The LIBINT Programmer's Manual

Edward F. Valeev

Department of Chemistry, Virginia Tech, Blacksburg, Virginia 24061 USA

LIBINT Version 2.0.3 (stable)
Created on: August 25, 2015

1 Introduction

LIBINT library contains functions to compute many-body integrals over Gaussian functions which appear in electronic and molecular structure theories. LIBINT Version 2.0.3 (stable)[1] can currently compute several different types of integrals:

- Two-body Coulomb (electron repulsion) integrals (ERIs). This is by far the most common type of integrals in molecular structure theory. Two-, three-, and four-center integrals, and their geometrical derivatives, are supported.
- Two-body integrals which appear in explicitly correlated R12/F12 methods with Gaussian correlation factors.[2–4] All R12 methods, such as MP2-R12, contain terms in the wave function that depend on the interelectronic distances r_{ij} (hence the name). Appearance of several types of *two*-body integrals is due to the use of the approximate resolution of the identity to reduce three- and four-body integrals to products of simpler integrals.

A somewhat unusual feature of LIBINT library is that its source code is generated by a computer program, i.e., a *compiler*. The purpose of the LIBINT compiler is twofold:

- First, the compiler eliminates need for tedious writing, debugging, and optimizing the integrals code. Instead, a programmer provides high-level specification of operators and recurrence relations, and heuristics of how to apply the recurrence relations and the compiler takes care of the rest. This currently requires some high-level programming, in a domain-specific mini-language taylored for expressing recurrence relations. For example, the following Obara-Saika recurrence relation,[5]

$$\begin{aligned}(\mathbf{a}+1_i\ \mathbf{b}|\mathbf{cd})^{(m)} &= (\mathbf{PA})_i(\mathbf{ab}|\mathbf{cd})^{(m)} + (\mathbf{WP})_i(\mathbf{ab}|\mathbf{cd})^{(m+1)} \\ &+\frac{a_i}{2\zeta}\left[(\mathbf{a}-1_i\ \mathbf{b}|\mathbf{cd})^{(m)} - \frac{\rho}{\zeta}(\mathbf{a}-1_i\ \mathbf{b}|\mathbf{cd})^{(m+1)}\right] \\ &+\frac{b_i}{2\zeta}\left[(\mathbf{a}\ \mathbf{b}-1_i|\mathbf{cd})^{(m)} - \frac{\rho}{\zeta}(\mathbf{a}\ \mathbf{b}-1_i|\mathbf{cd})^{(m+1)}\right] \\ &+\frac{c_i}{2(\zeta+\eta)}(\mathbf{ab}|\mathbf{c}-1_i\ \mathbf{d})^{(m+1)} \\ &+\frac{d_i}{2(\zeta+\eta)}(\mathbf{ab}|\mathbf{c}\ \mathbf{d}-1_i)^{(m+1)} \end{aligned} \quad (1)$$

  is specified in a high-level form in C++ as shown in Listing 1.
- Second, the goal of the compiler is to make possible optimizing the code for existing and future computer hardware. For example, the ongoing trend in processors is to support single instruction multiple data (SIMD) parallelism. To effectively take advantage of the SIMD units The LIBINT compiler can generate vectorized code at user's request. As the hardware evolves, manual code reengineering can be avoided – only the compiler needs to be modified.

Listing 1: Example specification of an Obara-Saika recurrence relation in LIBINT compiler (see src/bin/libint2/vrr_11_twoprep_11.h). The corresponding mathematical expression is shown in Eq. (1)

```
auto ABCD_m = factory.make_child(a,b,c,d,m);
auto ABCD_mp1 = factory.make_child(a,b,c,d,m+1);
expr_ = Vector("PA")[dir] * ABCD_m + Vector("WP")[dir] * ABCD_mp1;

auto am1 = a - _1;
if (exists(am1)) {
  auto Am1BCD_m = factory.make_child(am1,b,c,d,m);
  auto Am1BCD_mp1 = factory.make_child(am1,b,c,d,m+1);
  expr_ += Vector(a)[dir] * Scalar("oo2z") * (Am1BCD_m - Scalar("roz") * Am1BCD_mp1);
}

auto bm1 = b - _1;
if (exists(bm1)) {
```

```
  auto ABm1CD_m = factory.make_child(a,bm1,c,d,m);
  auto ABm1CD_mp1 = factory.make_child(a,bm1,c,d,m+1);
  expr_ += Vector(b)[dir] * Scalar("oo2z") * (ABm1CD_m - Scalar("roz") * ABm1CD_mp1);
}

auto cm1 = c - _1;
if (exists(cm1)) {
  auto ABCm1D_mp1 = factory.make_child(a,b,cm1,d,m+1);
  expr_ += Vector(c)[dir] * Scalar("oo2ze") * ABCm1D_mp1;
}

auto dm1 = d - _1;
if (exists(dm1)) {
  auto ABCDm1_mp1 = factory.make_child(a,b,c,dm1,m+1);
  expr_ += Vector(d)[dir] * Scalar("oo2ze") * ABCDm1_mp1;
}
```

There are also drawbacks to the compiler-based LIBINT approach. First, the generated code can be fairly large and thus take a long time to compile. It is a relatively benign problem in practice. Also, the LIBINT compiler is a fairly complicated program; this limits the extent to which an average programmer can modify it.

LIBINT currently implements recursive schemes based on the Obara-Saika method,[5,6] and Head-Gordon-Pople[7] and Hamilton-Lindh variations thereof.[8,9] Other recurrence relations can be easily implemented as needed.

