Received: 18 February 2022 | Revised: 22 April 2022 | Accepted: 28 April 2022

DOI: 10.1002/jcc.26942

SOFTWARE NOTE

Journal of COMPUTATIONAL CHEMISTRY WILEY

The SHARK integral generation and digestion system

Frank Neese

Department of Molecular Theory and Spectroscopy, Max Planck Institut für Kohlenforschung, Mülheim an der Ruhr, Germany

Correspondence
Frank Neese, Department of Molecular Theory and Spectroscopy, Max Planck Institut für Kohlenforschung, D-45470, Mülheim an der Ruhr, Germany.
Email: neese@kofo.mpg.de

Funding information
Max-Planck-Gesellschaft

Abstract

In this paper, the SHARK integral generation and digestion engine is described. In essence, SHARK is based on a reformulation of the popular McMurchie/Davidson approach to molecular integrals. This reformulation leads to an efficient algorithm that is driven by BLAS level 3 operations. The algorithm is particularly efficient for high angular momentum basis functions (up to $L = 7$ is available by default, but the algorithm is programmed for arbitrary angular momenta). SHARK features a significant number of specific programming constructs that are designed to greatly simplify the workflow in quantum chemical program development and avoid undesirable code duplication to the largest possible extent. SHARK can handle segmented, generally and partially generally contracted basis sets. It can be used to generate a host of one- and two-electron integrals over various kernels including, two-, three-, and four-index repulsion integrals, integrals over Gauge Including Atomic Orbitals (GIAOs), relativistic integrals and integrals featuring a finite nucleus model. SHARK provides routines to evaluate Fock like matrices, generate integral transformations and related tasks. SHARK is the essential engine inside the ORCA package that drives essentially all tasks that are related to integrals over basis functions in version ORCA 5.0 and higher. Since the core of SHARK is based on low-level basic linear algebra (BLAS) operations, it is expected to not only perform well on present day but also on future hardware provided that the hardware manufacturer provides a properly optimized BLAS library for matrix and vector operations. Representative timings and comparisons to the Libint library used by ORCA are reported for Intel i9 and Apple M1 max processors.

KEYWORDS
density functional theory, Hartree–Fock theory, integral algorithms, quantum chemistry

1 | INTRODUCTION

Generating and digesting integrals over the chosen basis set (atomic orbitals, AOs) is central to every quantum chemistry package. The efficiency with which these tasks are performed determine the overall turnaround time, at least in the case of Hartree–Fock (HF) and density functional theory (DFT). Furthermore, the ease with which the integrals can be generated and accessed inside the given program package plays a central role for the efficiency of the development process and the time it takes a programmer to get acquainted with the programming environment that they enter.

While it has been understood for a long time that correct atomic radial functions should have a cusp at the nucleus and decay exponentially,[1] the resulting multi-center integrals are too difficult to evaluate efficiently using analytic methods.[2,3] Hence, the overwhelming majority of calculations are nowadays based on contracted

Gaussian basis functions. In a contracted basis function, several primitive Gaussians are linearly combined into the actual basis function that is then used in the molecular calculation.[4] Contractions can be performed in a number of ways. In the most elementary form, "segmented" contractions consist of linear combinations in which a given primitive only contributes to one contracted basis function. Consequently, there are no duplicate primitives in the set. This construction principle is followed in the "def2" family of basis sets[5–8] in which all primitive exponents have been carefully optimized through direct energy minimization. Alternatively, one could expand all basis function on a given atom for a given angular momentum in the same set of primitives. This leads to a rectangular matrix of expansion coefficients that described the transformation from primitive space to the contracted space. This principle is followed in the construction of atomic natural orbital (ANO) basis sets.[9] Finally, an intermediate form of contraction is followed in the correlation consistent (cc) family of basis sets.[10–12] Here, the low angular momentum (s,p) orbitals that are occupied in the free atom are described by generally contracted basis functions to which uncontracted polarization of diffuse functions are added. Such basis sets may be characterized as "partially generally contracted" (PGC).

Clearly, the type of contraction has a major influence on the integral generation and digestion strategy. The cost of generating integrals over contracted Gaussians scale as the fourth power of the "contraction depth" (the number of primitives in each contracted Gaussians).

Given the importance of the subject, a significant number of algorithms has been developed for the calculation of one- and two-electron integrals over Gaussian basis functions. While a full review of the entire history is outside the scope of this article, I want to mention early works of Pople and Hehre (PH) using an "axis switch" technique,[13] and later developments by Head–Gordon and Pople (HGP),[14] the PRISM algorithm of Gill and Pople,[15,16] the Hermite polynomial algorithm of McMurchie and Davidson (MD),[17,18] the Rys polynomial (RP) approach,[19] the Obara–Saika (OS) scheme[20,21] and the accompanying coordinate expansion (ACE).[22] All of these schemes, except the PH scheme (that only works on low angular momentum (s,p) basis functions), is based on various recursion relations. In addition, to the generation of the integrals over Cartesian Gaussian functions, most programs actually work with spherical harmonic Gaussian functions in which case an additional transformation step from the Cartesian to the spherical harmonics basis is necessary.

Much effort has been devised to find algorithms that minimize the floating point (FLOP) count of the integral generation algorithm. However, a FLOP count optimal algorithm may not lead to the best performing algorithm since there are also memory operation (MOP) count considerations as well as the cost of logic in the innermost loops that may lead to performance deterioration due to cache misses.[23] Indeed, in his excellent review on the matter, Lindh concluded that "to a large extend the efficiency of a computer code is a result of the care taken during the implementation stage and not due to the particular method selected for implementation."[24]

In addition to these algorithmic considerations, one has to take into account the specific construction of the hardware that the calculations run on, which obviously is a quickly moving target. As a consequence, it is difficult to design an algorithm that is practically guaranteed to perform optimally not only on present day, but also on future hardware.

It is probably fair to say that the integral generation package is a at the heart of each large quantum chemistry package. While many programs have their individual, purpose specific written integral code, a few integral packages have been made publicly available and found their way into several program packages in current use. Modern installments of such libraries include the Libint,[25] and Libcint[26] codes.

In this paper, a new integral package will be described that was given the name SHARK (no acronym implied). SHARK exists inside the ORCA suite of quantum chemistry programs[27–30] and uses some of the fundamental ORCA infrastructure, but is otherwise largely independent of ORCA. SHARK was a major new feature in the release of ORCA 5.0 and drives most integral-related tasks in all ORCA versions after 5.0. Hence, it is timely to document the underlying strategies that are operative inside SHARK. This is documentation contains a limited amount of new integral generation science. However, it is hopefully still useful to describe the programming strategies followed in SHARK since they lead to a drastic simplification of the development workflow while ensuring a high degree of consistency and efficiency.

2 | THEORY

In order for the article to be self-contained, we briefly reiterate the machinery necessary for generating molecular integrals with the MD algorithm before focusing on the particular features of SHARK.

The primitive basis functions considered in this work are of the Cartesian Gaussian form.[4]

$$G^{A}_{ijk}(\alpha) = x^{i}_{A} y^{j}_{A} z^{l}_{A} \exp\left(-\alpha r^{2}_{A}\right) \tag{1}$$

Here x_A, y_A, z_A represent the three components of the electronic position vector $\boldsymbol{r}$ relative to nucleus A at position $\boldsymbol{R}_A = (X_A, Y_A, Z_A)$ such that $\boldsymbol{r}_A = \boldsymbol{r} - \boldsymbol{R}_A$ ($r_A = |\boldsymbol{r}_A|$) and α is the Gaussian exponent. For a given angular momentum $l = i + j + k$ there are $\frac{(l+1)(l+2)}{2}$ combinations of i, j, k that make up a shell of Cartesian Gaussian basis functions. Contracted Gaussian basis functions are formed as linear combinations.

$$G^{A}_{ijk} = \sum_{p} d_p G^{A}_{ijk}(\alpha_p) \tag{2}$$

With expansion coefficients $\boldsymbol{d}$ usually chosen such that the overall function is normalized $\langle G^{A}_{ijk} | G^{A}_{ijk} \rangle = 1$. In the case of general contraction, $\boldsymbol{d}$ is not a vector of length "contraction depth" but a matrix of dimension "contraction depth" × "number of basis functions on that atom with the given angular momentum l."

The transformation from Cartesian Gaussian to spherical harmonic Gaussian basis functions can be written:

$$G^{A}_{lm}=\sum_{ijk} t_{ijk,lm} G^{A}_{ijk} \tag{3}$$

The transformation matrix $\boldsymbol{t}$ only depends on the angular momentum but not on details of the basis functions or its center. Hence, t-matrices are readily pretabulated or made on the fly using analytic formulae.[31]

Given two primitive Gaussian s-functions, the Gaussian product theorem states that[4]:

$$G^{A}_{000}(\alpha)G^{B}_{000}(\beta)=K_{AB}\exp\left(-pr_{p}^{2}\right) \tag{4}$$

With

$$\boldsymbol{R}_{P}=\frac{1}{\alpha+\beta}(\alpha\boldsymbol{R}_{A}+\beta\boldsymbol{R}_{B}) \tag{5}$$

$$\boldsymbol{r}_{p}=\boldsymbol{r}-\boldsymbol{R}_{p} \tag{6}$$

$$p=\alpha+\beta \tag{7}$$

$$K_{AB}=\exp\left(-qR_{AB}^{2}\right) \tag{8}$$

$$R_{AB}=\boldsymbol{R}_{A}-\boldsymbol{R}_{B} \tag{9}$$

$$q=\frac{\alpha\beta}{\alpha+\beta} \tag{10}$$

For products of general angular momentum basis functions, McMurchie and Davidson[9,17,18] have proposed to expand the complicated polynomial that arises from multiplying $x_{A}^{i}y_{A}^{j}z_{A}^{l}$ and $x_{B}^{i'}y_{B}^{j'}z_{B}^{l'}$ in terms of Hermite polinomials.[32]

$$G^{A}_{ijk}(\alpha)G^{B}_{i'j'k'}(\beta)=\sum_{t=0}^{i+i'}\sum_{u=0}^{j+j'}\sum_{v=0}^{k+k'}E_{t}^{ii'}E_{u}^{jj'}E_{v}^{kk'}\Lambda_{t}\Lambda_{u}\Lambda_{v} \tag{11}$$

This expansion emphasizes that in the Cartesian representations Gaussians factorize in the three Cartesian directions. In this equation, the E-coefficients represent the expansion coefficients for the Hermite Gaussian function,

$$\Lambda_{t}(x)=H_{t}(x)\exp\left(-pr_{p}^{2}\right) \tag{12}$$

With the Hermite polynomial,

$$H_{t}(x)=(-1)^{t}\exp\left(x^{2}\right)\frac{d^{t}}{dx^{t}}\exp\left(x^{2}\right) \tag{13}$$

For the expansion coefficients MD derived the recursion relation:

$$E_{t}^{i+1j}=\frac{1}{2p}E_{t-1}^{ij}+(X_{P}-X_{A})E_{t}^{ij}+(t+1)E_{t+1}^{ij} \tag{14}$$

The recursion is terminated at

$$E_{0}^{00}=1 \tag{15}$$

Alternatively, one can choose $E_{0}^{00}=K_{AB}$ for one of the Cartesian directions or include products of contraction coefficients. This is just a matter of taste and program organization.

