Received: 27 July 2020 | Revised: 7 September 2020 | Accepted: 11 September 2020

DOI: 10.1002/jcc.26425

FULL PAPER

Journal of COMPUTATIONAL CHEMISTRY WILEY

General recurrence-relation generation scheme for molecular integral evaluation

Bin Gao

Hylleraas Center for Quantum Molecular Sciences, Department of Chemistry, University of Tromsø The Arctic University of Norway, Tromsø, Norway

Correspondence
Bin Gao, Hylleraas Center for Quantum Molecular Sciences, Department of Chemistry, University of Tromsø The Arctic University of Norway, N–9037 Tromsø, Norway.
Email: bin.gao@uit.no

Funding information
Norges Forskningsråd, Grant/Award Number: 262695; Norwegian Supercomputer Program, Grant/Award Number: NN4654K

Abstract

We develop a new scheme for evaluating different molecular integrals using Gaussian type orbitals. In this new scheme, the evaluation of integrals is performed in two steps during runtime. The first step is a top-down procedure that maps each recurrence relation into a jagged array (array of arrays), where each element of a member array represents either the final results or some intermediate integrals that are stored in our developed data structure "coarse-grained circular buffer". This step is the same for all different one- and two-electron operators so that the same algorithm and source codes can be used. In the second step, a bottom-up procedure is carried out that computes all the intermediate and the final molecular integrals by backtracking elements from the last member array of each jagged array. Different source codes should in principle be used for different electron operators in the second step, but which can be generated automatically by our developed recurrence-relation compiler. The currently proposed general recurrence-relation generation scheme provides a new, generic and automatic programming way for various one- and two-electron integrals needed in computational chemistry. Users can even introduce new electron operators and evaluate their integrals during runtime by combining the implementation of the proposed new scheme and the just-in-time compilation technique.

KEYWORDS

code generation, Gaussian type orbital, Hermite Gaussian, integral, recurrence relation

1 | INTRODUCTION

One fundamental task in computational-chemistry calculations is the evaluation of various one- and two-electron integrals and their derivatives over atomic-orbital basis sets, which are usually represented by Gaussian type orbitals for their efficiency in integral evaluation. Currently, almost all molecular integrals are calculated using recurrence relations—one starts from a simple integral that can be computed easily, and from which the final results are calculated by recursively using some mathematical formulas.

Even though one may program each integral in an efficient way using the Obara–Saika,[1,2] McMurchie–Davidson,[3] or Rys quadrature[4] schemes, a unified computational procedure for evaluating these integrals and their derivatives is nevertheless valuable, especially when exploring higher-order molecular properties with the recently proposed open-ended quasienergy derivative approach[5] where a large number of different complicated integrals are prerequisite for the calculations of molecular properties.

Take one-electron integrals for example, many groups have contributed to the generalization of evaluating various one-electron

integrals—see for example, publications[2,6–12] and references therein. We have also recently proposed a procedure for evaluating one-electron integrals and their geometrical derivatives by using a generalized one-electron operator, in which an arbitrary central-potential operator $f(|\boldsymbol{r}-\boldsymbol{C}|)$ around center $\boldsymbol{C}$ can be chosen for different one-electron operators,[13] for instance $f(|\boldsymbol{r}-\boldsymbol{C}|) = |\boldsymbol{r}-\boldsymbol{C}|^{-1}$, $|\boldsymbol{r}-\boldsymbol{C}|^{-2}$, and Dirac delta function $\delta(\boldsymbol{r}-\boldsymbol{C})$.

The aforementioned contributions have nicely illustrated the generalization of various integrals and their derivatives, but what is missing is the generalization of programming different recurrence relations for different one- and two-electron operators. For instance, one will have different recurrence relations for the potential $|\boldsymbol{r}-\boldsymbol{C}|^{-1}$ and the Dirac delta function $\delta(\boldsymbol{r}-\boldsymbol{C})$, and correspondingly different algorithms and source codes have to be written for these two different operators. One can program different recurrence relations manually, but such codes can be prone to error and need to be totally rewritten whenever any change is required in the corresponding recurrence relations—for instance, due to efficiency or stability reason. Manual programming therefore requires much effort for the implementation of integrals of new electron operators, and for the maintenance and the improvement of existing integral evaluation algorithms and codes.

Most modern programming languages provide recursive function that allows programmers to express operations in terms of the function itself, and therefore reduces the effort of programming recurrence relations. However, source codes using recursive functions are often inefficient for the integral evaluation, because it is not trivial to reuse computed intermediate integrals during recurrence relations.[13] Reusing intermediate results is of key importance for saving computation time in the integral evaluation, in particular for integrals with high angular momentum basis sets and/or higher order (geometrical) derivatives. Therefore, instead of using the recursive function, most efficient integral codes are currently either manually programmed or generated using the automatic programming technique. In the latter, the actual codes (mostly using C, C++, or Fortran languages) are generated from a set of codes (named as integral code generator thereafter) written at a higher abstraction level using, for instance Python language.