Unlike version 1, which came as three separate interdependent libraries, version 2 of LIBINT comes as a single library configured at code-generation time. The following features of the library can be configured:

- support for four-center ERI, including optional support for two- and three-center ERIs,
- derivative level for ERI,
- support for some special two-body integrals for explicitly-correlated integrals, e.g. of $[\hat{T}_1, \exp(-\alpha r_{12}^2)]$,
- optimization features (whether shell-sets of integrals can be unrolled, whether to perform Common Subexpression Elimination, etc.),
- vectorization features,
- algorithmic features (evaluation strategy),
- API features (name prefix, FLOP counter, whether to accumulate target integrals, floating-point type, shell ordering of Cartesian basis functions),
- shared library support.

Depending on its configuration, the library may not lack some capabilities.

2 Overview of LIBINT's API

(**Note**: Depending on configuration, LIBINT may support computation of several types of integrals. This section describes the parts of the interface that deal with theevaluatiin of four-center two-body Coulomb integrals. Additional notes on computing other types of integrals are provided in later sections.)

LIBINT library is a low-level C++ code. C linking convention is adopted to enable interoperability with other languages, such as C and FORTRAN. For notes on how to use LIBINT library from FORTRAN code, see subsection 7.

LIBINT API consists of 3 major components: type definitions, function and variable prototypes, and C preprocessor macros. The API is described in the following header files:

- `libint2.h` – main header file, it includes most other headers (not generated).
- `libint2_intrinsic_types.h` – architecture-specific definitions for long integer types (not generated).
- `libint2_types.h` – definitions for the integral evaluator types (generated).

- `libint2_params.h` – C preprocessor macros for library features (generated).
- `libint2_iface.h` – prototypes for functions and data as well as some misc macros (generated).
- `boys.h` – engines for computation of the Boys function and related quantities (not generated).

`libint2.h` (and, optionally, `boys.h`) is the only header that must be included explicitly in the user code.

The preprocessor macros provided by `libint2.h` are necessary so that the user code can access the library configuration parameters. For example, the library can be configured at the code generation time to support any finite angular momentum (quantum numbers) of the basis functions. The user code must still test whether the maximum angular momentum present in the basis set exceeds the library limit.

The preprocessor value macros provided by `libint2.h` are listed in Listing 2 The most important macro, `LIBINT2_REALTYPE`, specifies the floating-point type used by the library for all computations (usually, `double`). The next macro, `LIBINT2_MAX_VECLEN` specifies the maximum vector length supported by the library. For the scalar code this macro will be set to 1, whereas for the vectorized library this will have a value greater than 1. The next macro, `LIBINT2_API_PREFIX`, is used by the `LIBINT2_PREFIXED_NAME` function macro and should not be used explicitly (I list it here for completeness). The last three macros describe whether the library supports computation of the electron repulsion integrals, the maximum angular momentum of the basis functions supported for the ERI code (value of 3 corresponds to support of up to f functions, etc.), and the maximum order of the ERI derivatives which can be computed.

Listing 2: C preprocessor value macros provided by `libint2.h`. The angled brackets describe valid macro values.

```
#define LIBINT2_REALTYPE <C++ floating-point type>
#define LIBINT2_MAX_VECLEN <positive integer>
#define LIBINT2_API_PREFIX <string>

/* ERI-specific macros */
#define LIBINT2_SUPPORT_ERI <0 or 1>
#define LIBINT2_MAX_AM_ERI <nonnegative integer>
#define LIBINT2_DERIV_ERI_ORDER <nonnegative integer>
```

`LIBINT` API also specifies several functions and variables whose decralations are listed in Listing 3.

The first two functions perform static initialization and cleanup of the library. Thus `libint2_static_init` must be called before `LIBINT` is used and `libint2_static_cleanup` must be called after `LIBINT` is no longer needed (only one thread needs to call these functions).

The next two functions are used to initialize and cleanup the key data structure (*integral evaluator*) involved in the computation of the electron repulsion integrals. To initialize the integral evaluator, `libint2_init_eri` should be called with three arguments: 1) the pointer to the evaluator to be initialized (or to the first evaluator of an array of evaluators – see the notes on handling contracted Gaussians below); 2) the maximum angular momentum of basis functions this object will support (`max_am` will affect the memory requirements for the computation and therefore should be always set to the actual maximum value needed, not the maximum value supported by the library); 3) optional pointer to the scratch buffer which will be used to hold intermediate results. If the third argument is 0, then the call will dynamically allocate the needed space. If the user code needs to control memory allocation/deallocation, the scratch buffer needs to be allocated prior to the call. `libint2_need_memory_eri` can be used to compute the required size of the scratch buffer in units of `LIBINT2_REALTYPE`. Lastly, `libint2_build_eri` is a 4-dimensional array of pointers to functions that evaluate ERIs, e.g., `libint2_build_eri[1][0][2][0](&erieval)` will compute the $(ps|ds)$ set using evaluator `erieval` (`erieval` must have been initialized with `libint2_init_eri`).

Listing 3: `LIBINT` API functions and data. `N = LIBINT2_MAX_AM_ERI + 1`.

```
void libint2_static_init();
void libint2_static_cleanup();

/* ERI-specific API */
void libint2_init_eri(Libint_eri_t* libint, int max_am, LIBINT2_REALTYPE* buf);
void libint2_cleanup_eri(Libint_eri_t* libint);
size_t libint2_need_memory_eri(int max_am);
void (*libint2_build_eri[N][N][N][N])(Libint_eri_t *);
```

LIBINT API described so far is very simple. However, as you may have noticed, there has been no mention of where the basis set data is stored. This is because LIBINT does not maintain the basis set information. As part of LIBINT's philosophy to provide the leanest possible code, the user code is in charge of precomputing basis set data and Boys function values and then feeding it to the evaluator object of type Libint_eri_t. (Note that since currently all integral evaluations in LIBINT use the same evaluator object type, they are all just typedefs to the common type Libint_t, henceforth we will use both interchangeably).