For a quadruple of Gaussian shells, MD then derived the formula:

$$\begin{aligned}&\left(G^{A}_{ijk}(\alpha)G^{B}_{i'j'k'}(\beta)\middle|r_{12}^{-1}\middle|G^{C}_{lmn}(\gamma)G^{D}_{l'm'n'}(\delta)\right)\\&=\frac{C}{pq\sqrt{p+q}}\sum_{tuv}E_{t}^{ii'}E_{u}^{jj'}E_{v}^{kk'}\sum_{t'u'v'}E_{t'}^{ll'}E_{u'}^{mm'}E_{v'}^{nn'}(-1)^{t'+u'+v'}R_{t+t',u+u',v+v'}\end{aligned} \tag{16}$$

The R coefficients represent the electron–electron repulsion integrals over a pair of Hermite Gaussians. For these integrals, MD derived the recursion formula:

$$R_{t+1,u,v}=tR_{t-1,u,v}+(X_{P}-X_{Q})R_{tuv}^{(n+1)} \tag{17}$$

$$R_{tuv}^{(n+1)}=(-2p)^{n}F_{n}\left(\frac{pq}{p+q}R_{PQ}^{2}\right) \tag{18}$$

Here are $\boldsymbol{R}_{P}$ and $\boldsymbol{R}_{Q}$ are the intermediate points from the GPT for the bra and ket, respectively and $p=\alpha+\beta, q=\gamma+\delta$ are the respective effective exponents. The function F_{n} is the well-known incomplete Gamma function:

$$F_{n}=\int_{0}^{1}\exp\left(-xt^{2}\right)t^{2n}dt \tag{19}$$

Finally, the constant in front of the integral is.

$$C=8\left(\pi^{\frac{5}{2}}\right) \tag{20}$$

The direct evaluation of Equation (16) is tedious. It involves a sixfold summation creating short loops and a lot of computational overhead. Consequently, the MD algorithm, despite its mathematical convenience and elegance, has fallen in disfavor with quantum chemists.

The essence of the SHARK algorithm is to rewrite Equation (16) in a more computationally favorable way. Thus, one can reinterpret the indices tuv and $t'u'v'$ as compound indices and treat all members of the two shell pairs simultaneously to arrive at:

$$\left(G^{A}_{ijk}(\alpha)G^{B}_{i'j'k'}(\beta)\middle|r_{12}^{-1}\middle|G^{C}_{lmn}(\gamma)G^{D}_{l'm'n'}(\delta)\right)=\left(\boldsymbol{E}^{+}\boldsymbol{RE}\right)_{ijk;i'j'k',lmn;l'm'n'} \tag{21}$$

In this equation, the E- and R-coefficients have been interpreted as matrices:

$$\sqrt{C}\boldsymbol{E}(tuv,ijk;i'j'k') \tag{22}$$

$$\frac{1}{pq\sqrt{p+q}}\boldsymbol{R}(tuv,t'u'v')(-1)^{t'+u'+v'} \tag{23}$$

Which is more readily recognized, if we lump together indices $ijk=\mu, i'j'k'=\nu$ and $tuv=r$ the matrices read $\boldsymbol{E}(\mu\nu,r)$ and $\boldsymbol{R}(r,s)$ and the BLAS level 3 operations that generate a batch of integrals is readily implied. There are two key aspects in rewriting the MD equations this way: (1) there is a factorization of the integrals into three parts that can be exploited in an optimal algorithm and (2) the factorization generates batches of integrals by BLAS level 3 operations which drive the computer at peak performance using vendor supplied and standardized BLAS libraries.

If one goes down that route, one might as well include the transformation to the spherical Harmonics basis in the definition of the $\boldsymbol{E}$ -matrices and write:

$$\boldsymbol{E}(tuv,\mu\nu)=\sum_{tuv,t'u'v'} t_{ijk,l_\mu m_\mu} t_{i'j'k',l_\nu m_\nu} \boldsymbol{E}(ijk;i'j'k',tuv) \tag{24}$$

This clearly reduces the FLOP count and the angular momentum scaling since there are only $2l+1$ functions in a spherical harmonics Gaussian shell of angular momentum l as opposed to $\frac{(l+1)(l+2)}{2}$ members of a shell in the Cartesian basis.

```
$\boldsymbol{I}=\boldsymbol{0}$
Loop Bra primitives
    $\boldsymbol{T}=\boldsymbol{0}$
    Loop Ket primitives
        Calculate $\boldsymbol{R}$
      $\boldsymbol{T}+=\boldsymbol{R}\boldsymbol{E}_{ket}$
    End Ket
    $\boldsymbol{I}+=\boldsymbol{E}_{bra}^{+}\boldsymbol{T}$
End Bra
```

SCHEME 1 Pseudocode for the evaluation of two-electron integrals using the SHARK algorithm. $\boldsymbol{E}_{\text{bra;ket}}$ are the E-coefficients. $\boldsymbol{I}$ represents the final integral batch.

TABLE 1 Number of contributing Hermite products to the Cartesian Gaussian product space spanned by two Gaussians of angular momentum l_i,l_j

$l_i\backslash l_j$	0	1	2	3	4
0	1	4	10	20	35
1		10	20	35	56
2			35	56	84
3				84	120
4					165

The factorization offered in Equation (21) can be exploited further in a number of ways. Let us first look at the elementary MD formula, Equation (16) and focus on four shells that all have a common angular momentum L and contraction depth D. If that equation is directly applied, there will be O(L^8) integrals to be generated in the Cartesian basis, each of which involves summations of length O(L^2) leading to a total scaling of O(L^{10}). The subsequent transformation into the contracted basis can be carried out as a series of four partial transformations, Scaling as O($L^{10}D^4$). The final transformation into the spherical harmonics basis again involves four partial transformations, the most expensive of which scales as O(L^9).

In the factorized algorithm suggested by Equation (21), the size of the E- and R-matrices is of order E(O(L^2), O(L^2)) and R(O(L^2), O(L^2)). Hence, the two steps both only scale as O(L^6) and directly generate O(L^4) integrals in the spherical Harmonics basis. Furthermore, the second multiplication can be delayed until after the contraction step on either the bra- or the ket side by forming an intermediate:

$$\boldsymbol{T}(\mu\nu,t'u'v')=\sum_{i,j=1}^{D} d_i d_j \left(\boldsymbol{E}^{(ij)+}\boldsymbol{R}\right)_{\mu\nu,t'u'v'} \tag{25}$$

With cost O(L^6D^2) (here $\boldsymbol{E}^{(ij)}$ is the E-matrix for primitive combination i,j. Obviously, the contraction coefficients d_i,d_j can be part of the definition of the E-matrices and this is how it is done in practice). However, this step needs to be carried out for all O(D^2) primitive combinations in the ket and hence, the overall cost of this rate determining step is O(L^6D^4). The second step involves again O(L^6D^2) operations and generates the set of target integrals:

$$\boldsymbol{I}(\mu\nu,\kappa\tau)=\sum_{k,l=1}^{D} d_k d_l \left(\boldsymbol{T}\boldsymbol{E}^{(kl)}\right)_{\mu\nu,\kappa\tau} \tag{26}$$

Pseudocode for the proposed algorithm is shown in Scheme 1. Based on this analysis, the algorithm should perform particularly well for deeply contracted functions and higher angular momenta, where the efficiency of the BLAS level 3 operations is optimal. For low

$tuv=$	0,0,0	0,0,1	0,0,2	1,0,0	1,0,1	0,1,1	2,0,0	0,1,0	1,1,0	0,2,0
pz/pz	x	x	x							
pz/px	x	x		x	x					
pz/py	x	x				x		x		
px/pz	x	x		x	x					
px/px	x			x			x			
px/py	x			x				x	x	
py/pz	x	x				x		x		
py/px	x			x				x	x	
py/py	x							x		x

TABLE 2 Location of the nonzero products for two shells of p-functions in order to illustrate the process. Only matrix elements marked with ‘x’ are nonzero

angular momenta, organizing the data and performing the matrix multiplications will create a certain amount of overhead that will be dealt with accordingly as described below.

At first glance, the proposed algorithm may appear somewhat counterintuitive. The MD algorithm already has a FLOP count that is inferior to other algorithms such as the OS method.[20,21] The proposal to rewrite the *E*- and *R*-coefficients as matrices, taken face value, further worsens this FLOP count since not all combinations of t,u,v contribute to each basis function product (Tables 1 and 2). Hence, there is sparsity in the *E*-matrices and part of the computer time appears to be spent multiplying 0's. While this is undoubtedly the case, the overhead created is only significant for low angular momentum and workarounds will be described in the following text.

3 | IMPLEMENTATION

3.1 | Specialized treatment of low angular momenta

In implementing Equation (21), one faces the two challenges of overhead due to setting up the matrix multiplications and the suboptimal FLOP count that is particularly acute for low angular momenta. These problems are alleviated in the actual implementation using the following strategies:

1. Specialized loop-free matrix multiplication routines have been written for individual (low) angular momentum cases (up to $l_a+l_b=4$; here and below l_a+l_b refers to the sum of the two angular momenta on either the bra or ket side of the integral) that avoid multiplication by 0. This is readily achieved since in this algorithm, the positions of the 0-elements are precisely known
2. For the very lowest angular momenta $l_{\text{tot}} \leq 2$ highly optimized, loop-free routines have been developed that avoid the matrix multiplications all together.
3. Some other code optimizations that have been performed is to provide specialized routines for the R-matrices up to $l_{\max}=4\,(l_{\max}$ is the largest of the angular momenta involved). For higher angular momenta, the recursion relations are efficient, but for lower and intermediate angular momenta, additional speed is being gained by unrolling the short loops and optimizing the resulting algebraic expressions. This and most other loop unrolling that went into SHARK has been accomplished using the Maple V computer algebra system.

