.1040060\n")
        ;
    const string basisinfo12("D␣3␣1.0\n24.0400000␣␣0.0069760\n7.5710000␣␣0.0316690\n2.7320000␣-0.1040060\n")
      ;
    const ContractedGaussianBasisShell cgbs11(basisinfo11, v1);
    const ContractedGaussianBasisShell cgbs12(basisinfo12, v1);
    const Vec3D& center1(cgbs11.GetCenter());
    const int L1 = cgbs11.GetL();
    const int K1 = cgbs11.GetK();
    const int B1 = 2;
    const vector<double>& exponents1(cgbs11.GetExponents());
    vector<double> coefficients1;
    const vector<double>& coefficients11(cgbs11.GetCoefficients());
    const vector<double>& coefficients12(cgbs12.GetCoefficients());
    coefficients1.insert(coefficients1.end(), coefficients11.begin(), coefficients11.end());
    coefficients1.insert(coefficients1.end(), coefficients12.begin(), coefficients12.end());
    const vector<double>& mLfactors1(cgbs11.GetmLFactors());
    const ContractedGaussianBasisBlock block1(center1, L1, K1, B1, exponents1, coefficients1, mLfactors1);

    // Define GTO block2
    const Vec3D v2(0, 0, -1);
    const string basisinfo21("P␣2␣1.0\n5.3710000␣␣0.0348780\n2.4310000␣␣0.4400060\n");
```

```
    const string basisinfo22("P␣2␣1.0\n5.3710000␣␣0.0435430\n2.4310000␣-0.7870060\n");
    const ContractedGaussianBasisShell cgbs21(basisinfo21, v2);
    const ContractedGaussianBasisShell cgbs22(basisinfo22, v2);
    const Vec3D& center2(cgbs21.GetCenter());
    const int L2 = cgbs21.GetL();
    const int K2 = cgbs21.GetK();
    const int B2 = 2;
    const vector<double>& exponents2(cgbs21.GetExponents());
    vector<double> coefficients2;
    const vector<double>& coefficients21(cgbs21.GetCoefficients());
    const vector<double>& coefficients22(cgbs22.GetCoefficients());
    coefficients2.insert(coefficients2.end(), coefficients21.begin(), coefficients21.end());
    coefficients2.insert(coefficients2.end(), coefficients22.begin(), coefficients22.end());
    const vector<double>& mLfactors2(cgbs21.GetmLFactors());
    const ContractedGaussianBasisBlock block2(center2, L2, K2, B2, exponents2, coefficients2, mLfactors2);

    // Form Pair Data
    BlockPairData pBlockAB;
    ContractedGaussianBasisBlock::FormBlockPairData(block1, block2, pBlockAB);

    int BufferSize;
    double* IntegralBuffer;

    // Calculate the differential integrals.
    const int D = 2;
    BufferSize = GaussianPolynomialIntegral::GetPolynomialIntegralBufferSize(L1, L2, D);
    IntegralBuffer = (double*)malloc(sizeof(double)*BufferSize);
    double crtint_contrA_contrB2[pBlockAB.nBlockAB*pBlockAB.nCRTShellAB*(D+1)*(D+2)*(D+3)/6]; // Minimal
        size
    GaussianPolynomialIntegral::DifferentialIntegralBlock(pBlockAB, D, IntegralBuffer, crtint_contrA_contrB2
        );
    free(IntegralBuffer);

    // Output results
    printf("Differential␣integrals␣up␣to␣rank␣D␣=␣%d:\n", D);
    const int LA = pBlockAB.LA;
    const int LB = pBlockAB.LB;
    const int nCRTShellA = pBlockAB.nCRTShellA;
    const int nCRTShellB = pBlockAB.nCRTShellB;
    const int BA = pBlockAB.nBlockA;
    const int BB = pBlockAB.nBlockB;
    for(int iBA = 0, idxD = 0; iBA < BA; ++iBA)
    {
        for(int iBB = 0; iBB < BB; ++iBB)
        {
            printf("Block␣(%d,␣%d)\n", iBA, iBB);
            for(int d0 = 0; d0 <= D; ++d0)
            {
                CartesianAngularMomentum crtLD(d0);
                const int nCRTShellD = crtLD.GetNLLz();
                for(int mLD = 0; mLD < nCRTShellD; ++mLD, ++crtLD)
                {
                    CartesianAngularMomentum crtLA(LA);
                    for(int mLA = 0; mLA < nCRTShellA; ++mLA, ++crtLA)
                    {
                        CartesianAngularMomentum crtLB(LB);
                        for(int mLB = 0; mLB < nCRTShellB; ++mLB, ++crtLB, ++idxD)
                        {
                            printf("(%s|(d/dx)^%d(d/dy)^%d(d/dz)^%d|%s):␣%20.8f\n", crtLA.tostr().c_str(),
                                    crtLD.GetLx(), crtLD.GetLy(), crtLD.GetLz(),
                                    crtLB.tostr().c_str(), crtint_contrA_contrB2[idxD]);
                        }
                    }
                }
            }
        }
    }

    return 0;
}
```

The output is:

```
Block (0, 0)
(  2,   0,   0|(d/dx)^0(d/dy)^0(d/dz)^0|  1,   0,   0):           0.00000000
...
(  0,   0,   2|(d/dx)^0(d/dy)^0(d/dz)^0|  0,   0,   1):           0.07795362
(  2,   0,   0|(d/dx)^1(d/dy)^0(d/dz)^0|  1,   0,   0):          -0.00343104
...
(  0,   0,   2|(d/dx)^1(d/dy)^0(d/dz)^0|  0,   0,   1):           0.00000000
(  2,   0,   0|(d/dx)^0(d/dy)^1(d/dz)^0|  1,   0,   0):           0.00000000
...
(  0,   0,   2|(d/dx)^0(d/dy)^0(d/dz)^1|  0,   0,   1):          -0.28550356
...
... (d/dx)^2(d/dy)^0(d/dz)^0 ...
... (d/dx)^1(d/dy)^1(d/dz)^0 ...
... (d/dx)^1(d/dy)^0(d/dz)^1 ...
... (d/dx)^0(d/dy)^2(d/dz)^0 ...
... (d/dx)^0(d/dy)^1(d/dz)^1 ...
...
(  0,   0,   2|(d/dx)^0(d/dy)^0(d/dz)^2|  0,   1,   0):           0.00000000
(  0,   0,   2|(d/dx)^0(d/dy)^0(d/dz)^2|  0,   0,   1):           0.79076969
Block (0, 1)
...
Block (1, 0)
...
Block (1, 1)
...
```

3.4.4 Multipole Integrals

A moltipole integral is defined as:

$$M_{ab}(\mathbf{C}, m_x, m_y, m_z) = \int \chi_a(\mathbf{r})(x - C_x)^{m_x}(y - C_y)^{m_y}(z - C_z)^{m_z}\chi_b(\mathbf{r})\mathrm{d}\mathbf{r} \qquad (3.4.3)$$

The rank of this integral is defined as $m_x + m_y + m_z$. Given a rank `M`, `libreta` will calculate all the integrals for $m = 0$ to `M`! Say, `M`=2, then `libreta` will calculate:

- $m = 0$: $(a|b)$ (overlap integrals)
- $m = 1$: $(a|(x - C_x)|b), (a|(y - C_y)|b), (a|(z - C_z)|b)$ (dipole)
- $m = 2$: $(a|(x - C_x)^2|b), (a|(x - C_x)(y - C_y)|b), (a|(x - C_x)(z - C_z)|b), (a|(y - C_y)^2|b), (a|(y - C_y)(z - C_z)|b), (a|(z - C_z)^2|b)$

Totally $(M + 1)(M + 2)(M + 3)/6$ kinds of integrals will be calculated simultaneously.
moltipole integrals can by calculated by by calling