Listing 4: Definition of the LIBINT integral evaluator type.

```
typedef struct {
  _aB_s__0__s__1___TwoERep_s__0__s__1___Ab__up_0[VECLEN];
  _aB_s__0__s__1___TwoERep_s__0__s__1___Ab__up_1[VECLEN];
  _aB_s__0__s__1___TwoERep_s__0__s__1___Ab__up_2[VECLEN];
  _aB_s__0__s__1___TwoERep_s__0__s__1___Ab__up_3[VECLEN];
  _aB_s__0__s__1___TwoERep_s__0__s__1___Ab__up_4[VECLEN];
  /* and so on until 4 * LIBINT2_MAX_AM_ERI */

  LIBINT2_REALTYPE WP_x[VECLEN], WP_y[VECLEN], WP_z[VECLEN];
  LIBINT2_REALTYPE WQ_x[VECLEN], WQ_y[VECLEN], WQ_z[VECLEN];
  LIBINT2_REALTYPE PA_x[VECLEN], PA_y[VECLEN], PA_z[VECLEN];
  LIBINT2_REALTYPE QC_x[VECLEN], QC_y[VECLEN], QC_z[VECLEN];
  LIBINT2_REALTYPE AB_x[VECLEN], AB_y[VECLEN], AB_z[VECLEN];
  LIBINT2_REALTYPE CD_x[VECLEN], CD_y[VECLEN], CD_z[VECLEN];

  LIBINT2_REALTYPE oo2z[VECLEN];
  LIBINT2_REALTYPE oo2e[VECLEN];
  LIBINT2_REALTYPE oo2ze[VECLEN];
  LIBINT2_REALTYPE roz[VECLEN];
  LIBINT2_REALTYPE roe[VECLEN];

  LIBINT2_REALTYPE* stack;   /* used internally */
  LIBINT2_REALTYPE* vstack;  /* used internally */
  LIBINT2_REALTYPE* targets[LIBINT2_MAX_NTARGETS_eri];
  int veclen;

  LIBINT2_UINT_LEAST64* nflops;
  int zero_out_targets;
} Libint_eri_t;
```

The definition of Libint_eri_t is shown in Listing 4. Libint_eri_t is a C structure, i.e., its members can be manipulated directly. This is done to allow direct manipulation of Libint_eri_t from Fortran and other languages. Note that only ERI-specific parts of the actual definition Libint_eri_t are shown in Listing 4. Currently, the same data structure is used for all types of evaluators (i.e. for ERI evaluators and Gaussian geminal type evaluators), i.e. Libint_eri_t contains also members which are only used for evaluation of Gaussian geminal integrals also. In the future the evaluator types will be generated automatically and will be specific to each type of computation.

Let's look in detail at the members of Libint_eri_t which must be precomputed before calling the relevant libint2_build_eri function:

- _aB_s__0__s__1___TwoERep_s__0__s__1___Ab__up_m – values of auxiliary primitive integrals $(\mathbf{00}|\mathbf{00})^{(m)}$ (Eq. (25)) for $0 \leq m \leq \lambda(\mathbf{a}) + \lambda(\mathbf{b}) + \lambda(\mathbf{c}) + \lambda(\mathbf{d}) + C$, where $C = 0$ when computing ERIs, $C = 1$ when computing first derivative ERIs, etc.
- AB_i, CD_i – cartesian components of vectors $\mathbf{AB} \equiv \mathbf{A} - \mathbf{B}$ and $\mathbf{CD} \equiv \mathbf{C} - \mathbf{D}$.
- PA_i, QC_i – cartesian components of vectors $\mathbf{PA} \equiv \mathbf{P} - \mathbf{A}$ and $\mathbf{QC} \equiv \mathbf{Q} - \mathbf{C}$.
- WP_i, WQ_i – cartesian components of vectors $\mathbf{WP} \equiv \mathbf{W} - \mathbf{P}$ and $\mathbf{WQ} \equiv \mathbf{W} - \mathbf{Q}$.
- oo2z – $\frac{1}{2\zeta}$
- oo2n – $\frac{1}{2\eta}$

- `oo2zn` – $\frac{1}{2(\zeta+\eta)}$
- `roz` – $\frac{\rho}{\zeta}$
- `ron` – $\frac{\rho}{\eta}$

Most of these quantities are simple to evaluate. Evaluation of the Boys function needed to compute the auxiliary integrals $(\mathbf{00}|\mathbf{00})^{(m)}$ can be fairly complicated.[10] `LIBINT` includes C++ classes that can evaluate Boys and related functions relatively efficiently (see `boys.h` header). An example of how to use `LIBINT` to compute Boys function is shown in Listing 5.

Listing 5: Computation of Boys function using `LIBINT`.

```
#include <boys.h>

// initialize Boys function engine to support m values up to mmax
// not thread-safe, should be constructed by main thread
libint2::FmEval_Chebyshev3 fmeval(mmax);

// double* Fm initialized somewhere else
// on return Fm[m], m=0..M, contains Fm(T)
// thread-safe provided each thread uses its own Fm
fmeval.eval(Fm, T, M);
```

Note that the evaluator object contains data that is depends on the Gaussian exponents. Hence for a shell set over contracted Gaussian functions data for each combination of primitive Gaussians will be kept in its own evaluator object, resulting in an array of evaluator objects necessary to evaluate contracted integrals. This is best illustrated by an example shown in Listing 6. This is perhaps the most significant change relative to the version 1 of the library, where the single evaluator object kept the data for all primitive combinations contributing to the given shell set.