3.2 | Precomputation of shell pair data

An optimal implementation of the algorithm implied by Equation (21) of course also involves the precomputation of as many quantities as possible and arrangement of the data in a way that makes the algorithm perform optimally in the rate determining loops. To this end, SHARK uses a variable T_{cut} that is used to pre-screen primitive pairs. Obviously, this value must be set lower than the overall neglect threshold T_{int} that is used for screening Fock matrix contributions. The default value in ORCA has always been $T_{\text{cut}} \leq 0.01T_{\text{int}}$. Thus, for a given primitive pair of Gaussians, the matrix $\boldsymbol{E}(tuv,\mu\nu)$ is only created and stored if the Gaussian prefactor K_{AB} (multiplied by the product of contraction coefficients as well as $\sqrt{C}$) is larger than T_{cut}. These data are stored together with a few additional data items characterizing the shell pair that are helpful in forming shell quadruple data efficiently in the actual integral generation loop. This leads to a linear scaling number of retained matrices that usually can be held in memory. However, the program has provision to buffer these matrices on the hard drive. Thus, inside a given contracted pair of Gaussians, the algorithm automatically only processes non-negligible primitive shell pairs on both, the bra and the ket side.

3.3 | General contraction

In the case of generally contracted basis set, the factorization offered by Equation (21) is particularly advantageous but requires a slightly different algorithmic structure. Hence, in SHARK there always is a routine for segmented contraction and an analogous one for general contraction. The ORCA infrastructure detects generally contracted basis sets automatically and will switch the default algorithm. Alternatively, the user can set the desired contraction scheme and chose to carry out any calculation with any basis set in general contraction or segmented contraction.

For generally contracted basis sets, it is essential to not recalculate the integrals over primitive Gaussian shells over and over again, only to contract them with different contraction coefficients. Thus, the shell pair data in SHARK is organized fundamentally differently for generally contracted basis sets. Here, all shells on a given atom and given angular momentum are treated together (no matter how they were input by the user). Thus, the integral loops run over atom/angular momentum pairs. For example, the bra and ket sides of a four-index integral are characterized by $(A;l_A;B;l_B)$ and $(C;l_C;D;l_D)$ respectively with N_A,N_B,N_C,N_D basis functions and P_A,P_B,P_C,P_D primitives. In the setup procedure that determines the shell pair data, the algorithm then forms "giant" *E*-matrices:

$$E\Big(\{tuv\}_1\ldots\Big(\{tuv\}_{P_AP_B};\{\mu\nu\}_1\ldots\{\mu\nu\}_{N_AN_B}\Big) \tag{27}$$

Consequently, the R-matrix that is formed has the structure.

$$R\Big(\{tuv\}_1\ldots\Big(\{tuv\}_{P_AP_B};\{tuv\}_1\ldots\Big(\{tuv\}_{P_CP_D}\Big) \tag{28}$$

Two matrix multiplications then lead to the generation of all integrals contained in the combination $(A;l_A;B;l_B;C;l_C;D;l_D)$. The buffer space for these matrices is usually so large that the "giant" *E*-matrices are kept on disk and read as appropriate.

The benefit of the GC algorithm is that it generates a large number of integrals per unit time given the large size of the matrix multiplications. It also avoids all redundancies. However, the downside of the GC algorithm (and the general contraction scheme in general) is, that the efficiency of the prescreening process is greatly compromised since one can only screen at the levels of atoms and angular momenta. For direct SCF calculations that has the consequence that a calculation with, say, a correlation consistent basis set, always runs slower in general contraction than in segmented contraction by a factor of 1.5–3. However, for algorithms that process many densities and for which pre-screening is generally less effective (e.g., coupled-perturbed self-consistent field [CP-SCF] calculations during analytic Hessian evaluations), the efficiency of the algorithm is very high and outperforms the algorithm for segmented contraction. Furthermore, for bona fide generally contracted basis sets (e.g., ANOs), that are deeply contracted in all angular momenta, the general contraction scheme is crucial for achieving good performance.

3.4 | Partial general contraction

During the course of the development of SHARK, Dr. Helmich-Paris independently proposed an interesting way to handle PGC basis sets. This algorithm is published in reference 33 as part of the improved chain of spheres exchange (COSX) scheme and was copied into SHARK to offer full flexibility for all integrals.

In this algorithm, an intermediate basis is introduced in which all generally contracted shells are decontracted and left primitive. Depending on the underlying AO basis set, this intermediate basis may be moderately (20%–30% in correlation-consistent basis sets) to much larger (ANOs) than the original basis. The detection of basis sets that feature (partial) general contraction is automatized but can be overruled by the user.

The generation of integrals proceeds in the intermediate basis using the infrastructure for segmented basis sets. This makes sense, since all redundancies have been removed from the original basis set. Using the known contraction coefficients, quantities that the integrals are contracted with (densities, MO coefficients, singles amplitudes, and so on) are transformed into the intermediate basis. Whatever AO basis quantities are formed (e.g., Fock matrices, response matrices, partially transformed integrals) are formed in the intermediate basis and are being back transformed into the original basis after integral generation has finished (Scheme 2).

Since the decontraction/contraction steps are computationally inexpensive, this algorithm combines the best of the segmented and generally contracted basis set worlds as long as pre-screening is essential (e.g., in SCF and gradient calculations). The calculation of redundant integrals is completely avoided and at the same time, prescreening happens just as in segmented contraction. It is also important to note that no additional integral code needed to be written for the implementation of the PGC scheme since the computational machinery is identical to the segmented case.

```
DecontractDensity(P,NMat);
DecontractFock(F,NMat);

// Do the integral work

RecontractDensity(P,NMat);
RecontractFock(F,NMat);
```

SCHEME 2 Pseudocode for the partial contraction algorithm. Here NMat Fock matrices F are formed from NMat densities P.

At the same time, the algorithm creates overhead in the case that prescreening is not efficient or essential, but the increased size of the intermediate basis becomes critical (e.g., in integral transformations or the formation of many right-hand sides in analytic Hessian calculations). In such situations, the PGC algorithm will be inferior of the GC algorithm and also, within limits, to the segmented algorithm that calculates redundant integrals. Ongoing efforts will automatically determine the most efficient algorithm to be used for a given task.

3.5 | SHARK/Libint hybrid model

It will become apparent in the timing section, that the SHARK algorithm does not outperform Libint for all integral batches. In fact, for a number of intermediate angular momentum integral batches $(l_{\max} = 0 - 2)$ and lower contraction depths, the Libint routines perform faster than the corresponding SHARK ones. In order to take full advantage of this situation, a "hybrid" model has been implemented in which a predetermined "task" table decides for each angular momentum combination which algorithm is used. This is the default for SCF and gradient calculations but can be overridden by the user with the request to use only SHARK or only Libint routines. In developing SHARK, a new Libint interface has been created that is fully embedded in the SHARK infrastructure and philosophy. It also features code for GC basis sets. However, in this case, SHARK is much more efficient since it benefits from factorization while Libint does not and consequently has to go through four partial transformations that lead from the primitive to the contracted integrals.

3.6 | Integral digestion

A subject that deserves special attention is the digestion of the integrals in actual calculations since this can make or break the overall performance of an algorithm. One should not assume that this step is necessarily negligible compared to integral evaluation, especially, if one processes multiple target quantities and low angular momentum integrals. The latter have the lowest cost but are the by far most frequent. Hence, making sure that integrals over all angular momentum ranges are efficiently digested is highly important.

The perhaps most frequent use of integrals is the formation of Fock like matrices. There are two types of contributions that are of the Coulomb (J) and of the Exchange (X) type:

$$J^{(K)}_{\mu\nu} = f_J \sum_{\kappa\tau} (\mu\nu|\kappa\tau) P^{(K)}_{\kappa\tau} \tag{29}$$

$$X_{\mu\nu}^{(K)} = f_X \sum_{\kappa\tau} (\mu\kappa|\nu\tau) P_{\kappa\tau}^{(K)} \tag{30}$$

where $P_{\kappa\tau}^{(K)}$ is some effective density or related quantity and $(\mu\nu|\kappa\tau)$ represents a two-electron integral over contracted Gaussian basis functions μ,ν,κ,τ in (11|22) notation. The scaling factors f_J and f_X have been purposely left general. In the actual code they are passed by the calling routine and they differ depending on what type of Fock-like matrix is formed (RHF, UHF, response, CIS, …). In the interest of generality, it has been assumed that there might be a range of target $K = 1 \ldots N_J$ Coulomb matrices and $K = 1 \ldots N_X$ target exchange matrices. In evaluating Equations (29) and (30), it is important to recognize the integrals are being formed in batches representing the four shells involved. Second, there might be redundancies in the four indices and in the interest of efficiency, usually integral algorithms only generate one out of up to eight permutationally related set of integrals.

For a sufficient large system, the overwhelming majority of batches will feature non-redundant integral labels. Hence, it is important that the digestion algorithm does not run over all redundancy tests for all integrals batches but only checks for redundancies when they are known to be present.

The Coulomb digestion algorithm proceeds by defining sub-blocks of the Coulomb and density matrices $\mathbf{J}^{(IJ)}(D,IJ)$ and $\mathbf{P}^{(KL)}(D,KL)$ where $D = 1..N_J \, or \, N_X$ and shell pairs IJ and KL have the dimensions of $\dim(l_\mu)\dim(l_\nu)$ and $\dim(l_\kappa)\dim(l_\tau)$, respectively $(\dim(l) = 2l+1)$. The blocks of the density and the initialization of these sub-blocks happens as far outside in the loop structure as possible. In SHARK, the integral loops are organized such that the four angular momenta run outermost and the inner loops run over those surviving shell pairs that have the correct angular momentum combination.

Hence, for a given set of integrals $\mathbf{I}(IJ,KL)$ one forms:

$$\mathbf{J}^{(IJ)} += \mathbf{P}^{(KL)} \mathbf{I}^{+} \tag{31}$$

$$\mathbf{J}^{(KL)} += \mathbf{P}^{(IJ)} \mathbf{I} \tag{32}$$

The blocks are then added to the output Coulomb matrix and multiplied by f_J. In the case that $I = J$, only one triangle of $\mathbf{J}^{(IJ)}$ is added. Likewise, the redundancy in $\mathbf{P}^{(KL)}$ is taken care of by multiplying the density block by $1/2$ in the case that $K = L$. For low angular momentum cases $(l_{\text{tot}} \le 6)$, the matrix multiplications are not explicitly performed, but loop free, unrolled code is being employed. There also is specially optimized code for the case of a single Coulomb matrix being formed.