```
void GaussianPolynomialIntegral::MultipoleIntegralBlock(
    const BlockPairData& pBlockAB,
    const Vec3D& centerC,
    int M,
    double* IntegralBuffer,
    double* crtint_contrA_contrB);
```

- `pblockAB` Block pair data.
- `centerC` The multipole center.
- `M` The highest rank of multipole you want to calcualte.
- `IntegralBuffer` A buffer. Minimal size: can be obtained by `GaussianPolynomialIntegral::GetPolynomialIntegralBufferSize(LA, LB, M)`.
- `crtint_contrA_contrB` The results. Minimal size: $B_aB_b(L_a + 1)(L_a + 1)/2(L_b + 1)(L_b + 1)/2(M + 1)(M + 2)(M + 3)/6$. For each m, the powers of $(x - C_x)$ etc. are also lexically ordered.

The following code computes the multipole integrals:

```
#include "GaussianOneElectronCoulombIntegral.h"
#include "GaussianPolynomialIntegral.h"
#include "ContractedGaussianBasisBlock.h"
#include "ContractedGaussianBasisShell.h"
#include "CartesianAngularMomentum.h"
#include "Vec3D.h"
#include <string>
#include <cstdlib>
#include <cstdio>

using namespace libreta;
using namespace std;

int main(int argc, char** argv)
{
    // Define GTO block1
    const Vec3D v1(0, 0, +1);
    const string basisinfo11("D␣3␣1.0\n24.0400000␣␣0.0069760\n7.5710000␣␣0.0316690\n2.7320000␣␣0.1040060\n")
        ;
    const string basisinfo12("D␣3␣1.0\n24.0400000␣␣0.0069760\n7.5710000␣␣0.0316690\n2.7320000␣-0.1040060\n")
        ;
    const ContractedGaussianBasisShell cgbs11(basisinfo11, v1);
    const ContractedGaussianBasisShell cgbs12(basisinfo12, v1);
    const Vec3D& center1(cgbs11.GetCenter());
    const int L1 = cgbs11.GetL();
    const int K1 = cgbs11.GetK();
    const int B1 = 2;
    const vector<double>& exponents1(cgbs11.GetExponents());
    vector<double> coefficients1;
    const vector<double>& coefficients11(cgbs11.GetCoefficients());
    const vector<double>& coefficients12(cgbs12.GetCoefficients());
    coefficients1.insert(coefficients1.end(), coefficients11.begin(), coefficients11.end());
    coefficients1.insert(coefficients1.end(), coefficients12.begin(), coefficients12.end());
    const vector<double>& mLfactors1(cgbs11.GetmLFactors());
```

```
    const ContractedGaussianBasisBlock block1(center1, L1, K1, B1, exponents1, coefficients1, mLfactors1);

    // Define GTO block2
    const Vec3D v2(0, 0, -1);
    const string basisinfo21("P␣2␣1.0\n5.3710000␣␣0.0348780\n2.4310000␣␣0.4400060\n");
    const string basisinfo22("P␣2␣1.0\n5.3710000␣␣0.0435430\n2.4310000␣-0.7870060\n");
    const ContractedGaussianBasisShell cgbs21(basisinfo21, v2);
    const ContractedGaussianBasisShell cgbs22(basisinfo22, v2);
    const Vec3D& center2(cgbs21.GetCenter());
    const int L2 = cgbs21.GetL();
    const int K2 = cgbs21.GetK();
    const int B2 = 2;
    const vector<double>& exponents2(cgbs21.GetExponents());
    vector<double> coefficients2;
    const vector<double>& coefficients21(cgbs21.GetCoefficients());
    const vector<double>& coefficients22(cgbs22.GetCoefficients());
    coefficients2.insert(coefficients2.end(), coefficients21.begin(), coefficients21.end());
    coefficients2.insert(coefficients2.end(), coefficients22.begin(), coefficients22.end());
    const vector<double>& mLfactors2(cgbs21.GetmLFactors());
    const ContractedGaussianBasisBlock block2(center2, L2, K2, B2, exponents2, coefficients2, mLfactors2);

    // Form Pair Data
    BlockPairData pBlockAB;
    ContractedGaussianBasisBlock::FormBlockPairData(block1, block2, pBlockAB);

    int BufferSize;
    double* IntegralBuffer;
    // Calculate the multipole integrals.
    const int M = 2;
    const Vec3D centerM(1.,2.,3.);
    BufferSize = GaussianPolynomialIntegral::GetPolynomialIntegralBufferSize(L1, L2, M);
    IntegralBuffer = (double*)malloc(sizeof(double)*BufferSize);
    double crtint_contrA_contrB3[pBlockAB.nBlockAB*pBlockAB.nCRTShellAB*(M+1)*(M+2)*(M+3)/6]; // Minimal
        size
    GaussianPolynomialIntegral::MultipoleIntegralBlock(pBlockAB, centerM, M, IntegralBuffer,
        crtint_contrA_contrB3);
    free(IntegralBuffer);