Listing 6: Initialization and use of `LIBINT` with contracted basis functions.

```
libint2_static_init(); // static initialization (once per program)
std::vector<Libint_t> inteval(contrdepth4); // array of contrdepth4 evaluators
                                            // contrdepth4 = contrdepth^4, where contrdepth is
                                                 the max contraction
                                            // depth of Gaussians in the basis set

// initialize the array of evaluators .. this allocates the scratch array
// use libint2_need_memory_eri() to figure out how much memory is needed
libint2_init_eri(&inteval[0], // ptr to the first evaluator in the array
                 lmax,        // maximum angular momentum
                 0);

// loop over (contracted) Gaussian shells
for(int s0=0; s0<nshell; ++s0) {
  for(int s1=0; s1<nshell; ++s1) {
    for(int s2=0; s2<nshell; ++s2) {
      for(int s3=0; s3<nshell; ++s3) {

        // decide whether to evaluate the integral (uniqueness, magnitude, etc.)
        // ..

        // loop over each primitive combination for this shell set
        int p0123 = 0;
        for(int p0=0; p0<nprim[s0]; ++p0) {
          for(int p1=0; p1<nprim[s0]; ++p1) {
            for(int p2=0; p2<nprim[s0]; ++p2) {
              for(int p3=0; p3<nprim[s0]; ++p3) {

                // optionally screen out primitive combinations

                // compute primitive data and put into inteval[p0123]
                // ...
```

```
                ++p0123;
              }
            }
          }
        }
        // report the number of primitive combinations to libint
        inteval[0].contrdepth = p0123;

        // evaluate the contracted integral shell set
        libint2_build_eri[am[s0]][am[s1]][am[s2]][am[s3]](&inteval[0]);

        // grab the resulting shell set in inteval[0].targets[0][ ]
      }
    }
  }
}

libint2_cleanup_eri(&inteval[0]);
libint2_static_cleanup();
```

After the integrals have been built, they are placed somewhere in the scratch buffer. To recover their location, the array of pointers `targets` is provided, e.g., the computed ERI shell-set is located at `target[0]`. ERI evaluation produces only one shell-set of integrals, but other types of computations may produce several shell-sets of integrals at a time, e.g., usually all 12 derivative ERI integrals are computed at the same time. That's why `target` is an array of pointers, not a pointer.

Shell-sets of integrals contain integrals in "row major" order.[11] For example, if the number of functions in each shell is n_a, n_b, n_c, and n_d, respectively, then the integral $(ab|cd)$ is found at position $abcd = ((an_b+b)n_c+c)n_d+d$.

The rest of `Libint_eri_t` is used to control various aspects of its behavior:

- `veclength` is used in vectorized computation of integrals.
- `nflops` is used to count the total number of FLOPs (the library must have been configured with `--enable-flop-counter`).
- `zero_out_targets` is used to zero out the target integral buffers. This is only useful if `LIBINT` was configured with `--enable-accum-ints`.

Note that currently the `LIBINT` compiler minimizes the amount of code it generates by taking advantage of the permutational symmetry of the integrals. This means that only certain combinations of the angular momenta can be handled. In standard configuration `LIBINT` can evaluate a shell quartet $(\mathbf{ab}|\mathbf{cd})$ if $\lambda(\mathbf{a}) \geq \lambda(\mathbf{b})$, $\lambda(\mathbf{c}) \geq \lambda(\mathbf{d})$, and $\lambda(\mathbf{c}) + \lambda(\mathbf{d}) \geq \lambda(\mathbf{a}) + \lambda(\mathbf{b})$. (There is also the ordering used by `ORCA` program for which `LIBINT` can be configured — it will not be discussed here). If one needs to compute a quartet that doesn't conform the rule, e.g. of type $(pf|sd)$, permutational symmetry of integrals can be utilized to compute such quartet:

$$(pq|rs) = (pq|sr) = (qp|rs) = (qp|sr) = (rs|pq) = (rs|qp) = (sr|pq) = (sr|qp) \tag{2}$$

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

The final integrals that `LIBINT` computes are not normalized. The best way to include the normalization is to scale the auxiliary integrals $(\mathbf{00}|\mathbf{00})^{(m)}$ by the normalization factors. However, Gaussians in a shell of angular momentum > 1 have different normalization factors. Usually the convention is to unit-normalize only the functions which have all quanta along one Cartesian direction, e.g., d_{xx}, f_{xxx}, etc. Some programs (e.g., GAMESS) require all functions to be unit-normalized; this can be achieved by scaling the final integrals.

3 Using `LIBINT` to compute non-Coulomb integrals

As elegantly shown by Ahlrichs,[12] Obara-Saika and related schemes can also be used to compute many two-body integrals with spherically-symmetric kernels $f(r_{12})$. Such integrals appear in, for example, explicitly correlated

R12/F12 methods or in range-separated form of Kohn-Sham density functional theory. To compute such integrals, instead of the Coulomb-kernel $(\mathbf{00}|\mathbf{00})^{(m)}$ integrals related to the Boys function by Eq. 28 LIBINT needs to be fed the corresponding kernel-specific $(\mathbf{00}|\mathbf{00})^{(m)}$ integrals related to the the kernel-specific $G_m(\rho, T)$ function of Ahlrichs. Explicit expressions for G_m for several important kernels are given in Section 5 of Ref.[12]. For example, for the Coulomb kernel $G_m(\rho, T)$ are related to the Boys function as (Eq. (39) of Ahrlichs paper):

$$G_0(\rho, T) = \frac{2\pi}{\rho} F_m(T). \tag{3}$$

Hence Eq. (28) changes to

$$(\mathbf{00}|\mathbf{00})^{(m)} \quad = \quad G_m(\rho, \rho|\mathbf{PQ}|^2)\sqrt{\frac{\rho^3}{\pi^3}} S_{12} S_{34} C_1 C_2 C_3 C_4. \tag{4}$$

where the overlaps S_{12} and S_{34} are defined in Eqs. (26) and (27), **not** the overlaps from Ahlrichs' paper.