The case of the Exchange matrix is more complicated. Here, it is advantageous to first resort the integrals into two buffers $\mathbf{I}^{(1)}(IK,JL)$ and $\mathbf{I}^{(2)}(IL,JK)$. Then, the contributions can be formed as:

$$\mathbf{X}^{(IK)} += \mathbf{P}^{(JL)} \mathbf{I}^{(1)+} \tag{33}$$

$$\mathbf{X}^{(JL)} += \mathbf{P}^{(IK)} \mathbf{I}^{(1)} \tag{34}$$

$$\mathbf{X}^{(IL)} += \mathbf{P}^{(JK)} \mathbf{I}^{(2)+} \tag{35}$$

$$\mathbf{X}^{(JK)} += \mathbf{P}^{(IL} \mathbf{I}^{(2)} \tag{36}$$

Followed by summing the four blocks into the target matrix multiplied by f_X. In case of redundancies, it is the most efficient route to take of those into account in resorting the integral buffers since the redundancies are independent of the density-like matrices they are contracted with. Given functions μ,ν,κ,τ, the redundancy factor that each integral is multiplied with is:

$$(\mu\nu|\kappa\tau) \leftarrow (\mu\nu|\kappa\tau)\left[\left(\frac{1}{1+\delta_{\mu\nu}}\right)\left(\frac{1}{1+\delta_{\kappa\tau}}\right)\left(1-\frac{1}{2}\delta_{\mu\nu,\kappa\tau}\right)\right] \tag{37}$$

With the compound index $\mu\nu$ defined by $\mu\nu = \mu N + \nu$ if $\mu \ge \nu$ and $\mu\nu = \nu N + \mu$ otherwise ($N =$ number of basis functions). Addition of the sub-blocks to the target matrices then only requires a minimal amount of additional logic.

Other tasks like the assembly of gradients or direct contributions to the nuclear second derivatives profit from similar strategies.

It would be difficult to overemphasize the importance of the efficient integral digestion. The largest speedups in ORCA 5.0 over 4.2.1 amount to almost an order of magnitude for tasks involving many density-like matrices and these speedups can be mostly attributed to the vastly improved, highly optimized integral digestion in SHARK.[30]

3.7 | Taking advantage of factorization: Split-J and Split-RI-J

The factorization of the integrals allows for some interesting algorithmic developments in which the generation of the individual two-electron integrals is bypassed altogether. Such algorithms have of course been known for a long time[34–36] and will consequently only briefly been re-iterated here.

Consider the formation of a set of Coulomb-type matrices:

$$J_{\mu\nu}^{(K)} = f_J \sum_{\kappa\tau} (\mu\nu|\kappa\tau) P_{\kappa\tau}^{(K)} = f_J \sum_{\kappa\tau} \left(\mathbf{E}^{\mu\nu} \mathbf{R} \mathbf{E}^{\kappa\tau}\right)_{\mu\nu,\kappa\tau} P_{\kappa\tau}^{(K)} \tag{38}$$

It becomes evident, that the contraction of $\mathbf{E}^{\kappa\tau}$ with $P_{\kappa\tau}^{(K)}$ can be taken outside of the inner loops by defining the Hermite Gaussian density

$$\widetilde{P}(K,tuv) = \sum_{\kappa\tau} E(tuv,\kappa\tau) P_{\kappa\tau}^{(K)} \tag{39}$$

Followed by the computation of the computation of the Hermite basis Coulomb matrix

$$\widetilde{J}(K,tuv) = \sum_{t'u'v'} \widetilde{P}(K,tuv) R(tuv,t'u'v') = \left(\widetilde{\mathbf{P}}\mathbf{R}\right)_{K,t'u'v'} \tag{40}$$

The latter equation, of course, does not imply a large "global" matrix multiplication but the generation of the relevant sub-blocks of the

Coulomb matrices through BLAS-driven matrix multiplications. These matrix multiplications are implied by the summations below that will be written as summations in the interest of clarity. The algorithm is completed by the back transformation to the spherical harmonics Gaussian basis

$$J^{(K)}_{\mu\nu}=f_J\sum_{tuv}E(tuv,\mu\nu)\tilde{J}(K,tuv) \tag{41}$$

Thus, in the rate-determining step, the Coulomb matrices are formed directly in the Hermite Gaussian basis.

As shown in reference 36, an even more efficient algorithm can be formulated by combining the integral factorization with the resolution of the identity (RI) approximation.[37–39] Here, the point of departure is to insert the RI approximation (that introduces an auxiliary basis set) in the Equation (29) for the Coulomb matrices.

$$J^{(K)}_{\mu\nu}\approx f_J\sum_{\kappa\tau PQ}(\mu\nu|P)V^{-1}_{PQ}(Q|\kappa\tau)P^{(K)}_{\kappa\tau} \tag{42}$$

With the metric $V_{PQ}=(P|Q)$ consist of two-index repulsion integrals over the auxiliary basis functions P and Q. Each of the three-index repulsion integrals can be equally well-factorized using Equation (21):

$$(\mu\nu|P)=\left(\mathbf{E}^{(\mu\nu)}\mathbf{R}^{(\mu\nu,P)}\mathbf{E}^{(P)}\right)_{\mu\nu,P} \tag{43}$$

Thus, leading to

$$J^{(K)}_{\mu\nu}\approx f_J\sum_{\kappa\tau}\left(\mathbf{E}^{(\mu\nu)}\mathbf{R}^{(\mu\nu,P)}\mathbf{E}^{(P)}\right)_{\mu\nu,P}V^{-1}_{PQ}\left(\mathbf{E}^{(\kappa\tau)}\mathbf{R}^{(\kappa\tau,Q)}\mathbf{E}^{(Q)}\right)_{\kappa\tau,Q}P^{(K)}_{\kappa\tau} \tag{44}$$

This equation is best evaluated by the following steps:

1. Formation of the Hermite densities $\tilde{\mathbf{P}}$
2. Calculation of the "auxiliary Hermite pre-densities"

$$\tilde{P}(K,t'u'v')=\sum_{tuv}\tilde{P}(K,tuv)R(tuv,t'u'v') \tag{45}$$

where the labels $t'u'v'$ refer to the Hermite expansion of the auxiliary basis function

3. Completion of the "auxiliary spherical predensities"

$$\tilde{P}(K,Q)=\sum_{t'u'v'}\tilde{P}(K,t'u'v')E(t'u'v',Q) \tag{46}$$

4. Formation of the "auxiliary densities" through the solution of the linear equation systems (e.g., through Cholesky decomposition)

$$\tilde{\tilde{\mathbf{P}}}=\mathbf{V}^{-1}\tilde{\mathbf{P}} \tag{47}$$

5. Transformation of auxiliary density to the Hermite space

$$\tilde{\tilde{P}}(K,tuv)=\sum_{P}\tilde{\tilde{P}}(K,P)E(tuv,P) \tag{48}$$

6. Calculation of the Hermite RI–Coulomb matrix

$$\tilde{J}(K,tuv)=\sum_{t'u'v'}\tilde{\tilde{P}}(K,t'u'v')R(tuv,t'u'v') \tag{49}$$

7. Back transformation to the spherical Gaussian basis

$$J^{(K)}_{\mu\nu}\approx f_J\sum_{tuv}\tilde{J}(K,tuv)E(tuv,\mu\nu) \tag{50}$$

In this way, optimal use is made of the factorizations inherent in Equation (21) and the approximate factorization of the four-index integral offered by the RI-approximation. The rate limiting steps are step 2) and 6) for which the Hermite basis repulsion integrals are calculated. In re-implementing the Split-RI-J algorithm, it was found to be impossible to outperform the original 2003 implementation that, however, was only performant up to g-functions in the original and auxiliary basis set. The 2021 reimplementation of the algorithm is fully optimized up to $l=7$ in both the orbital and auxiliary basis sets and operates on an arbitrary number of densities. New highly optimized code has been produced using Maple V for the rate-determining steps (2) and (6).

Clearly, similar algorithms are always possible if the contraction involves both indices of the bra or ket side of the integral.

It is noted in passing that the RI–K integral approximation has also been reimplemented for energies and gradients. Here the integral factorization is, unfortunately, not of any help and the calculation has to proceed through the explicit evaluation of the three-index integrals. Hence, the implemented algorithm is identical to the one described in reference 40, but the implementation is far more compact due to the Loop/Kernel/Consumer infrastructure described below.

3.8 | Range separation

The concept of range separation has been introduced in order to divide integrals into a short-range and long-range part and this is has been used to considerable advantage in the development of range separated hybrid- and double hybrid functionals. It would lead too far to described all these developments here. It will only been briefly discussed, how range separation was included in SHARK in a very general way.

Range separation is characterized by three parameters: The range separation parameter ω, a scaling parameter for the non-range separated integral f and a scaling parameter for the range separated integral g. The range separation only affects the evaluation of the Boys function, Equation (19). Let the argument of the incomplete Gamma function (Equation [19]) be T (hence the integral is denoted as $(\mu\nu|\kappa\tau)(T)$), the range separated integrals can be written as:

$$(\mu\nu|\kappa\tau)_{RS}=f(\mu\nu|\kappa\tau)(T)+g(\mu\nu|\kappa\tau)(\kappa T) \tag{51}$$

where

$$\kappa=\frac{\omega^2}{\omega^2+\varrho} \tag{52}$$

$$\varrho=\frac{(a+b)(c+d)}{a+b+c+d} \tag{53}$$

And the F-function needs to be modified to:

$$F_m^{(RS)}(T)\leftarrow fF_m(T)+g\kappa^{m+\frac{1}{2}}F_m(\kappa T) \tag{54}$$

In this way, the range separation can be implemented at the lowest level of the integral calculation and without calculating multiple sets of integrals or elaborate logic in the code. SHARK allows the range separation for the Coulomb and exchange terms to be different. Thus, there is a total of six range separation parameters $\omega_J, f_J, g_J, \omega_X, f_X, g_X$ and range separation can be individually turned "on" or "off" for the Coulomb and the exchange contribution. This leads to only minimal changes in the program logic an ensures that Range separation is automatically available for all integral forming tasks performed by SHARK (Scheme 3). The general form is not necessary to evaluate present day range separated density functionals (that only manipulate the exchange term), but was chosen with future developments in mind.

3.9 | Derivatives

One particularly convenient aspect of the MD integral algorithm is the ease with which derivatives can be calculated. In fact, many operators can be represented as derivatives of a more elementary operator and or the derivatives can be relegated to the basis function products through integration by parts.

```
// No Range separation
if (!DoRangeSep){
  # Do the integral work
}
// With range separation
else{
  if (DoJ){
    // Do J without range separation
    // If both have the same range separation then
    // do X as well
    SetRangeSepJ();
    # do the integral work
    if (DoX && (!JequalX)){
      // Do X with range separation
      SetRangeSepX();
      # do the integral work
    }
  }
}
```

SCHEME 3 Pseudocode for the range separation algorithm in SHARK. SetRangeSepJ and SetRangeSepX set the global range separation paraemters to the respective values for J and X.