    // Output results
    printf("Multipole␣integrals␣up␣to␣rank␣M␣=␣%d:\n", M);
    const int LA = pBlockAB.LA;
    const int LB = pBlockAB.LB;
    const int nCRTShellA = pBlockAB.nCRTShellA;
    const int nCRTShellB = pBlockAB.nCRTShellB;
    const int BA = pBlockAB.nBlockA;
    const int BB = pBlockAB.nBlockB;
    for(int iBA = 0, idxD = 0; iBA < BA; ++iBA)
    {
        for(int iBB = 0; iBB < BB; ++iBB)
        {
            printf("Block␣(%d,␣%d)\n", iBA, iBB);
            for(int m0 = 0; m0 <= M; ++m0)
            {
                CartesianAngularMomentum crtLM(m0);
                const int nCRTShellM = crtLM.GetNLLz();
                for(int mLM = 0; mLM < nCRTShellM; ++mLM, ++crtLM)
                {
                    CartesianAngularMomentum crtLA(LA);
                    for(int mLA = 0; mLA < nCRTShellA; ++mLA, ++crtLA)
                    {
                        CartesianAngularMomentum crtLB(LB);
                        for(int mLB = 0; mLB < nCRTShellB; ++mLB, ++crtLB, ++idxD)
                        {
                            printf("(%s|(x-%.3f)^%d(y-%.3f))^%d(z-%.3f))^%d|%s):␣%20.8f\n",
                                    crtLA.tostr().c_str(),
                                    centerM.x, crtLM.GetLx(),
                                    centerM.y, crtLM.GetLy(),
                                    centerM.z, crtLM.GetLz(),
                                    crtLB.tostr().c_str(), crtint_contrA_contrB2[idxD]);
                        }
                    }
                }
            }
        }
    }

    return 0;
}
```

The output is is similar to that for the differential integrals in the last Subsection.

3.4.5 First and Second Derivatives of One-electron Integrals

The calculation of first and/or second derivatives are very similar to that of the integrals themselves. The following points should be kept in mind:

- `libreta` never calculate the derivatives along. For example, when `XXDerivSecondBlock` is called, it will not only calculate second derivatives, but also calculate the integrals and first derivatives.

- For differential integrals, we only calculate 3 and 6 first and second derivatives, respectively, since only 2 nuclei are involved.
- For multipole and one-electron Coulomb integrals, we calculate 6 and 21 first and second derivatives, respectively, since 3 nuclei are involved.
- For more details, read `libreta/bin/include/GaussianPolynomialIntegral.h` and `libreta/bin/include/GaussianOneElectronCoulombIntegral.h`.

The following code computes the one-electron Coulomb integrals as well as their first and second derivatives:

```
#include "GaussianOneElectronCoulombIntegral.h"
#include "GaussianPolynomialIntegral.h"
#include "ContractedGaussianBasisBlock.h"
#include "ContractedGaussianBasisShell.h"
#include "CartesianAngularMomentum.h"
#include "Vec3D.h"
#include <string>
#include <cstdlib>
#include <cstdio>

using namespace libreta;
using namespace std;

int main(int argc, char** argv)
{
    // Define GTO block1
    const Vec3D v1(0, 0, +1);
    const string basisinfo11("D␣3␣1.0\n24.0400000␣␣0.0069760\n7.5710000␣␣0.0316690\n2.7320000␣␣0.1040060\n")
        ;
    const string basisinfo12("D␣3␣1.0\n24.0400000␣␣0.0069760\n7.5710000␣␣0.0316690\n2.7320000␣-0.1040060\n")
        ;
    const ContractedGaussianBasisShell cgbs11(basisinfo11, v1);
    const ContractedGaussianBasisShell cgbs12(basisinfo12, v1);
    const Vec3D& center1(cgbs11.GetCenter());
    const int L1 = cgbs11.GetL();
    const int K1 = cgbs11.GetK();
    const int B1 = 2;
    const vector<double>& exponents1(cgbs11.GetExponents());
    vector<double> coefficients1;
    const vector<double>& coefficients11(cgbs11.GetCoefficients());
    const vector<double>& coefficients12(cgbs12.GetCoefficients());
    coefficients1.insert(coefficients1.end(), coefficients11.begin(), coefficients11.end());
    coefficients1.insert(coefficients1.end(), coefficients12.begin(), coefficients12.end());
    const vector<double>& mLfactors1(cgbs11.GetmLFactors());
    const ContractedGaussianBasisBlock block1(center1, L1, K1, B1, exponents1, coefficients1, mLfactors1);

    // Define GTO block2
    const Vec3D v2(0, 0, -1);
    const string basisinfo21("P␣2␣1.0\n5.3710000␣␣0.0348780\n2.4310000␣␣0.4400060\n");
    const string basisinfo22("P␣2␣1.0\n5.3710000␣␣0.0435430\n2.4310000␣-0.7870060\n");
    const ContractedGaussianBasisShell cgbs21(basisinfo21, v2);
    const ContractedGaussianBasisShell cgbs22(basisinfo22, v2);
    const Vec3D& center2(cgbs21.GetCenter());
    const int L2 = cgbs21.GetL();
    const int K2 = cgbs21.GetK();
    const int B2 = 2;
    const vector<double>& exponents2(cgbs21.GetExponents());
    vector<double> coefficients2;
    const vector<double>& coefficients21(cgbs21.GetCoefficients());
    const vector<double>& coefficients22(cgbs22.GetCoefficients());
    coefficients2.insert(coefficients2.end(), coefficients21.begin(), coefficients21.end());
    coefficients2.insert(coefficients2.end(), coefficients22.begin(), coefficients22.end());
    const vector<double>& mLfactors2(cgbs21.GetmLFactors());
    const ContractedGaussianBasisBlock block2(center2, L2, K2, B2, exponents2, coefficients2, mLfactors2);

    // Form Pair Data
    BlockPairData pBlockAB;
    ContractedGaussianBasisBlock::FormBlockPairData(block1, block2, pBlockAB);

    int BufferSize;
    double* IntegralBuffer;

    // Calculate the one-electron Coulomb integrals.
    const int ResultSize = pBlockAB.nBlockAB*pBlockAB.nCRTShellAB;
    double crtint_contrA_contrB[ResultSize]; // Minimal size
    double derivfirstcrtint_contrA_contrB[ResultSize*6]; // Minimal size
    double derivsecondcrtint_contrA_contrB[ResultSize*21]; // Minimal size
    const Vec3D centerC(0.5, 0.5, 0.5);
    const double chargeQ = 2.;

    BufferSize = GaussianOneElectronCoulombIntegral::GetOneElectronCoulombIntegralDerivSecondBufferSize(L1,
        L2, B1, B2);
    IntegralBuffer = (double*)malloc(sizeof(double)*BufferSize);
    GaussianOneElectronCoulombIntegral::OneElectronCoulombIntegralDerivSecondBlock(pBlockAB, centerC,
        chargeQ, IntegralBuffer, crtint_contrA_contrB, derivfirstcrtint_contrA_contrB,
        derivsecondcrtint_contrA_contrB);
    free(IntegralBuffer);