LIBINT supports computation of the G_m function for the important case of a contracted Gaussian kernel that appears in explicitly correlated methods:

$$f_k(r_{12}) = r_{12}^k \sum_{i=1}^{n} c_i \exp(-\alpha_i r_{12}^2), \quad k = -1, 0, 2. \tag{5}$$

Example code is shown in Listing 7.

Listing 7: Computation of the G_m function for the contracted Gaussian kernel using LIBINT.

```
#include <boys.h>

// initialize Gaussian Gm function engine to support m values up to mmax
// k specifies the exponent of r_{12} in the kernel (0, -1, and 2 supported)
libint2::GaussianGmEval<double, k> gmeval(mmax, 1e-15);  // 1e-15 is the desired precision

// vector of {exponent,coefficient} pairs = contracted Gaussian kernel
std::vector<std::pair<double,double> > gauss;

// double* Gm initialized somewhere else
// on return Gm[m], m=0..M, contains Gm(rho,T)
gmeval.eval(Gm, rho, T, M, gauss);
```

4 Using LIBINT to compute geometrical derivatives of integrals.

LIBINT can also evaluate geometrical derivatives of two-body integrals with respect to basis function positions. One shell set of four-center two-body integrals $(\mathbf{ab}|\mathbf{cd})$ has total of 12 first-order geometrical derivatives:

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

and $12 \times 12 = 144$ second-order derivatives, although only $\frac{12(12+1)}{2} = 78$ of those are unique because of permutation symmetry with respect to the order of taking the derivative:

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

The translational invariance of the integral can be used to eliminate derivatives involving one of the centers (as a matter of convention, LIBINT skips the third center, **C**). This allows to eliminate 3 of the 12 first-order derivatives.

$$\frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial C_i} = -\frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial A_i} - \frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial B_i} - \frac{\partial(\mathbf{ab}|\mathbf{cd})}{\partial D_i} \quad i \in \{x, y, z\} \tag{6}$$

and 33 of the 78 second-order derivatives

$$\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial C_j} = -\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial A_j} - \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial B_j} - \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial D_j} \quad i, j \in \{x, y, z\} \tag{7}$$

$$\begin{aligned}\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial C_i \partial C_j} = &\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial A_j} + \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial C_j} + \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial D_j} \\ &\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_j \partial B_i} + \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_i \partial C_j} + \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_i \partial D_j} \\ &\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_j \partial D_i} + \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_j \partial D_i} + \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial D_i \partial D_j} \quad i \le j \in \{x, y, z\}\end{aligned} \tag{8}$$

$$\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_i \partial C_j} = -\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_j \partial B_i} - \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_i \partial B_j} - \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_i \partial D_j} \quad i, j \in \{x, y, z\} \tag{9}$$

$$\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial C_i \partial D_j} = -\frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial A_i \partial D_j} - \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial B_i \partial D_j} - \frac{\partial^2(\mathbf{ab}|\mathbf{cd})}{\partial D_i \partial D_j} \quad i, j \in \{x, y, z\} \tag{10}$$

$$\tag{11}$$

Here's an easy way to understand how many derivatives are permutationally and translationally invariant: translational invariance allows to omit one center from consideration, hence we are left with 4-1=3 centers and $3 \times 3 = 9$ first-order geometrical derivatives; for the kth-order derivatives the order of differentiation does not matter, hence we are left with $(9 + k)!/9!k!$ unique derivatives, i.e. $9 \times 10/2 = 45$ second-order derivatives.

To minimize the operation count, LIBINT evaluates a complete set of unique shell-set derivatives at once. On return, `Libint_t::target[i]` will contain the `i`-th set of derivative integrals. For the first-order derivatives sets $i = 0..2$ corresponds to the derivative with respect to $A_x..A_z$, respectively, $i = 3..5$ — $B_x..B_z$, and $i = 6..8$ — $D_x..D_z$; derivatives with respect to coordinates $C_x..C_z$ can be recovered using Eq. (6). Similarly, for the second-order derivatives `Libint_t::target[ij]` will contain `ij`-th set of derivative integrals, where ij is the composite "upper-triangle" index: $ij = i \times (19 - i)/2 + (j - i), 0 \le i \le j < 9$. For example, `Libint_t::target[29]` thus contains the $\partial^2/\partial B_x \partial D_z$ derivative set ($ij = 29 \rightarrow i = 3, j = 8$).

The additional types of "compute" functions specific to the geometric derivatives are:

```
extern void (*libint2_build_eri1[5][5][5][5])(Libint_t *);
extern void (*libint2_build_eri2[4][4][4][4])(Libint_t *);
```

The former refers to functions which compute first derivative ERIs, and the second refers to functions which compute second derivative ERIs. The dimensions of each array are determined by the following 2 configure-time macros defined in `libint2_params.h`:

```
#define LIBINT2_MAX_AM_ERI1 5
#define LIBINT2_MAX_AM_ERI2 4
```

(the actual values ail depend on how the library was configured).

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

5 Using LIBINT to compute three- and two-center integrals

No notes yet — see `tests/eri/test_eri.cc` for a working example of how to compute 2- and 3-center integrals and their geometrical derivatives.

6 Example: using `libint` to compute four-center two-body Coulomb integrals

A C++ function that uses `LIBINT` to evaluate four-center two-body Coulomb integrals over *primitive* (non-contracted) Gaussians is shown in Listing 8. A standalone example of how to use `LIBINT` to compute the integrals over *contracted* Gaussians can be found in `doc/sample/test.cc` .