For the sake of the argument to be made, we consider only one dimension and write the charge distribution between two Cartesian Gaussians as:

$$\Omega_{ij}=\sum_{t=0}^{i+j}E_{ij}^{t}\Lambda_t \tag{55}$$

Here i,j represent the powers of the polynomial in front of the Cartesian Gaussian, E_{ij}^{t} are the Hermite expansion coefficients, and Λ_t is a Hermite Gaussian function of order t. Taking the derivative of the charge distribution w.r.t. center A yields:

$$\frac{\partial\Omega_{ij}}{\partial A}=\sum_{t=0}^{i+j+1}\frac{\partial E_{ij}^{t}}{\partial A}\Lambda_t \tag{56}$$

This shows that the calculation of the derivative requires a higher effort since it increases the Hermite expansion by the order of the derivative taken. However, Helgaker and Taylor invented an elegant re-formulation that circumvents the incrementation.[41] By noticing that the expansion coefficients E_{ij}^{t} only depend on the difference vector $\mathbf{D}=\boldsymbol{R}_A-\boldsymbol{R}_B$ and the Hermite integrals $R_{t+t'}$ only on the intermediate point $\boldsymbol{P}=\frac{a}{a+b}\boldsymbol{R}_A+\frac{b}{a+b}\boldsymbol{R}_B$, they showed that:

$$\frac{\partial\Omega_{ij}}{\partial D}=\sum_{t=0}^{i+j}E_{ij}^{t;1}\Lambda_t \tag{57}$$

$$\frac{\partial\Omega_{ij}}{\partial P}=\sum_{t=0}^{i+j}E_{ij}^{t}\Lambda_{t+1} \tag{58}$$

With no incremented sums. The new expansion coefficients satisfy a very similar recursion relation to the original ones.[41] Finally, in order to pass from variables $\boldsymbol{D},\boldsymbol{P}$ back to $\boldsymbol{A},\boldsymbol{B}$, one can make use of the relationships:

$$\frac{\partial}{\partial\boldsymbol{A}}=\frac{a}{a+b}\frac{\partial}{\partial\boldsymbol{P}}+\frac{\partial}{\partial\boldsymbol{D}} \tag{59}$$

$$\frac{\partial}{\partial\boldsymbol{B}}=\frac{b}{a+b}\frac{\partial}{\partial\boldsymbol{P}}-\frac{\partial}{\partial\boldsymbol{D}} \tag{60}$$

Using these relationships, one can define an efficient and compact integral derivative algorithm using the SHARK strategy (Scheme 4).

3.10 | Kernel/Consumer programming model

A key architectural feature of SHARK that has nothing to do with the algorithm itself but is of major consequence program maintenance and how programmers interact with the package is the Loop/Kernel/Consumer (LKC) infrastructure that we have designed. Based on an idea of Dr. Wennmohs, humungous amount of largely duplicated code can be avoided by using this model.

```
$\boldsymbol{I}^{X,Y,Z;1-3} = \mathbf{0}$
Loop Bra primitives
    $\boldsymbol{T}^{0-2} = \mathbf{0}$
    Loop Ket primitives
        $\boldsymbol{T}^{0} = \boldsymbol{R}\boldsymbol{E}_{ket}$
        $\boldsymbol{T}^{X,Y,Z;1} = \boldsymbol{R}^{X,Y,Z}\boldsymbol{E}_{ket}$
        $\boldsymbol{T}^{X,Y,Z;2} = -\frac{c}{q}\boldsymbol{R}^{X,Y,Z}\boldsymbol{E}_{ket} + \boldsymbol{R}\boldsymbol{E}_{ket}^{X,Y,Z}$
    End Ket
    $\boldsymbol{I}^{X,Y,Z;0} += \boldsymbol{E}_{bra}^{X,Y,Z}\boldsymbol{T}^{0}$
    $\boldsymbol{I}^{X,Y,Z;1} += \frac{a}{p}\boldsymbol{E}_{bra}\boldsymbol{T}^{X,Y,Z;1}$
    $\boldsymbol{I}^{X,Y,Z;2} += \frac{b}{p}\boldsymbol{E}_{bra}\boldsymbol{T}^{X,Y,Z;1}$
    $\boldsymbol{I}^{X,Y,Z;3} += \boldsymbol{E}_{bra}\boldsymbol{T}^{X,Y,Z;2}$
    End Bra
    $\boldsymbol{IA}^{X,Y,Z} = \boldsymbol{I}^{X,Y,Z;1} + \boldsymbol{I}^{X,Y,Z;0}$
    $\boldsymbol{IB}^{X,Y,Z} = \boldsymbol{I}^{X,Y,Z;2} - \boldsymbol{I}^{X,Y,Z;0}$
    $\boldsymbol{IC}^{X,Y,Z} = \boldsymbol{I}^{X,Y,Z;3}$
    $\boldsymbol{ID}^{X,Y,Z} = -\boldsymbol{IA}^{X,Y,Z} - \boldsymbol{IB}^{X,Y,Z} - \boldsymbol{IC}^{X,Y,Z}$
```

SCHEME 4 Pseudocode for the evaluation of two-electron integral derivatives. Here a,b,c,d are Gaussian exponents $p = a + b; q = c + d$. The Hermite integrals $\boldsymbol{R}$ are the undifferentiated integrals, $\boldsymbol{R}^{X,Y,Z}$ are derivative integrals (e.g. $R^{X}_{t,u,v} = R_{t+1,u,v}$). Likewise $\boldsymbol{E}_{\text{bra;ket}}$ are the regular E-coefficients and $\boldsymbol{E}^{X,Y,Z}_{\text{bra;ket}}$ represent the derivative coefficients $E^{t;1}_{ij}$. $\boldsymbol{IA},\boldsymbol{B},\boldsymbol{C},\boldsymbol{D}$ are the final derivative integral batches and translational symmetry has been used.

In order to introduce the LKC infrastructure, consider what a typical task in a quantum chemical program package consists of: one initializes output quantities, then starts a nested loop over basis set shells. In the innermost loop, one calculates a batch of integrals over a given kernel and these integrals are then immediately used ("consumed"). These tasks are highly repetitive and quite frequently, hundreds if not thousands of lines of setup and integral generation code are duplicated only to use the generated integrals in a slightly different way or to calculate integrals over a different kernel. This creates a large legacy-code problem down the road if one wants to change anything in the integral generation chain, since all changes must be implemented in multiple places.

In our LKC model, there (ideally) is only a single place in the entire code, where a loop over basis set shells pair combinations is performed. This "`IntegralLoop`" function receives basically two parameters: 1) an object that is derived from an abstract "`IntegralKernel`" object that takes shell pair/triple/quadruple information and generates integrals over a given integral kernel and 2) an object that is derived from an abstract "`IntegralConsumer`" class. The purpose of the latter object is to digest the generated integrals in whatever way is necessary for the task at hand.

The proposed infrastructure can be realized by using virtual functions that exist in all modern programming languages. A virtual function is an abstract function with a given, well-defined parameter list. In order to turn the abstract function into a concrete function, the process of inheritance need to be invoked, in which another class is derived from the abstract parent class that replaces the abstract function by an actual function that performs the desired task.

```
Function OneElectronIntegralLoop(KERNEL,CONSUMER)
  If (NOT GeneralContraction){
    Loop ish over shells
      Loop jsh<=ish over shells
       ICART=0
       For iprim=0..nprim-1
         Compute or get shell pair data
         KERNEL->PrimitiveIntegrals(IPRIM)
         Sum ICART += da*db*IPRIM
       end
       Transform ICART to Spherical Harmonics ILM
       CONSUMER->UseIntegrals(ILM)
    end ish,jsh
  }
  else{
    Loop ish over shells
      Loop jsh<=ish over shells
       For iprim=0..nprim-1
         Compute or get shell pair data
         KERNEL->PrimitiveIntegrals(IPRIM)
         Store IPRIM in IBUFF
       end
       Loop Basis Functions ij
         Transform to contracted functions
         Store in JBUFF
       end
       Loop over integral sets ibas,jbas
         Transform to spherical harm. ILM from JBUFF
         CONSUMER->UseIntegrals(ILM)
       end
    end ish,jsh
  };
}; // end of IntegralLoop
```

SCHEME 5 Outline of the kernel/consumer infrastructure for one-electron integrals in SHARK. The KERNEL is an object that computes a batch of integrals over a primitive pair of Gaussian shells. The CONSUMER is an object that takes a batch of integrals over contracted shells and does something with it that is defined in its definition of the virtual function `UseIntegrals`.

Both, the kernel and consumer virtual functions are called from within the `IntegralLoop`. Obviously, SHARK features a substantial library of available integral kernels that are listed below. Consequently, the only task to be performed by the programmer using SHARK, is to derive a simple class that consumes the integrals in whatever way is necessary to accomplish the task. In many cases even existing consumers can be used.

3.10.1 | One-electron integrals

In the case, of one-electron integrals, the LKC concept outlined above has been followed completely and without exception. Here the `IntegralLoop` is set up in two alternative ways: one branch for segmented basis sets and one branch for generally contracted basis sets (Scheme 5).

The kernel function is defined such that it calculates a batch of integrals over a primitive pair of Gaussian functions using the

TABLE 3 List of available one-electron kernels

Integral	Description
$\langle\mu\vert\nu\rangle$	Overlap and its first and second derivative, GIAOs, rel.
$\langle\mu\vert\nabla^2\vert\nu\rangle$	Kinetic Energy its first and second derivative, GIAOs, rel.
$\langle\mu\vert\boldsymbol{\nabla}\vert\nu\rangle$	Linear momentum
$\langle\mu\vert\boldsymbol{\nabla}\times\mathbf{r}\vert\nu\rangle$	Angular momentum, GIAOs
$-\sum_A Z_A\langle\mu\vert \mathrm{r}_A^{-1}\vert\nu\rangle$	Nuclear attraction (Nuclear charge Z_A at position $\boldsymbol{R}_A$) its first and second derivative, GIAOs
$\sum_i q_i\langle\mu\vert \mathrm{r}_i^{-1}\vert\nu\rangle$	Point charge fields (charges q_i at positions $\boldsymbol{r}_i$) its first and second derivative
$\langle\mu\vert\mathbf{r}\vert\nu\rangle$	Dipole moment its first derivative
$\langle\mu\vert\mathbf{r}^T\mathbf{r}\vert\nu\rangle$	Quadrupole moment
$\langle\mu\vert\left(\frac{\partial}{\partial\mathbf{r}}\frac{1}{\mathbf{r}}\right)\vert\nu\rangle$	Electric field
$\langle\mu\vert\left(\frac{\partial^2}{\partial\mathbf{r}^2}\frac{1}{r}\right)\vert\nu\rangle$	Electric field gradient
$\frac{\alpha^2}{2}\langle\mu\vert\frac{\boldsymbol{l}^{(A)}}{r_A^3}\vert\nu\rangle$	Nucleus/Spin-orbit coupling, GIAO
$\langle\mu\vert\mathrm{dia,para}\vert\nu\rangle$	Dia- and paramagnetic integrals over GIAOs for NMR calculations. Gauge integrals with common integrals for EPR calculations.[a]
$\langle\mu\vert \mathrm{e}^{i\boldsymbol{kr}}\vert\nu\rangle$	Exact oscillator strength

Notes: "GIAO" indicates that the gauge including atomic orbital versions of these integrals are available. "rel" indicates that the scalar relativistic versions of these integrals (here the kernel $\widehat{\boldsymbol{o}}$ is replaced by $\widehat{\boldsymbol{p}\boldsymbol{o}\boldsymbol{p}}$, where $\widehat{\boldsymbol{p}}$ the linear momentum operator). All integrals involving a Coulomb operator are also available for finite nuclei.