    // Output results
```

```
    printf("One-electron␣Coulomb␣integral␣derivatives:\n");
    const int LA = pBlockAB.LA;
    const int LB = pBlockAB.LB;
    const int nCRTShellA = pBlockAB.nCRTShellA;
    const int nCRTShellB = pBlockAB.nCRTShellB;
    const int BA = pBlockAB.nBlockA;
    const int BB = pBlockAB.nBlockB;

    const char* firstderiv[] = {"d/dAx", "d/dAy", "d/dAz", "d/dBx", "d/dBy", "d/dBz"};
    for(int nr = 0; nr < 6; ++nr)
    {
        printf("First␣derivatives:␣%s\n", firstderiv[nr]);
        for(int iBA = 0, idxD = 0; iBA < BA; ++iBA)
        {
            for(int iBB = 0; iBB < BB; ++iBB)
            {
                printf("Block␣(%d,␣%d)\n", iBA, iBB);
                CartesianAngularMomentum crtLA(LA);
                for(int mLA = 0; mLA < nCRTShellA; ++mLA, ++crtLA)
                {
                    CartesianAngularMomentum crtLB(LB);
                    for(int mLB = 0; mLB < nCRTShellB; ++mLB, ++crtLB, ++idxD)
                    {
                        printf("(%s|V|%s):␣%20.8f\n", crtLA.tostr().c_str(), crtLB.tostr().c_str(),
                            derivfirstcrtint_contrA_contrB[idxD+nr*ResultSize]);
                    }
                }
            }
        }
    }

    const char* secondderiv[] = {"d2/dAxdAx", "d2/dAxdAy", "d2/dAxdAz", "d2/dAxdBx",
        "d2/dAxdBy", "d2/dAxdBz", "d2/dAydAy", "d2/dAydAz", "d2/dAydBx", "d2/dAydBy",
        "d2/dAydBz", "d2/dAzdAz", "d2/dAzdBx", "d2/dAzdBy", "d2/dAzdBz", "d2/dBxdBx",
        "d2/dBxdBy", "d2/dBxdBz", "d2/dBydBy", "d2/dBydBz", "d2/dBzdBz"};
    for(int nr = 0; nr < 21; ++nr)
    {
        printf("Second␣derivatives:␣%s\n", secondderiv[nr]);
        for(int iBA = 0, idxD = 0; iBA < BA; ++iBA)
        {
            for(int iBB = 0; iBB < BB; ++iBB)
            {
                printf("Block␣(%d,␣%d)\n", iBA, iBB);
                CartesianAngularMomentum crtLA(LA);
                for(int mLA = 0; mLA < nCRTShellA; ++mLA, ++crtLA)
                {
                    CartesianAngularMomentum crtLB(LB);
                    for(int mLB = 0; mLB < nCRTShellB; ++mLB, ++crtLB, ++idxD)
                    {
                        printf("(%s|V|%s):␣%20.8f\n", crtLA.tostr().c_str(), crtLB.tostr().c_str(),
                            derivsecondcrtint_contrA_contrB[idxD+nr*ResultSize]);
                    }
                }
            }
        }
    }


    return 0;
}
```

3.5 Two-electron Integrals

Related test files:

- `libreta/bin/test/int2e.cpp`
- `libreta/bin/test/int2ederiv.cpp`

Related header files:

- `libreta/bin/include/GaussianTwoElectronCoulombIntegral.h`
- `libreta/bin/include/ContractedGaussianBasisBlock.h`

3.5.1 Args_TwoElectronCoulombIntegral and Integral Order

To calculate the two-electron Coulomb integral, you have to form a `Args_TwoElectronCoulombIntegral` object using two `BlockPairData`. This can be achieved by

```
GaussianTwoElectronCoulombIntegral::Args_TwoElectronCoulombIntegral(
    const BlockPairData& pBlockAB,
    const BlockPairData& pBlockCD);
```

- `pBlockAB` and `pBlockCD`: The corresponding two `BlockPairData`.

You can check `libreta/bin/include/GaussianTwoElectronCoulombIntegral.h`) for its definition. A VERY IMPORTANT thing you have to keep in mind is, `libreta` will switch the order of two block-pairs when `pBlockAB.idxLAB<pBlockCD.idxLAB`, i.e.

$$L_a(L_a+1)/2+L_b<L_c(L_c+1)/2+L_d \tag{3.5.1}$$

For example, $(ds|fp)$ will be rearranged to $(fp|ds)$. See also Section 3.1.

3.5.2 Two-electron Integrals

Two-electron Coulomb Integral is defined as

$$(ab|cd)=\int \chi_a(\mathbf{r}_1)\chi_b(\mathbf{r}_1)\frac{1}{r_{12}}\chi_c(\mathbf{r}_2)\chi_d(\mathbf{r}_2)\mathrm{d}\mathbf{r}_1\mathrm{d}\mathbf{r}_2 \tag{3.5.2}$$

which can be calculated by

```
void GaussianTwoElectronCoulombIntegral::TwoElectronCoulombIntegralBlock(
    const Args_TwoElectronCoulombIntegral& args,
    double* IntegralBuffer,
    double* crtint_contrA_contrB_contrC_contrD);
```

- `args`: Arguments for the calculation.
- `IntegralBuffer`: A buffer. Minimal size: can be obtained by `GaussianTwoElectronCoulombIntegral::GetTwoElectronCoulombIntegralBufferSize(LA, LB, LC, LD, BA, BB, BC, BD)`.
- `crtint_contrA_contrB_contrC_contrD` The results. Minimal size: $B_aB_bB_cB_d(L_a+1)(L_a+2)/2(L_b+1)(L_b+2)/2(L_c+1)(L_c+2)/2(L_d+1)(L_d+2)/2$.

Here is an example for $(dp|ps)$:

```
#include "GaussianTwoElectronCoulombIntegral.h"
#include "ContractedGaussianBasisBlock.h"
#include "ContractedGaussianBasisShell.h"
#include "CartesianAngularMomentum.h"
#include "Vec3D.h"
#include <string>
#include <cstdlib>
#include <cstdio>

using namespace libreta;
using namespace std;

int main(int argc, char** argv)
{
    // Define GTO block1
    const Vec3D v1(0.8, 0.7, -.5);
    const string basisinfo11("D␣3␣1.0\n2.40400000␣␣0.0069760\n1.5710000␣␣0.0316690\n0.7320000␣␣0.1040060\n")
        ;
    const string basisinfo12("D␣3␣1.0\n2.40400000␣␣0.0069760\n1.5710000␣␣0.0316690\n0.7320000␣-0.1040060\n")
        ;
    const ContractedGaussianBasisShell cgbs11(basisinfo11, v1);
    const ContractedGaussianBasisShell cgbs12(basisinfo12, v1);
    const Vec3D& center1(cgbs11.GetCenter());
    const int L1 = cgbs11.GetL();
    const int K1 = cgbs11.GetK();
    const int B1 = 2;
    const vector<double>& exponents1(cgbs11.GetExponents());
    vector<double> coefficients1;
    const vector<double>& coefficients11(cgbs11.GetCoefficients());
    const vector<double>& coefficients12(cgbs12.GetCoefficients());
    coefficients1.insert(coefficients1.end(), coefficients11.begin(), coefficients11.end());
    coefficients1.insert(coefficients1.end(), coefficients12.begin(), coefficients12.end());
    const vector<double>& mLfactors1(cgbs11.GetmLFactors());
    const ContractedGaussianBasisBlock block1(center1, L1, K1, B1, exponents1, coefficients1, mLfactors1);

    // Define GTO block2
    const Vec3D v2(0.3, 0.4, -1);
    const string basisinfo21("P␣2␣1.0\n5.3710000␣␣0.0348780\n0.24310000␣␣0.4400060\n");
    const string basisinfo22("P␣2␣1.0\n5.3710000␣␣0.0435430\n0.24310000␣-0.7870060\n");
    const ContractedGaussianBasisShell cgbs21(basisinfo21, v2);
    const ContractedGaussianBasisShell cgbs22(basisinfo22, v2);
    const Vec3D& center2(cgbs21.GetCenter());
```

```
const int L2 = cgbs21.GetL();
const int K2 = cgbs21.GetK();
const int B2 = 2;
const vector<double>& exponents2(cgbs21.GetExponents());
vector<double> coefficients2;
const vector<double>& coefficients21(cgbs21.GetCoefficients());
const vector<double>& coefficients22(cgbs22.GetCoefficients());
coefficients2.insert(coefficients2.end(), coefficients21.begin(), coefficients21.end());
coefficients2.insert(coefficients2.end(), coefficients22.begin(), coefficients22.end());
const vector<double>& mLfactors2(cgbs21.GetmLFactors());
const ContractedGaussianBasisBlock block2(center2, L2, K2, B2, exponents2, coefficients2, mLfactors2);

// Form Pair Data 1,2
BlockPairData pBlockAB;
ContractedGaussianBasisBlock::FormBlockPairData(block1, block2, pBlockAB);

// Define GTO block3
const Vec3D v3(0.6, -0.8, -1.3);
const string basisinfo31("P␣2␣1.0\n1.0400000␣␣0.0069760\n0.35710000␣␣0.0316690\n");
const string basisinfo32("P␣2␣1.0\n1.0400000␣␣0.0069760\n0.35710000␣␣-0.0316690\n");
const ContractedGaussianBasisShell cgbs31(basisinfo31, v3);
const ContractedGaussianBasisShell cgbs32(basisinfo32, v3);
const Vec3D& center3(cgbs31.GetCenter());
const int L3 = cgbs31.GetL();
const int K3 = cgbs31.GetK();
const int B3 = 2;
const vector<double>& exponents3(cgbs31.GetExponents());
vector<double> coefficients3;
const vector<double>& coefficients31(cgbs31.GetCoefficients());
const vector<double>& coefficients32(cgbs32.GetCoefficients());
coefficients3.insert(coefficients3.end(), coefficients31.begin(), coefficients31.end());
coefficients3.insert(coefficients3.end(), coefficients32.begin(), coefficients32.end());
const vector<double>& mLfactors3(cgbs31.GetmLFactors());
const ContractedGaussianBasisBlock block3(center3, L3, K3, B3, exponents3, coefficients3, mLfactors3);

// Define GTO block4
const Vec3D v4(-0.5, +0.5, 0.5);
const string basisinfo41("S␣2␣1.0\n1.0400000␣␣0.006760\n0.35710000␣␣0.031690\n");
const string basisinfo42("S␣2␣1.0\n1.0400000␣␣0.006760\n0.35710000␣␣-0.031690\n");
const ContractedGaussianBasisShell cgbs41(basisinfo41, v4);
const ContractedGaussianBasisShell cgbs42(basisinfo42, v4);
const Vec3D& center4(cgbs41.GetCenter());
const int L4 = cgbs41.GetL();
const int K4 = cgbs41.GetK();
const int B4 = 2;
const vector<double>& exponents4(cgbs41.GetExponents());
vector<double> coefficients4;
const vector<double>& coefficients41(cgbs41.GetCoefficients());
const vector<double>& coefficients42(cgbs42.GetCoefficients());
coefficients4.insert(coefficients4.end(), coefficients41.begin(), coefficients41.end());
coefficients4.insert(coefficients4.end(), coefficients42.begin(), coefficients42.end());
const vector<double>& mLfactors4(cgbs41.GetmLFactors());
const ContractedGaussianBasisBlock block4(center4, L4, K4, B4, exponents4, coefficients4, mLfactors4);

// Form Pair Data 3,4
BlockPairData pBlockCD;
ContractedGaussianBasisBlock::FormBlockPairData(block3, block4, pBlockCD);

// Form args
const GaussianTwoElectronCoulombIntegral::Args_TwoElectronCoulombIntegral args(pBlockAB, pBlockCD);

// Calculation.
const int LA = pBlockAB.LA;
const int LB = pBlockAB.LB;
const int LC = pBlockCD.LA;
const int LD = pBlockCD.LB;
const int nCRTShellA = pBlockAB.nCRTShellA;
const int nCRTShellB = pBlockAB.nCRTShellB;
const int nCRTShellC = pBlockCD.nCRTShellA;
const int nCRTShellD = pBlockCD.nCRTShellB;
const int BA = pBlockAB.nBlockA;
const int BB = pBlockAB.nBlockB;
const int BC = pBlockCD.nBlockA;
const int BD = pBlockCD.nBlockB;
const size_t ResultSize = nCRTShellA*nCRTShellB*nCRTShellC*nCRTShellD*BA*BB*BC*BD;

double crtint_contrA_contrB_contrC_contrD[ResultSize];
double derivfirstcrtint_contrA_contrB_contrC_contrD[ResultSize*9];
double derivsecondcrtint_contrA_contrB_contrC_contrD[ResultSize*45];

size_t BufferSize;
double* IntegralBuffer;