Listing 8: Using `LIBINT` from C++.

```
#include <iostream>
#include <algorithm>
#include <libint2.h>
#include <boys.h>

using namespace std;
using namespace libint2;

/** This function evaluates ERI over 4 primitive Gaussian shells.
    See doc/sample/test_eri.cc for an example of how to deal with
    contracted Gaussians.

    For simplicity, many details are omitted here, e.g. normalization.
  */
    void
    compute_eri(unsigned int am1, double alpha1, double A[3],
                unsigned int am2, double alpha2, double B[3],
                unsigned int am3, double alpha3, double C[3],
                unsigned int am4, double alpha4, double D[3]
               )
{
    // I will assume that libint2_static_init() has been called elsewhere!

    //------------------------------------------------------------------------------------
    // ------ the code in this section would usually be placed outside this function ------
    //        to occur once per calculation, not every time this function is called
    // - allocate ERI evaluator object
    Libint_eri_t erieval;
    const unsigned int max_am = max(max(am1,am2),max(am3,am4));
    libint2_init_eri(&erieval,max_am,0);
#if LIBINT_CONTRACTED_INTS
    // if have support for contracted integrals, set the contraction length to 1
    erieval.contrdepth = 1;
#endif
    // - initialize Boys function evaluator
    FmEval_Chebyshev3 fmeval(max_am);
    //------------------------------------------------------------------------------------

    //
    // Compute requisite data -- many of these quantities would be precomputed
    // for all nonnegligible shell pairs somewhere else
    //
    const double gammap = alpha1 + alpha2;
    const double Px = (alpha1*A[0] + alpha2*B[0])/gammap;
    const double Py = (alpha1*A[1] + alpha2*B[1])/gammap;
    const double Pz = (alpha1*A[2] + alpha2*B[2])/gammap;
    const double PAx = Px - A[0];
    const double PAy = Py - A[1];
    const double PAz = Pz - A[2];
    const double PBx = Px - B[0];
    const double PBy = Py - B[1];
    const double PBz = Pz - B[2];
    const double AB2 = (A[0]-B[0])*(A[0]-B[0])
                     + (A[1]-B[1])*(A[1]-B[1])
                     + (A[2]-B[2])*(A[2]-B[2]);

    erieval.PA_x[0] = PAx;
    erieval.PA_y[0] = PAy;
```

```
erieval.PA_z[0] = PAz;
erieval.AB_x[0] = A[0] - B[0];
erieval.AB_y[0] = A[1] - B[1];
erieval.AB_z[0] = A[2] - B[2];
erieval.oo2z[0] = 0.5/gammap;

const double gammaq = alpha3 + alpha4;
const double gammapq = gammap*gammaq/(gammap+gammaq);
const double Qx = (alpha3*C[0] + alpha4*D[0])/gammaq;
const double Qy = (alpha3*C[1] + alpha4*D[1])/gammaq;
const double Qz = (alpha3*C[2] + alpha4*D[2])/gammaq;
const double QCx = Qx - C[0];
const double QCy = Qy - C[1];
const double QCz = Qz - C[2];
const double QDx = Qx - D[0];
const double QDy = Qy - D[1];
const double QDz = Qz - D[2];
const double CD2 = (C[0]-D[0])*(C[0]-D[0])
                 + (C[1]-D[1])*(C[1]-D[1])
                 + (C[2]-D[2])*(C[2]-D[2]);

erieval.QC_x[0] = QCx;
erieval.QC_y[0] = QCy;
erieval.QC_z[0] = QCz;
erieval.CD_x[0] = C[0] - D[0];
erieval.CD_y[0] = C[1] - D[1];
erieval.CD_z[0] = C[2] - D[2];
erieval.oo2e[0] = 0.5/gammaq;

const double PQx = Px - Qx;
const double PQy = Py - Qy;
const double PQz = Pz - Qz;
const double PQ2 = PQx*PQx + PQy*PQy + PQz*PQz;
const double Wx = (gammap*Px + gammaq*Qx)/(gammap+gammaq);
const double Wy = (gammap*Py + gammaq*Qy)/(gammap+gammaq);
const double Wz = (gammap*Pz + gammaq*Qz)/(gammap+gammaq);

erieval.WP_x[0] = Wx - Px;
erieval.WP_y[0] = Wy - Py;
erieval.WP_z[0] = Wz - Pz;
erieval.WQ_x[0] = Wx - Qx;
erieval.WQ_y[0] = Wy - Qy;
erieval.WQ_z[0] = Wz - Qz;
erieval.oo2ze[0] = 0.5/(gammap+gammaq);
erieval.roz[0] = gammapq/gammap;
erieval.roe[0] = gammapq/gammaq;

double K1 = exp(-alpha1*alpha2*AB2/gammap);
double K2 = exp(-alpha3*alpha4*CD2/gammaq);
double pfac = 2*pow(M_PI,2.5)*K1*K2/(gammap*gammaq*sqrt(gammap+gammaq));

//
// evaluate Boys function F_m for all m in [0,am]
//
unsigned int am = am1 + am2 + am3 + am4;
double* F = new double[am+1];
fmeval.eval(F, PQ2*gammapq, am);

// (00|00)^m = pfac * F_m
erieval.LIBINT_T_SS_EREP_SS(0)[0] = pfac*F[0];
erieval.LIBINT_T_SS_EREP_SS(1)[0] = pfac*F[1];
erieval.LIBINT_T_SS_EREP_SS(2)[0] = pfac*F[2];
erieval.LIBINT_T_SS_EREP_SS(3)[0] = pfac*F[3];
erieval.LIBINT_T_SS_EREP_SS(4)[0] = pfac*F[4];
// etc.

// compute ERIs
libint2_build_eri[am1][am2][am3][am4](&erieval);
```

```
    // Print out the integrals
    const double* eri_shell_set = erieval.targets[0];
    const unsigned int n1 = (am1 + 1) * (am1 + 2)/2;
    const unsigned int n2 = (am2 + 1) * (am2 + 2)/2;
    const unsigned int n3 = (am3 + 1) * (am3 + 2)/2;
    const unsigned int n4 = (am4 + 1) * (am4 + 2)/2;
    for(int a=0; a<n1; a++) {
      for(int b=0; b<n2; b++) {
        for(int c=0; c<n3; c++) {
          for(int d=0; d<n4; d++) {
            cout << "a = " << a
                 << "b = " << b
                 << "c = " << c
                 << "d = " << d
                 << "(ab|cd) = " << *eri_shell_set;
            ++eri_shell_set;
          }
        }
      }
    }

    // ------ like the code at the beginning, this usually goes outside this function ------
    libint2_cleanup_eri(&erieval);
}
```