[a]See, for example, references 42, 43 for detailed forms of the operators involved.

straightforward, original MD algorithm. Equations for individual kernels can be found, for example, in the authoritative review of Helgaker and Taylor.[4] The presently known one-electron kernels in SHARK are summarized in Table 3.

The only difference between the segmented and GC branches of the one-electron `IntegralLoop` is how the contraction is applied. In the case of segmented contraction, the target integrals are directly summed into the final integral target array using the contraction coefficient products. After completion of the loop of primitive pairs, the consumer function is triggered. In the case of GC bases, the primitive integrals are summed into multiple target arrays reflecting all possible basis function combinations in the general contraction. After completion of this task, the consumer function is triggered for each of these individual target integral arrays. In this way, the generation of one-electron integrals is equally efficient for segmented and GC basis sets (Scheme 5). It should be noted that this is the one and only place in SHARK, where integrals are explicitly transformed into the spherical Harmonics basis. No other code part or calling function ever needs to concern itself with such small technicalities.

In the majority of cases, the result of a one-electron integral generation process is simply a matrix holding the respective integrals. The structure of the resulting matrix depends on the symmetry of the kernel. SHARK distinguishes between symmetric, anti-symmetric, and unsymmetric kernels. A specialized function recognizes the symmetry of the Kernel, calls the `IntegralLoop` function and assembles the output matrix of the appropriate symmetry. The SHARK interface simply defines a function `OneElectronIntegrals(KERNEL,H)`. That is called from outside SHARK. It simply passes a Kernel object and what is returned is a single or a series of matrices of the appropriate kernel length and symmetry.

TABLE 4 List of available two-electron kernels

Integral	Description
$(\mu\nu\vert \mathrm{r}_{12}^{-1}\vert\kappa\tau)$	Electron–electron respulsion and its first and second (only libint) derivative, GIAOs, range separation
$\left(\mu\nu\vert\left(\frac{\partial^2}{\partial\mathbf{r}_{12}^2}\frac{1}{r_{12}}\right)\vert\kappa\tau\right)$	Electron–electron spin–spin interaction
$(\mu\nu\vert\mathbf{SOC}\vert\kappa\tau)$	Spin-own-orbit and spin-other orbit interaction, GIAO[a]
$(\mu\nu\vert \mathrm{f}_{12}\vert\kappa\tau)$	Five different Kernels for f_{12} calculations (Libint only) (f_{12}, f_{12}^2,$\mathrm{f}_{12}r_{12}^{-1}$, $\left[\mathrm{f}_{12},\left[\widehat{T}_1,\mathrm{f}_{12}\right]\right]$). f_{12} is the geminal operator and $\widehat{T}_1$ is the kinetic energy operator for electron 1.[b]

Notes: "GIAO" indicates that the gauge including atomic orbital versions of these integrals are available. All integrals are also available as three index versions for the RI approximation.

[a]See for example reference 44 for the detailed forms of the operators.

[b]See for example reference 45 for detailed forms of the operators.

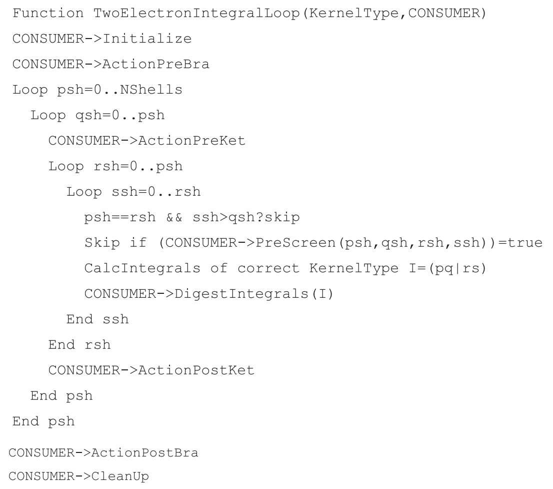
A less-frequent case is met, when the one-electron integrals are to be used on the fly to assemble a target quantity. This is the case, for example, in the calculation of gradients or the calculation of nuclear magnetic resonance shielding tensors. In this case, SHARK defines the appropriate consumers and triggers the IntegralLoop for the generation of integrals in a wrapper that is called from the outside ORCA environment.

3.10.2 | Two-electron integrals

For two-electron integrals, the idealized LKC model followed for one-electron integrals is definitely feasible. However, one would lose too much of the efficiency of the algorithm due to the astronomically large number of virtual function calls. Hence, in this case, the SHARK follows a compromise: a limited number of integral kernels are known to SHARK and are explicitly triggered inside the `IntegralLoop` while the consumer is still fully general and triggered by the virtual function calls described above.

The presently known two-electron kernels in SHARK are summarized in Table 4.

The `IntegralLoop` in the two-electron case has a slightly more elaborate Consumer infrastructure to allow for convenient "insert points" that are necessary to perform the required range of quantum chemical tasks (Scheme 6). First of all, each Consumer *must* define a virtual pre-screening function that tells the integral loop to skip negligible batches. Second, each Consumer *must* define a virtual function



```
Function TwoElectronIntegralLoop(KernelType,CONSUMER)
CONSUMER->Initialize
CONSUMER->ActionPreBra
Loop psh=0..NShells
  Loop qsh=0..psh
    CONSUMER->ActionPreKet
    Loop rsh=0..psh
      Loop ssh=0..rsh
        psh==rsh && ssh>qsh?skip
        Skip if (CONSUMER->PreScreen(psh,qsh,rsh,ssh))=true
        CalcIntegrals of correct KernelType I=(pq|rs)
        CONSUMER->DigestIntegrals(I)
      End ssh
    End rsh
    CONSUMER->ActionPostKet
  End psh
End psh
CONSUMER->ActionPostBra
CONSUMER->CleanUp
```

SCHEME 6 Pseudocode illustrating the Loop/Kernel/Consumer infrastructure for two-electron operators

for integral digestion. All other virtual functions indicated in the pseudocode in Scheme 6 are optional.

To illustrate the concept consider the formation of a Fock matrix or a series of Fock matrices. Here, nothing is necessary, except the definition of the two mandatory virtual functions. The first function performs the pre-screening and the second function implements or calls code to execute Equations (31)–(36).

By contrast, the case of an integral transformation is more elaborate. Here, a good algorithm would typically assign memory to $\dim(l_p)\dim(l_q)$ temporary matrices $\boldsymbol{T}$ of the AO dimension N in the function `ActionPreKet` and initialize them. The `DigestIntegrals` function would simply store the integral $\boldsymbol{T}[p,q](r,s)=(pq|rs)$ (one matrix for each function pair p,q in the shell pair psh,qsh). The function `ActionPostKet` would then transform these matrices to the MO basis using BLAS level 3 operations and store them on disk. After finishing the integral loop, all half-transformed integrals would reside on disk and would be ready for further processing. Once such a consumer has been written it can also be applied to different Kernel types without writing a single line of additional code.

As mentioned above, ideally there would be only a single `IntegralLoop` but the actual code has several. The various loops trigger four-index SHARK integrals, four-index Libint integrals, three-index SHARK integrals or three-index Libint integrals. The latter two functions each feature two variants in which the index of the auxiliary basis function runs in the outermost or innermost loop respectively. Both are necessary for efficient completion of certain tasks like the completion of a Coulomb matrix, the generation of an auxiliary basis density or the transformation of the three-index integrals to the molecular orbital basis. Each of these functions can be called with an arbitrary auxiliary basis set as input.

3.11 | Task drivers

ORCA, like many other quantum chemical packages, features a range of different approximate methods. For example, the Coulomb term can be approximated using the Split-RI-J approximation[36] and the exchange with the chain of spheres (COSX)[33,46–48] or RI-K method.[40] There may or may not be any exact exchange present in a given density functional, there might be range separation, solvent terms or external potentials to name only a few options. Often, a considerable amount of program logic is necessary to maneuver through the forest of approximations together with all consistency checks that determine whether all conditions for a given approximation to be eligible for a given task have been met. In order to drastically simplify the workflow, SHARK defines a number of "task drivers" that automatically take care of the active approximations such that the host application can drastically simplify the logic flow. Typical tasks drivers involve the formation of Fock matrices, response matrices, gradients, or residual vectors.

4 | TIMING COMPARISONS

Timing comparisons between ORCA 4.2.1 and ORCA 5.0 for somewhat real-life quantum chemical calculations have been published recently in the article describing the features and functionality of ORCA 5.0.[30] Hence, in this section, there will only be a brief comparison between the SHARK and Libint algorithms. In doing so, it should be realized that it is very difficult to reach a fair comparison that does full justice to both algorithms. For example, in practice the SHARK algorithm profits from pre-computing the E-matrices while there is much more limited precomputation possible for Libint. Second, SHARK is extremely tightly integrated with ORCA, which is possible, because they are both developed simultaneously. It is difficult, if not impossible, to achieve a similarly tight integration for any outside software package in the host application. Hence, there are ORCA specific wrappers that call the genuine Libint library routines. As will become evident below, any time lost in these wrappers is a feature of ORCA, not of Libint. It should also be stated clearly and unambiguously, that the comparison is in no shape or form designed or intended to make Libint look inferior. Libint is a wonderful and highly efficient integral package. It has been and will continue to be an important part of ORCA. It is merely supplemented by and integrated into the SHARK infrastructure.