BufferSize = GaussianTwoElectronCoulombIntegral::GetTwoElectronCoulombIntegralBufferSize(LA, LB, LC, LD,
      BA, BB, BC, BD);
IntegralBuffer = (double*)malloc(sizeof(double)*BufferSize);
GaussianTwoElectronCoulombIntegral::TwoElectronCoulombIntegralBlock(args, IntegralBuffer,
     crtint_contrA_contrB_contrC_contrD);
free(IntegralBuffer);

printf("Two-electron␣Coulomb␣integrals:\n");
for(int iBA = 0, idxD = 0; iBA < BA; ++iBA)
{
    for(int iBB = 0; iBB < BB; ++iBB)
    {
        for(int iBC = 0; iBC < BC; ++iBC)
        {
            for(int iBD = 0; iBD < BD; ++iBD)
            {
```

```
                    printf("Block (%d, %d, %d, %d)\n", iBA, iBB, iBC, iBD);
                    printf("%32s\n", "AM");
                    CartesianAngularMomentum crtLA(LA);
                    for(int mLA = 0; mLA < nCRTShellA; ++mLA, ++crtLA)
                    {
                        CartesianAngularMomentum crtLB(LB);
                        for(int mLB = 0; mLB < nCRTShellB; ++mLB, ++crtLB)
                        {
                            CartesianAngularMomentum crtLC(LC);
                            for(int mLC = 0; mLC < nCRTShellC; ++mLC, ++crtLC)
                            {
                                CartesianAngularMomentum crtLD(LD);
                                for(int mLD = 0; mLD < nCRTShellD; ++mLD, ++crtLD, ++idxD)
                                {
                                    printf("((%d,%d,%d)(%d,%d,%d)|(%d,%d,%d)(%d,%d,%d)): %12.8f\n",
                                            crtLA.GetLx(), crtLA.GetLy(), crtLA.GetLz(),
                                            crtLB.GetLx(), crtLB.GetLy(), crtLB.GetLz(),
                                            crtLC.GetLx(), crtLC.GetLy(), crtLC.GetLz(),
                                            crtLD.GetLx(), crtLD.GetLy(), crtLD.GetLz(),
                                            crtint_contrA_contrB_contrC_contrD[idxD]);
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    return 0;
}
```

3.5.3 First and Second Derivatives of Two-electron Integrals

For two-electron integrals, we only calculate 9 and 45 first and second derivatives, respectively, since 4 nuclei are involved. An example is given below:

```
#include "GaussianTwoElectronCoulombIntegral.h"
#include "ContractedGaussianBasisBlock.h"
#include "ContractedGaussianBasisShell.h"
#include "CartesianAngularMomentum.h"
#include "Vec3D.h"
#include <string>
#include <cstdlib>
#include <cstdio>

using namespace libreta;
using namespace std;

int main(int argc, char** argv)
{
    // Define GTO block1
    const Vec3D v1(0.8, 0.7, -.5);
    const string basisinfo11("D 3 1.0\n2.40400000  0.0069760\n1.5710000  0.0316690\n0.7320000  0.1040060\n")
        ;
    const string basisinfo12("D 3 1.0\n2.40400000  0.0069760\n1.5710000  0.0316690\n0.7320000 -0.1040060\n")
        ;
    const ContractedGaussianBasisShell cgbs11(basisinfo11, v1);
    const ContractedGaussianBasisShell cgbs12(basisinfo12, v1);
    const Vec3D& center1(cgbs11.GetCenter());
    const int L1 = cgbs11.GetL();
    const int K1 = cgbs11.GetK();
    const int B1 = 2;
    const vector<double>& exponents1(cgbs11.GetExponents());
    vector<double> coefficients1;
    const vector<double>& coefficients11(cgbs11.GetCoefficients());
    const vector<double>& coefficients12(cgbs12.GetCoefficients());
    coefficients1.insert(coefficients1.end(), coefficients11.begin(), coefficients11.end());
    coefficients1.insert(coefficients1.end(), coefficients12.begin(), coefficients12.end());
    const vector<double>& mLfactors1(cgbs11.GetmLFactors());
    const ContractedGaussianBasisBlock block1(center1, L1, K1, B1, exponents1, coefficients1, mLfactors1);

    // Define GTO block2
    const Vec3D v2(0.3, 0.4, -1);
    const string basisinfo21("P 2 1.0\n5.3710000  0.0348780\n0.24310000  0.4400060\n");
    const string basisinfo22("P 2 1.0\n5.3710000  0.0435430\n0.24310000 -0.7870060\n");
    const ContractedGaussianBasisShell cgbs21(basisinfo21, v2);
    const ContractedGaussianBasisShell cgbs22(basisinfo22, v2);
    const Vec3D& center2(cgbs21.GetCenter());
    const int L2 = cgbs21.GetL();
    const int K2 = cgbs21.GetK();
    const int B2 = 2;
    const vector<double>& exponents2(cgbs21.GetExponents());
    vector<double> coefficients2;
    const vector<double>& coefficients21(cgbs21.GetCoefficients());
    const vector<double>& coefficients22(cgbs22.GetCoefficients());
    coefficients2.insert(coefficients2.end(), coefficients21.begin(), coefficients21.end());
    coefficients2.insert(coefficients2.end(), coefficients22.begin(), coefficients22.end());
    const vector<double>& mLfactors2(cgbs21.GetmLFactors());
    const ContractedGaussianBasisBlock block2(center2, L2, K2, B2, exponents2, coefficients2, mLfactors2);

    // Form Pair Data 1,2
    BlockPairData pBlockAB;
    ContractedGaussianBasisBlock::FormBlockPairData(block1, block2, pBlockAB);
```

```
// Define GTO block3
const Vec3D v3(0.6, -0.8, -1.3);
const string basisinfo31("P␣2␣1.0\n1.0400000␣␣0.0069760\n0.35710000␣␣0.0316690\n");
const string basisinfo32("P␣2␣1.0\n1.0400000␣␣0.0069760\n0.35710000␣␣-0.0316690\n");
const ContractedGaussianBasisShell cgbs31(basisinfo31, v3);
const ContractedGaussianBasisShell cgbs32(basisinfo32, v3);
const Vec3D& center3(cgbs31.GetCenter());
const int L3 = cgbs31.GetL();
const int K3 = cgbs31.GetK();
const int B3 = 2;
const vector<double>& exponents3(cgbs31.GetExponents());
vector<double> coefficients3;
const vector<double>& coefficients31(cgbs31.GetCoefficients());
const vector<double>& coefficients32(cgbs32.GetCoefficients());
coefficients3.insert(coefficients3.end(), coefficients31.begin(), coefficients31.end());
coefficients3.insert(coefficients3.end(), coefficients32.begin(), coefficients32.end());
const vector<double>& mLfactors3(cgbs31.GetmLFactors());
const ContractedGaussianBasisBlock block3(center3, L3, K3, B3, exponents3, coefficients3, mLfactors3);