To see how to use LIBINT for efficient computation of integrals over contracted basis function refer to the sample code found in `tests/eri` directory.

7 Notes on using LIBINT from Fortran

Although LIBINT source is written in C++, it should be possible to use the library from Fortran programs. (Un)fortunately, I am not a Fortran programmer. Thus I can only provide general guidelines here.

One of the main issues is the number of Fortran standards available. The most recent standard, Fortran 2003, seems to have the best support for interoperability with C programs, but its compiler support is stil lacking. Unfortunately, the most popular standard, Fortran 77, is also the most restrictive.

In general, C functions can be easily called from Fortran programs, but sharing data structures is not straightforward. Thus the main culprit is how to modify `Libint_eri_t` objects from Fortran. Fortran 2003 provides direct support for binding C data structures to Fortran types. Older Fortran standards can access C data structures indirectly, via common blocks. An (non-working) example of how a Fortran subroutine can manipulate `Libint_eri_t` is shown in Listing 9. The common block `erieval` referened in that Listing is created by declaring in C++ a global variable as follows:

```
extern Libint_eri_t erieval;
```

Listing 9: Accesing `Libint_eri_t` structure from a Fortran code.

```
      c assuming that LIBINT2_REALTYPE is 8-bytes long
      real(8) F0(1)
      real(8) F1(1)
      real(8) F2(1)
      real(8) F3(1)
      real(8) F4(1)
      etc.
      real(8) WP_x(1), WP_y(1), WP_z(1)
      real(8) WQ_x(1), WQ_y(1), WQ_z(1)
      real(8) PA_x(1), PA_y(1), PA_z(1)
      real(8) QC_x(1), QC_y(1), QC_z(1)
      real(8) AB_x(1), AB_y(1), AB_z(1)
      real(8) CD_x(1), CD_y(1), CD_z(1)
      real(8) oo2z(1), oo2e(1), oo2ze(1), roz(1), roe(1)
      c in 64-bit environment pointers are 8-bytes long
```

```
c assuming LIBINT2_MAX_NTARGETS is 10
integer(8) targets(10)
integer(4) veclength
integer(8) nflops
integer(4) zero_out_targets
c common erieval represents the C object erieval
common/erieval/ F0, F1, ... , WP_x, WP_y, etc.

c now can access elements of erieval
AB_x[0] = Ax[0] - Bx[0]
etc.
```

The caveat of accessing C data structures from Fortran programs is that the members of the data structure must be declared in the common block in the exact order in which they appear in the definition of `Libint_eri_t`. The actual definition of `Libint_eri_t` in `libint2_types.h` must always be consulted.

Calling `LIBINT` functions from Fortran should be straightforward. Using the function pointer array `libint2_build_eri` is probably not feasible in older Fortran standards, but perhaps can be accomplished in Fortran 2003. Actual function names must be used instead, i.e., a C++ expression

```
libint2_build_eri[1][0][2][0](&erival);
```

will be replaced with a Fortran expression (I'm not sure how to pass pointer to erieval to the function from Fortran!)

```
_aB_p__0__d__1___TwoERep_s__0__s__1___Ab__up_0()
```

where `_aB_p__0__d__1___TwoERep_s__0__s__1___Ab__up_0` is the name of the function to which `libint2_build_eri[1][0][2][0]` points.

Please send in your comments on how to actually make `LIBINT` work from Fortran.

Appendices

A Notation

Following Obara and Saika,[5] we write an *unnormalized primitive Cartesian* Gaussian function centered at $\mathbf{A}$ as

$$\begin{aligned}\phi(\mathbf{r};\zeta,\mathbf{n},\mathbf{A}) &= (x-A_x)^{n_x}(y-A_y)^{n_y}(z-A_z)^{n_z} \\ &\quad \times \exp[-\zeta(\mathbf{r}-\mathbf{A})^2]\ , \end{aligned} \tag{12}$$

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

A set of $(\lambda(\mathbf{n}) + 1)(\lambda(\mathbf{n}) + 2)/2$ functions with the same $\lambda(\mathbf{n})$, ζ, and centered at the common center but with different $\mathbf{n}$ form a *Cartesian shell*, or just a *shell*. For example, an s shell ($\lambda = 0$) has one function, a p shell ($\lambda = 1$) – 3 functions, etc. There is no unique choice for the order of functions in shells. The standard `LIBINT` ordering is:

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

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

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

Other shell orderings are supported as well, e.g., those employed by the `intv3` engine in the `MPQC` program, the ordering used in the `GAMESS` program, or that in the `ORCA` program. These can be specified when the library is generated (see the `--with-cartgauss-ordering` configure flag). Support of a new ordering is trivial to implement. If your program relies on an ordering different from the above, please contact the author of `LIBINT`.