For the timing test, batches of integrals were repeatedly calculated over a contracted Gaussian function with contraction depth D and angular momentum l. That same basis function was arbitrarily attached at four different positions ((0,0,0), (0,2,0),(0,0,2), and (2,0,0) atomic units) for the evaluation of the shell quadruple integrals. The Gaussian used was a normalized STO-DG expansion with a Slater exponent of 1.75 and main quantum number $n=l+1$. The calculations were carried out on single core of an 8-core Macintosh Notebook running a 2.4 GHz intel i9 processor as well a newer Macbook with the Apple M1 max processor. The number of repetitions of each batch was set to $\#(\text{repeats})=\max\left(5,\frac{5\times10^7}{\left(\max\left(1,l^8\right)D^4\right)}+1\right)$ which was found to be appropriate to guarantee calculation times that could be measured with meaningful accuracy (roughly 0.5–20 s). No screening was performed. It should be noted that the SHARK calculations had

TABLE 5 Timing comparison of integral batches with all functions having angular momentum *l* and contraction depth *D* on a single core of 8 core Macbook Pro running an Intel i9 processor

*l**D*	1	2	3	4	5	6
0	1.134	1.728	2.043	2.363	2.453	2.609
1	1.121	1.054	1.109	1.161	1.161	1.194
2	2.165	1.208	1.185	1.217	1.249	1.239
3	4.225	2.706	2.775	2.741	2.791	2.772
4	5.606	3.371	3.296	3.314	3.410	3.381
5	6.580	2.643	2.460	2.414	2.467	2.457
6	17.797	3.292	2.560	2.447	2.434	2.410
7	35.590	4.636	2.773	2.440	2.373	2.329

Notes: The numbers in the table give tm(Libint)/tm(SHARK), where tm(X) is the time taken by algorithm X to calculate the indicated number of integral batches.

TABLE 6 Timing comparison of integral batches with all functions having angular momentum *l* and contraction depth *D* on a single core of 8 core Macbook Pro running an Apple M1 max processor

*l**D*	1	2	3	4	5	6
0	1.371	1.724	2.073	2.268	2.347	2.430
1	1.100	1.003	1.052	1.169	1.157	1.174
2	1.199	0.765	0.778	0.793	0.802	0.791
3	5.471	3.628	3.610	3.355	3.787	3.406
4	8.758	5.284	5.210	5.229	5.264	5.227
5	9.792	5.589	4.409	4.360	4.655	4.788
6	18.567	5.912	4.852	4.704	4.751	4.604
7	31.497	6.749	5.360	5.227	5.412	5.031

Notes: The numbers in the table give tm(Libint)/tm(SHARK), where tm(X) is the time taken by algorithm X to calculate the indicated number of integral batches.

to be done with prototype code since the integration of the SHARK integral routines into the interface is so deep, that an isolated calculation of this kind would not have been possible.

Since absolute timings are essentially meaningless, Tables 5 and 6 give the ratio of the time taken by Libint versus the time taken by SHARK as a function of angular momentum and contraction depth for the two different processor-type investigated. Looking at Tables 5 and 6, it becomes evident that under the chosen conditions, the SHARK algorithm is faster than the Libint algorithm for all investigated batches. For the case of $l = 0$, this solely stems from the tighter integration into the interface that avoids a certain amount of overhead in the Libint wrapper. For $l = 1$, the performance is very similar, while for $l = 2$, SHARK wins on the Intel and Libint wins on the M1. It appears that here optimized code was not produced for SHARK and the matrices involved are not yet large enough to let the efficiency of BLAS take over. For higher angular momenta, the advantage of the SHARK algorithm reaches a factor 2–3.5 on the intel i9 and as much as 4–6 on the Apple M1 max. It should be noted that for mixed lower angular momentum batches, Libint often performs better than SHARK, presumably due to being FLOP count optimal. The very high speedups reached for high angular momenta and a single primitive Gaussian are reproducible but appear anomalous and remain unexplained. The numbers obtained for 2 or more primitives in the contraction appear to be more trustworthy.

In order to investigate a more real-life example, timings were also measured for a single RHF Fock matrix build on the Penicillin molecule with the def2-TZVPP basis set using the identical initial guess density matrix. Here, both Libint and SHARK run through the exact same code, perform pre-screening identically (according to ORCA's TightSCF criteria) and also call identical integral digestion routines such that the differences in execution time should largely reflect the time taken to compute the integrals.

The results collected in Table 7 show some interesting trends. First of all, low angular momentum integrals take far more overall time then high angular momentum integrals but there is not one angular momentum combination that is strongly dominating the computation time. The reason for this behavior is that there are far more integrals over low angular momentum batches to be calculated and the lower angular momentum basis functions are more deeply contracted than the higher angular momentum functions, that mostly serve as polarizations with a contraction depth of only 1–2. The situation would only change significantly in all-electron calculations on systems with many 4f or 5f elements or heavier transition metals from the second and third transition row.

For the low angular momentum cases, the performance of SHARK and Libint is quite similar. The timing advantage for SHARK arises almost completely from the deep integration of the code into the host application. In particular for the (ss|ss) integral class, the timing difference is entirely caused by the ORCA specific wrapper of the Libint library function. More efficient wrappers would be conceivable. While for the higher angular momentum integrals SHARK is usually the more efficient algorithm, there are noticeable exceptions. In the medium angular momentum range, Libint often excels as is most clearly visible in the timings for the (pd|pd) integral class. In a variety of the mixed angular momentum cases of intermediate angular momentum, Libint also has the edge.

The column labeled "hybrid" invokes the hybrid algorithm that was described above. Judging by the overall time, it is successful, since it outperforms either pure SHARK or pure Libint. However, looking at individual batches, it appears that the algorithm does not always pick the most efficient variant. Moreover, comparing the three columns "SHARK," "Libint," and "Hybrid," the latter should be essentially identical to one of the two former. While this is mostly the case, there can be significant deviations of 20% and more. These timing fluctuations are unavoidable on modern hardware, despite the fact that care was taken that no other applications were running and no other load was on the CPU than the quantum chemical calculation. The difficulty to even measure reliable and accurate timings is obviously an obstacle for developing efficient algorithms. Consequently, it is plausible that the hybrid scheme does not always make perfect choices with respect to the "best" algorithm for a given batch, despite the fact that it is overall successful. In this specific example, the

TABLE 7 Timings for some integral classes in the formation of a single RHF Fock matrix for the Penicilin molecule with the def2-TZVPP basis (1007 basis functions) and the density created by the PModel guess

	Intel i9			Apple M1 max[a]		
	SHARK	Libint	Hybrid	SHARK	Libint	Hybrid
(ss/ss)	2.578	4.653	2.567	1.519	2.461	1.530
(ss/pp)	7.348	8.958	7.265	4.676	5.200	4.666
(ss/dd)	3.744	3.422	3.957	2.137	2.123	2.144
(ss/ff)	1.057	1.080	1.137	0.730	0.697	0.743
(pp/pp)	6.371	6.705	6.376	4.536	4.488	4.589
(pp/dd)	7.425	8.432	7.715	5.853	5.121	5.112
(pp/ff)	3.623	3.632	2.637	1.954	2.533	1.952
(dd/dd)	1.752	3.091	3.186	1.346	2.131	2.119
(dd/ff)	1.342	3.557	1.320	0.887	2.322	0.889
(ff/ff)	0.246	0.955	0.239	0.112	0.609	0.112
(sp/sp)	14.482	16.892	14.375	9.508	9.432	9.647
(sd/sd)	6.906	6.702	6.833	4.083	4.054	4.117
(sf/sf)	3.016	1.966	2.076	1.732	1.462	1.482
(pd/pd)	21.786	14.546	15.028	11.802	9.991	9.958
(pf/pf)	4.460	4.887	4.923	3.473	3.617	3.604
(df/df)	1.956	5.156	1.997	1.257	3.632	1.251
Total	417.052	480.891	375.654	328.781	315.378	297.282

Note: A single core of the indicated CPU was used in these calculations.
[a] Compiler flags used:

```
clang++ -std=c++11 -mcpu=apple-m1 -mtune=apple-m1 -mmacosx-version-min=12.1.
```

program spends almost equal amounts of time in the SHARK and Libint portions of the code.

The comparison of the intel i9 and Apple M1 max timings shows that the latter is more efficient by 40%–50%. In conjunction with the SHARK algorithm, it is also remarkably consistent and shows smaller timing fluctuations than the other combinations. For comparison, on the M1 the calculation times for the def2-SVP basis set were 16.5 s (SHARK), 17.3 s (Libint), and 16.0 s (Hybrid) while for the much larger def2-QZVPP basis set, the measured times were 3175.1 s (SHARK), 3862.4 s (Libint), and 2878.6 s (Hybrid). Hence, it appears that for the basis set that is used in most computational chemistry studies (double- and triple-zeta bases), the algorithms are comparable but that for larger basis sets, the SHARK concept has efficiency advantages. In all cases, the "Hybrid" concept to pick the most efficient strategy for a given integral batch pays off, the more the larger the basis set is.

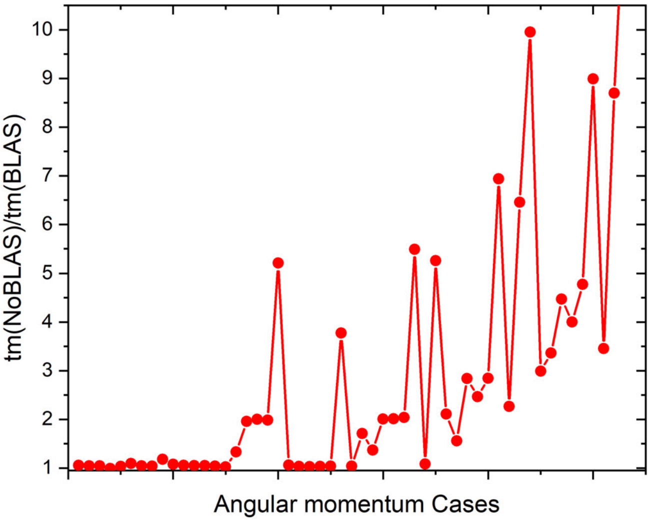


FIGURE 1 Comparison of timings for individual integral classes for the Penicillin example discussed above with the def2-TZVPP basis set for program version using the vendor BLAS library compared to a BLAS free program version. The angular momenta in each batch (la,lb|lc,ld) increase from the left to the right according to lb ≤ la, lc ≤ la, ld ≤ lc(lc! = la)/lb(la = lc).

A referee has asked the sensible question how much the use of BLAS actually accelerates the calculations. To this end, a program version was compiled in which the BLAS calls were directed toward hand written matrix multiplications. We can then compare the performance of the BLAS free versus the BLAS linked version of SHARK. The result is shown in Figure 1. It is evident that for the low angular momentum cases, BLAS does not help (in fact, it is not used there since this is code that proceeds via hand optimized routines. However, the speedup due to BLAS quickly reaches a factor of 2–6 for more elaborate cases and peaks at a factor of 20 for (ff|ff) integrals (off-scale in Figure 1). Overall, the Fock matrix formation example discussed above took 617.9 s without BLAS versus 328.8 s with BLAS. Thus, the average factor for the def2-TZVPP basis set is about a speedup of two. It will be much larger for larger basis sets and smaller for double zeta and other smaller bases. Thus, taken together, the SHARK algorithm definitely profits from using optimized BLAS libraries in a major way.

Quite obviously, one should take comparisons like the one presented in this section with a grain of salt since it depends on very many factors, including hardware, state of the hardware, system load, compilers, libraries and also the quantum chemical system that is being investigated. However, it is hopefully still fair to state that there is evidence that the performance of the SHARK integral algorithm is close to the state of the art.