// Define GTO block4
const Vec3D v4(-0.5, +0.5, 0.5);
const string basisinfo41("S␣2␣1.0\n1.0400000␣␣0.006760\n0.35710000␣␣0.031690\n");
const string basisinfo42("S␣2␣1.0\n1.0400000␣␣0.006760\n0.35710000␣␣-0.031690\n");
const ContractedGaussianBasisShell cgbs41(basisinfo41, v4);
const ContractedGaussianBasisShell cgbs42(basisinfo42, v4);
const Vec3D& center4(cgbs41.GetCenter());
const int L4 = cgbs41.GetL();
const int K4 = cgbs41.GetK();
const int B4 = 2;
const vector<double>& exponents4(cgbs41.GetExponents());
vector<double> coefficients4;
const vector<double>& coefficients41(cgbs41.GetCoefficients());
const vector<double>& coefficients42(cgbs42.GetCoefficients());
coefficients4.insert(coefficients4.end(), coefficients41.begin(), coefficients41.end());
coefficients4.insert(coefficients4.end(), coefficients42.begin(), coefficients42.end());
const vector<double>& mLfactors4(cgbs41.GetmLFactors());
const ContractedGaussianBasisBlock block4(center4, L4, K4, B4, exponents4, coefficients4, mLfactors4);

// Form Pair Data 3,4
BlockPairData pBlockCD;
ContractedGaussianBasisBlock::FormBlockPairData(block3, block4, pBlockCD);

// Form args
const GaussianTwoElectronCoulombIntegral::Args_TwoElectronCoulombIntegral args(pBlockAB, pBlockCD);

// Calculation.
const int LA = pBlockAB.LA;
const int LB = pBlockAB.LB;
const int LC = pBlockCD.LA;
const int LD = pBlockCD.LB;
const int nCRTShellA = pBlockAB.nCRTShellA;
const int nCRTShellB = pBlockAB.nCRTShellB;
const int nCRTShellC = pBlockCD.nCRTShellA;
const int nCRTShellD = pBlockCD.nCRTShellB;
const int BA = pBlockAB.nBlockA;
const int BB = pBlockAB.nBlockB;
const int BC = pBlockCD.nBlockA;
const int BD = pBlockCD.nBlockB;
const size_t ResultSize = nCRTShellA*nCRTShellB*nCRTShellC*nCRTShellD*BA*BB*BC*BD;

double crtint_contrA_contrB_contrC_contrD[ResultSize];
double derivfirstcrtint_contrA_contrB_contrC_contrD[ResultSize*9];
double derivsecondcrtint_contrA_contrB_contrC_contrD[ResultSize*45];

size_t BufferSize;
double* IntegralBuffer;

BufferSize = GaussianTwoElectronCoulombIntegral::GetTwoElectronCoulombIntegralDerivSecondBufferSize(LA,
     LB, LC, LD, BA, BB, BC, BD);
IntegralBuffer = (double*)malloc(sizeof(double)*BufferSize);
GaussianTwoElectronCoulombIntegral::TwoElectronCoulombIntegralDerivSecondBlock(args, IntegralBuffer,
     crtint_contrA_contrB_contrC_contrD,
        derivfirstcrtint_contrA_contrB_contrC_contrD, derivsecondcrtint_contrA_contrB_contrC_contrD);
free(IntegralBuffer);

printf("Two-electron␣Coulomb␣integrals:\n");
const char* firstderiv[] = {"d/dAx", "d/dAy", "d/dAz", "d/dBx", "d/dBy", "d/dBz", "d/dCx", "d/dCy", "d/
     dCz"};
for(int nr = 0; nr < 9; ++nr)
{
    printf("First␣derivatives:␣%s\n", firstderiv[nr]);
    for(int iBA = 0, idxD = 0; iBA < BA; ++iBA)
    {
        for(int iBB = 0; iBB < BB; ++iBB)
        {
            for(int iBC = 0; iBC < BC; ++iBC)
            {
                for(int iBD = 0; iBD < BD; ++iBD)
                {
                    printf("Block␣(%d,␣%d,␣%d,␣%d)\n", iBA, iBB, iBC, iBD);
                    printf("%32s\n", "AM");
                    CartesianAngularMomentum crtLA(LA);
                    for(int mLA = 0; mLA < nCRTShellA; ++mLA, ++crtLA)
                    {
                        CartesianAngularMomentum crtLB(LB);
                        for(int mLB = 0; mLB < nCRTShellB; ++mLB, ++crtLB)
                        {
                            CartesianAngularMomentum crtLC(LC);
```

```
                                for(int mLC = 0; mLC < nCRTShellC; ++mLC, ++crtLC)
                                {
                                    CartesianAngularMomentum crtLD(LD);
                                    for(int mLD = 0; mLD < nCRTShellD; ++mLD, ++crtLD, ++idxD)
                                    {
                                        printf("((%d,%d,%d)(%d,%d,%d)|(%d,%d,%d)(%d,%d,%d)):␣%12.8f\n",
                                                crtLA.GetLx(), crtLA.GetLy(), crtLA.GetLz(),
                                                crtLB.GetLx(), crtLB.GetLy(), crtLB.GetLz(),
                                                crtLC.GetLx(), crtLC.GetLy(), crtLC.GetLz(),
                                                crtLD.GetLx(), crtLD.GetLy(), crtLD.GetLz(),
                                                derivfirstcrtint_contrA_contrB_contrC_contrD[idxD+nr*
                                                     ResultSize]);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    const char* secondderiv[] = {
      "d2/dAxdAx", "d2/dAxdAy", "d2/dAxdAz",
      "d2/dAxdBx", "d2/dAxdBy", "d2/dAxdBz",
      "d2/dAxdCx", "d2/dAxdCy", "d2/dAxdCz",
      "d2/dAydAy", "d2/dAydAz", "d2/dAydBx",
      "d2/dAydBy", "d2/dAydBz", "d2/dAydCx",
      "d2/dAydCy", "d2/dAydCz", "d2/dAzdAz",
      "d2/dAzdBx", "d2/dAzdBy", "d2/dAzdBz",
      "d2/dAzdCx", "d2/dAzdCy", "d2/dAzdCz",
      "d2/dBxdBx", "d2/dBxdBy", "d2/dBxdBz",
      "d2/dBxdCx", "d2/dBxdCy", "d2/dBxdCz",
      "d2/dBydBy", "d2/dBydBz", "d2/dBydCx",
      "d2/dBydCy", "d2/dBydCz", "d2/dBzdBz",
      "d2/dBzdCx", "d2/dBzdCy", "d2/dBzdCz",
      "d2/dDxdCx", "d2/dDxdCy", "d2/dCxdCz",
      "d2/dDydCy", "d2/dCydCz", "d2/dCzdCz"
    };