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

$$N(\zeta,\mathbf{n}) = \left[\left(\frac{2}{\pi}\right)^{3/4}\frac{2^{(\lambda(\mathbf{n}))}\zeta^{(2\lambda(\mathbf{n})+3)/4}}{[(2n_x-1)!!(2n_y-1)!!(2n_z-1)!!]^{1/2}}\right] \tag{13}$$

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

$$\begin{aligned}\phi(\mathbf{r};\boldsymbol{\zeta},\mathbf{C},\mathbf{n},\mathbf{A}) &= (x-A_x)^{n_x}(y-A_y)^{n_y}(z-A_z)^{n_z} \\ &\quad\times\sum_{i=1}^{M} C_i \exp[-\zeta_i(\mathbf{r}-\mathbf{A})^2]\ ,\end{aligned} \tag{14}$$

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

$$C_i = c_i N(\zeta_i,\mathbf{n}) \tag{15}$$

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

$$\frac{\pi^{3/2}(2n_x-1)!!(2n_y-1)!!(2n_z-1)!!}{2^{\lambda(\mathbf{n})}}\sum_{i=1}^{M}\sum_{j=1}^{M}\frac{C_iC_j}{(\zeta_i+\zeta_j)^{\lambda(\mathbf{n})+3/2}} = 1 \tag{16}$$

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

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

$$\int \phi(\mathbf{r}_1;\zeta_a,\mathbf{a},\mathbf{A})\phi(\mathbf{r}_2;\zeta_c,\mathbf{c},\mathbf{C})\hat{O}(\mathbf{r}_1,\mathbf{r}_2)\phi(\mathbf{r}_1;\zeta_b,\mathbf{b},\mathbf{B})\phi(\mathbf{r}_2;\zeta_d,\mathbf{d},\mathbf{D})d\mathbf{r}_1 d\mathbf{r}_2 \equiv (\mathbf{ab}|\hat{O}|\mathbf{cd}) \equiv \langle\mathbf{ac}|\hat{O}|\mathbf{bd}\rangle \tag{17}$$

A set of integrals $\{(\mathbf{ab}|\hat{O}(\mathbf{r}_1,\mathbf{r}_2)|\mathbf{cd})\}$ over all possible combinations of functions $\mathbf{a}\in$ ShellA, $\mathbf{b}\in$ ShellB, etc. will be termed a *shell-set*, or simply a *set*, of integrals. For example, a $(ps|sd)$ set consists of $3\times1\times1\times6=18$ integrals.

The following definitions have been used throughout this work:

$$\zeta = \zeta_a + \zeta_b \tag{18}$$
$$\eta = \zeta_c + \zeta_d \tag{19}$$
$$\rho = \frac{\zeta\eta}{\zeta+\eta} \tag{20}$$
$$\mathbf{P} = \frac{\zeta_a\mathbf{A} + \zeta_b\mathbf{B}}{\zeta} \tag{21}$$
$$\mathbf{Q} = \frac{\zeta_c\mathbf{C} + \zeta_d\mathbf{D}}{\eta} \tag{22}$$
$$\mathbf{W} = \frac{\zeta\mathbf{P} + \eta\mathbf{Q}}{\zeta+\eta} \tag{23}$$

The Boys function is defined as

$$F_m(T) = \int_0^1 dt\ t^{2m}\ \exp(-Tt^2) \tag{24}$$

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

$$(\mathbf{00}|\mathbf{00})^{(m)} = 2F_m(\rho|\mathbf{PQ}|^2)\sqrt{\frac{\rho}{\pi}}S_{12}S_{34} \tag{25}$$

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

$$S_{12} = \left(\frac{\pi}{\zeta}\right)^{3/2}\exp\left(-\frac{\zeta_a\zeta_b}{\zeta}|\mathbf{AB}|^2\right) \tag{26}$$
$$S_{34} = \left(\frac{\pi}{\eta}\right)^{3/2}\exp\left(-\frac{\zeta_c\zeta_d}{\eta}|\mathbf{CD}|^2\right) \tag{27}$$

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

$$(\mathbf{00}|\mathbf{00})^{(m)} = 2F_m(\rho|\mathbf{PQ}|^2)\sqrt{\frac{\rho}{\pi}}S_{12}S_{34}C_1C_2C_3C_4 \tag{28}$$

where the coefficients C_a, C_b, C_c, and C_d are normalization-including contraction coefficients (Eqs. (15) and (16)) for the first basis function out of each respective shell in the target shell of integrals.

References

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

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

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

[4] B. J. Persson and P. R. Taylor. Accurate quantum-chemical calculation: The use of gaussian-type geminal functions in the treatment of electron correlation. *J. Chem. Phys.*, 105:5915, 1996.

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

[6] S. Obara and A. Saika. General recurrence formulas for molecular integrals over cartesian gaussian functions. *J. Chem. Phys.*, 89:1540, 1988.

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

[8] T. P. Hamilton and H. F. Schaefer. New variations in two-electron integral evaluation in the context of direct scf procedures. *Chem. Phys.*, 150:163, 1991.

[9] R. Lindh, U. Ryu, and B. Liu. The reduced multiplication scheme of the rys quadrature and new recurrence relations for auxiliary function based 2-electron integral evaluation. *J. Chem. Phys.*, 95:5889, 1991.

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

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

[12] R. Ahlrichs. A simple algebraic derivation of the Obara-Saika scheme for general two-electron interaction potentials. *Phys. Chem. Chem. Phys.*, 8(26):3072, 2006.