5 | CONCLUSIONS

In this paper, the SHARK integral generation and digestion system has been described. While SHARK is partial to the ORCA package, it is hoped that some of the concepts described prove to be useful for other developers.

The SHARK integral algorithm is an efficient variant of the McMurchie-Davidson[17,18] integral algorithm. It leads to a factorization of integral batches such that only integrals over primitive Hermite Gaussian basis functions must be formed in the innermost loop and the actual integrals are assembled through two BLAS level 3 matrix operations. The focus on BLAS level 3 operations in both, integral generation and digestion, practically guarantees, that the algorithm remains performant on future hardware architectures. Matrix multiplications are always central to scientific computing and every vendor supplies highly optimized library routines to complete this task. Secondly, the factorization of the integrals is highly convenient since it allows the algorithm to move a significant amount of work outside the innermost loop which simplifies contraction steps or allows for efficient specialized algorithms (e.g., General Contraction, Split-J, Split-RI-J, and so on) to be implemented. Furthermore, the elegant nature of the MD algorithm makes it straightforward to derive and implement integral formulae for a wide range of one- and two-electron operators. Of course, closely related algorithms have been in use for a long time. For example Split-J is conceptual identical to the pioneering work of Almlöf and Ahmadi[34] and the "J-engine" algorithm by Head-Gordon and co-workers.[35] Likewise, algorithms very close to the original Split-RI-J[36] have been developed and extended by other research groups.[49,50] A similar idea to focus on BLAS level 3 operations is also at the heart of the COLD PRISM algorithm proposed by Gill and co-workers.[51]

It is noted in passing that the algorithm is rather easy to implement once the infrastructure to compute the $\boldsymbol{E}$ and $\boldsymbol{R}$ matrices is available. The calculation of the E-matrices is not time critical unless it is repeated in the inner loops. Hence, the direct application of the original recursion relations is completely sufficient in terms of computational efficiency. On the other hand, the formation of the R-matrices *is* time critical and some effort should be made to make this code perform well. For higher angular momenta (say beyond roughly $l_{\text{tot}} = 8$), the recursion relations are efficient. For lower angular momentum cases, optimized rolled out code is probably to be preferred. Given the availability of these two tools and an algorithm to enumerate and address the Hermite combinations *tuv*, the algorithm can be programmed with little effort. In fact, the actual code is not much longer than the pseudo-code fragments shown in this article.

In addition to the compact and efficient algorithm, the virtual function-based Loop/Kernel/Consumer (LKC) concept implemented in the SHARK interface allows us to replace many thousands of lines of legacy code with just a few lines of highly transparent code that is also highly reliable since the underlying infrastructure has been carefully optimized and debugged. In essence, the LKC concept leads to very much cleaner, leaner code that is much easier to maintain and extend compared to what this author has seen (or created) in quantum chemical program packages over the years. In addition, one should not underestimate the power of strongly separating the technical tasks like integral generation from the algorithmically "productive" tasks that implement the essential scientific ideas behind a given methodological project. In conclusion, the development of SHARK provides a compact, solid, transparent and efficient programming platform that will greatly accelerate future program development inside the ORCA programming environment.

ACKNOWLEDGMENTS

It is the author's pleasure the dedicate this paper to the celebration of Prof. Gernot Frenking's 75th birthday. The author gratefully acknowledges the support that he has received from him over the years, in particular in the early phases of my scientific career. Gernot's enthusiasm for applying quantum mechanics to chemistry has always been infectious. The author wishes him many more years of exciting science! SHARK has been a lovechild of the pandemic lockdown situation in 2020 and 2021. The author had first presented the basic ideas documented here in a group meeting in 2016 but it took the time and contemplation accompanying the lockdown situation to finally come to life. The author is most grateful to Frank Wennmohs, Ute Becker, Georgi Stoychev, Kantharuban Sivalingam and Benjamin Helmich-Paris for many discussions and help with coding the SHARK project. Last but not least, the author gratefully acknowledges the generous financial support of the Max Planck Society that is instrumental for our continued development efforts. Open Access funding enabled and organized by Projekt DEAL.

DATA AVAILABILITY STATEMENT

Data available on request from the authors.

ORCID

Frank Neese https://orcid.org/0000-0003-4691-0547

REFERENCES

[1] J. C. Slater, *The Quantum Theory of Atomic Structure*, Vol. II, McGraw-Hill, New York **1960**.

[2] J. Fernandez-Rico, R. Lopez, A. Aguado, I. Ema, G. Ramirez, *J. Comput. Chem.* **1998**, *19*(11), 1284.

[3] E. O. Steinborn, H. H. H. Homeier, E. J. Weniger, *J. Mol. Struct. THEOCHEM* **1992**, *92*, 207.

[4] T. Helgaker, P. R. Taylor, in *Modern Electronic Structure Theory* (Ed: D. R. Yarkony), World Scientific, Singapore **1995**, p. 725.

[5] F. Weigend, *Phys. Chem. Chem. Phys.* **2006**, *8*, 1057.

[6] F. Weigend, *J. Comput. Chem.* **2008**, *29*(2), 167.

[7] F. Weigend, R. Ahlrichs, *Phys. Chem. Chem. Phys.* **2005**, *7*(18), 3297.

[8] F. Weigend, F. Furche, R. Ahlrichs, *J. Chem. Phys.* **2003**, *119*, 12753.

[9] J. Almlöf, P. R. Taylor, *Adv. Quantum Chem.* **1991**, *22*, 301.

[10] T. H. Dunning, *J. Chem. Phys.* **1994**, *100*, 5829.

[11] D. E. Woon, T. H. Dunning, *J. Chem. Phys.* **1993**, *98*(2), 1358.

[12] D. E. Woon, T. H. Dunning, *J. Chem. Phys.* **1995**, *103*(11), 4572.

[13] J. A. Pople, W. J. Hehre, *J. Comput. Phys.* **1978**, *27*(2), 161.

[14] M. Head-Gordon, J. A. Pople, *J. Chem. Phys.* **1988**, *89*(9), 5777.

[15] P. M. W. Gill, M. Head-Gordon, J. A. Pople, *Int. J. Quantum Chem.* **1989**, *S23*, 269.

[16] P. M. W. Gill, J. A. Pople, *Int. J. Quantum Chem.* **1991**, *40*, 753.

[17] L. E. McMurchie, E. R. Davidson, *J. Chem. Phys.* **1977**, *66*(7), 2959.

[18] L. E. McMurchie, E. R. Davidson, *J. Comput. Phys.* **1978**, *26*(2), 218.

[19] M. Dupuis, J. Rys, H. F. King, *J. Chem. Phys.* **1976**, *65*, 111.

[20] S. Obara, A. Saika, *J. Chem. Phys.* **1986**, *84*, 3963.

[21] S. Obara, A. Saika, *J. Chem. Phys.* **1988**, *84*, 1540.

[22] K. Ishida, *J. Chem. Phys.* **1991**, *95*, 5198.

[23] P. M. W. Gill, *Adv. Quantum Chem.* **1994**, *25*, 141.

[24] R. Lindh, in *Encyclopedia Comp Chem* (Ed: P. von Rague-Schleyer), John Wiley & Sons, Chichester **2002**.

[25] Valeev, E.. **2021**. http://libint.valeyev.net/

[26] Q. Sun, *J. Comput. Chem.* **2015**, *36*, 1664.

[27] F. Neese, *WIRES Comput. Mol. Sci.* **2012**, *2*(1), 73.

[28] F. Neese, *WIRES Comput. Mol. Sci.* **2018**, *8*(1), 6.

[29] F. Neese, F. Wennmohs, U. Becker, C. Riplinger, *J. Chem. Phys.* **2020**, *152*(22), 224108.

[30] F. Neese, *WIRES Comput. Mol. Sci.* **2022**, e1606.

[31] H. B. Schlegl, M. J. Frisch, *Int. J. Quantum Chem.* **1994**, *54*, 83.

[32] C. Cohen Tanudji, B. Diu, F. Laloe, *Quantum Mechanics*, Vol. 2, Wiley, Paris **1977**.

[33] B. Helmich-Paris, B. de Souza, F. Neese, R. Izsák, *J. Chem. Phys.* **2021**, *155*(10), 104109.

[34] G. R. Ahmadi, J. Almlöf, *Chem. Phys. Lett.* **1995**, *246*(4–5), 364.

[35] C. A. White, M. Head-Gordon, *J. Chem. Phys.* **1996**, *104*(7), 2620.

[36] F. Neese, *J. Comput. Chem.* **2003**, *24*(14), 1740.

[37] O. Vahtras, J. Almlöf, M. W. Feyereisen, *Chem. Phys. Lett.* **1993**, *213*(5–6), 514.

[38] E. J. Baerends, D. E. Ellis, P. Ros, *Chem. Phys.* **1973**, *2*(1), 41.

[39] B. I. Dunlap, J. W. D. Connolly, J. R. Sabin, *J. Chem. Phys.* **1979**, *71*(8), 3396.

[40] S. Kossmann, F. Neese, *Chem. Phys. Lett.* **2009**, *481*(4–6), 240.

[41] T. Helgaker, P. R. Taylor, *Theor. Chim. Acta* **1992**, *83*, 177.

[42] G. L. Stoychev, A. A. Auer, R. Izsak, F. Neese, *J. Chem. Theory Comput.* **2018**, *14*(2), 619.

[43] F. Neese, *J. Chem. Phys.* **2001**, *115*(24), 11080.

[44] F. Neese, *J. Chem. Phys.* **2005**, *122*, 3.

[45] W. Klopper, F. R. Manby, S. Ten-No, E. F. Valeev, *Int. Rev. Phys. Chem.* **2006**, *25*(3), 427.

[46] F. Neese, F. Wennmohs, A. Hansen, U. Becker, *Chem. Phys.* **2009**, *356*(1–3), 98.

[47] R. Izsak, F. Neese, *J. Chem. Phys.* **2011**, *135*, 14.

[48] R. Izsak, F. Neese, W. Klopper, *J. Chem. Phys.* **2013**, *139*, 9.

[49] A. Sodt, J. E. Subotnik, M. Head-Gordon, *J. Chem. Phys.* **2006**, *125*(9), 194109.

[50] S. Reine, A. Krapp, M. F. Iozzi, V. Bakken, T. Helgaker, F. Pawloswski, P. Salek, *J. Chem. Phys.* **2010**, *1334*(4), 044102.

[51] T. R. Adams, R. D. Adamson, P. M. W. Gill, *J. Chem. Phys.* **1997**, *107*(1), 124.

How to cite this article: F. Neese, *J. Comput. Chem.* **2022**, 1. https://doi.org/10.1002/jcc.26942