    for(int nr = 0; nr < 45; ++nr)
    {
        printf("Second␣derivatives:␣%s\n", secondderiv[nr]);
        for(int iBA = 0, idxD = 0; iBA < BA; ++iBA)
        {
            for(int iBB = 0; iBB < BB; ++iBB)
            {
                for(int iBC = 0; iBC < BC; ++iBC)
                {
                    for(int iBD = 0; iBD < BD; ++iBD)
                    {
                        printf("Block␣(%d,␣%d,␣%d,␣%d)\n", iBA, iBB, iBC, iBD);
                        printf("%32s\n", "AM");
                        CartesianAngularMomentum crtLA(LA);
                        for(int mLA = 0; mLA < nCRTShellA; ++mLA, ++crtLA)
                        {
                            CartesianAngularMomentum crtLB(LB);
                            for(int mLB = 0; mLB < nCRTShellB; ++mLB, ++crtLB)
                            {
                                CartesianAngularMomentum crtLC(LC);
                                for(int mLC = 0; mLC < nCRTShellC; ++mLC, ++crtLC)
                                {
                                    CartesianAngularMomentum crtLD(LD);
                                    for(int mLD = 0; mLD < nCRTShellD; ++mLD, ++crtLD, ++idxD)
                                    {
                                        printf("((%d,%d,%d)(%d,%d,%d)|(%d,%d,%d)(%d,%d,%d)):␣%12.8f\n",
                                                crtLA.GetLx(), crtLA.GetLy(), crtLA.GetLz(),
                                                crtLB.GetLx(), crtLB.GetLy(), crtLB.GetLz(),
                                                crtLC.GetLx(), crtLC.GetLy(), crtLC.GetLz(),
                                                crtLD.GetLx(), crtLD.GetLy(), crtLD.GetLz(),
                                                derivsecondcrtint_contrA_contrB_contrC_contrD[idxD+nr*
                                                     ResultSize]);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    return 0;
}
```

3.6 Cartesian to Spherical Transformations

Related test files:

- `libreta/bin/test/AngularMomentum.cpp`
- `libreta/bin/test/Cartesian2Spherical.cpp`

Related header files:

- `libreta/bin/include/CartesianSphericalTransformation.h`

The integrals over Cartesian GTOs can be transformed to the ones over Spherical GTOs. If your preferred form of spherical GTOs is different the one used in `libreta`, then you will need to write your own program to do this job. Otherwise, simply use the ones provided here.

`OneIndexCartesianToSpherical(L, crt, sph)`

`TwoIndexCartesianToSpherical(LA, LB, crt, sph)`

`FourIndexCartesianToSpherical(LA, LB, LC, LD, crt, sph);`

where:

- `One/Two/FoureIndexCartesianToSpherical` performs the transformation for **a**, (**a**|**b**), and (**ab**|**cd**), respectively.
- `L, LA, LB, LC, LD` The corresponding angular momentum.
- `crt` stores the quantities in Cartesian GTO basis.
- `sph` stores the quantities in spherical GTO basis.

In priciple, these functions can carry out the transformations for any `L`. But for `FoureIndexCartesianToSpherical`, the transformation is most efficient when $L_a(L_a+1)/2+L_b < L_c(L_c+1)/2+L_d$. For the order of Cartesian and spherical angular momenta, please refer to `libreta/bin/test/AngularMomentum.cpp`.

Below is an example:

```
#include "CartesianAngularMomentum.h"
#include "SphericalAngularMomentum.h"
#include "CartesianSphericalTransformation.h"
#include <string>
#include <cstdlib>
#include <cstdio>

using namespace libreta;
using namespace std;

int main(int argc, char** argv)
{
    const int LA = 2;
    const int LB = 1;
    const int LC = 2;
    const int LD = 1;
    const int nCRTShellLA = CartesianAngularMomentum::GetNLLz(LA);
    const int nSphShellLA = SphericalAngularMomentum::GetNLLz(LA);
    const int nCRTShellLB = CartesianAngularMomentum::GetNLLz(LB);
    const int nSphShellLB = SphericalAngularMomentum::GetNLLz(LB);
    const int nCRTShellLC = CartesianAngularMomentum::GetNLLz(LC);
    const int nSphShellLC = SphericalAngularMomentum::GetNLLz(LC);
    const int nCRTShellLD = CartesianAngularMomentum::GetNLLz(LD);
    const int nSphShellLD = SphericalAngularMomentum::GetNLLz(LD);

    double crt[nCRTShellLA*nCRTShellLB*nCRTShellLC*nCRTShellLD];
    for(int i = 0; i < nCRTShellLA*nCRTShellLB*nCRTShellLC*nCRTShellLD; ++i) crt[i] = i/(i+0.5);
    double sph[nSphShellLA*nSphShellLB*nSphShellLC*nSphShellLD];
    CartesianSphericalTransformation::FourIndexCartesianToSpherical(LA, LB, LC, LD, crt, sph);

    SphericalAngularMomentum sphLA(LA);
    for(int mLA = 0, idxD = 0; mLA < nSphShellLA; ++mLA, ++sphLA)
    {
        SphericalAngularMomentum sphLB(LB);
        for(int mLB = 0; mLB < nSphShellLB; ++mLB, ++sphLB)
        {
            SphericalAngularMomentum sphLC(LC);
            for(int mLC = 0; mLC < nSphShellLC; ++mLC, ++sphLC)
            {
                SphericalAngularMomentum sphLD(LD);
                for(int mLD = 0; mLD < nSphShellLD; ++mLD, ++sphLD, ++idxD)
                {
                    printf("␣(%s%s|%s%s):␣%15.8f\n", sphLA.tostr().c_str(), sphLB.tostr().c_str(), sphLC.
                        tostr().c_str(), sphLD.tostr().c_str(),
                            sph[idxD]);
```

```
                }
            }
        }
    }

    return 0;
}
```

Appendix A

Version History

- Version 0.1 (Released on: Jan. 8, 2018): The first released version.