The automatic programming technique therefore reduces the programming effort to some extent. However, we note that most integral code generators cannot treat, for instance arbitrary angular momentum and/or arbitrary order of (geometrical) derivatives. Moreover, different integral code generators have to be written for different forms of electron operators. All these limitations again restrict the development of new molecular integrals and studies of new molecular properties, or one has to dedicate much effort on such development.

Therefore, the current contribution aims to develop a new scheme to reduce the programming effort for the integral evaluation of different electron operators. We name the new scheme as "general recurrence-relation generation scheme", which divides the integral evaluation into two steps: All recurrence relations are first mapped into a series of jagged arrays in which each element of a member array represents either intermediate integrals or the final results; Secondly, all the intermediate and the final integrals are computed by backtracking elements from the last member array of each jagged array. We have also developed a data structure "coarse-grained circular buffer", which together with the jagged arrays guarantee the reuse of all intermediate results and also efficient use of computer memory. A general recurrence-relation compiler has also been developed for the second step, so that the new scheme can work for almost all physically relevant molecular integrals and their derivatives. More exactly, our recurrence-relation compiler can for the time being handle different multi-index recurrence relations in Equation (25) of order $(t_1, \cdots, t_k, \cdots, t_q)$ where $0 \le t_p \le 2$ $(p \ne k)$ and $1 \le t_k \le 2$.

The remainder of this paper is organized as follows: we first present our notation conventions and theoretical background for the integral evaluation in computational chemistry. The general recurrence-relation generation scheme is described afterwards, as well as its design and implementation. Finally, we discuss the performance of the proposed scheme by using different examples and give our final concluding remarks.

2 | THEORY

2.1 | Notation conventions

Let us first define our notation conventions: A bold capital letter such as $\boldsymbol{R}_\kappa$ denotes the position of a nucleus (or a center) $\boldsymbol{\kappa}$. The vector from $\boldsymbol{R}_\lambda$ to $\boldsymbol{R}_\kappa$ is denoted by $\boldsymbol{R}_{\kappa\lambda} = \boldsymbol{R}_\kappa - \boldsymbol{R}_\lambda$. The capital letters X_κ, Y_κ, and Z_κ represent the Cartesian coordinates of a nucleus (or a center) at the position $\boldsymbol{R}_\kappa$, whereas R_κ denotes the norm of the vector $\boldsymbol{R}_\kappa$. The position of an electron relative to a nucleus (or a center) at the position $\boldsymbol{R}_\kappa$ is given by $\boldsymbol{r}_\kappa = \boldsymbol{r} - \boldsymbol{R}_\kappa$. Small letters x_κ, y_κ and z_κ, and r_κ denote the three Cartesian coordinates of the electron relative to the position $\boldsymbol{R}_\kappa$, and the norm of the vector $\boldsymbol{r}_\kappa$, respectively.

Moreover, we use the multi-index notation extensively[14] to simplify the expressions for the recurrence relations. For instance, the $|\boldsymbol{K}|$-th order geometrical derivatives with respect to a center at the position $\boldsymbol{R}_\kappa$ will be written as

$$\partial^{\boldsymbol{K}}_{\boldsymbol{R}_\kappa} = \left(\frac{\partial}{\partial X_\kappa}\right)^{K_X}\left(\frac{\partial}{\partial Y_\kappa}\right)^{K_Y}\left(\frac{\partial}{\partial Z_\kappa}\right)^{K_Z} = \frac{\partial^{|\boldsymbol{K}|}}{\partial X_\kappa^{K_X}\partial Y_\kappa^{K_Y}\partial Z_\kappa^{K_Z}}, \tag{1}$$

where the three-dimensional multi-index $\boldsymbol{K} = (K_X, K_Y, K_Z)^T$ is a vector of nonnegative integers and $|\boldsymbol{K}| = K_X + K_Y + K_Z$ is the norm (length) of the multi-index $\boldsymbol{K}$.

For different recurrence relations, we will often use $\boldsymbol{e}_\xi$ for the increment along one Cartesian direction ξ, where $\xi = x, y$ or z.

2.2 | Integral evaluation

The integrals we consider are evaluated over either the contracted real solid-harmonic Gaussian type orbitals (GTOs) located at one center $\boldsymbol{R}_\kappa$ with the orbital quantum number l_κ

$$\chi_\kappa^{\mathrm{SGTO}}(\boldsymbol{r}) = S_{l_\kappa m_\kappa}(\boldsymbol{r}_\kappa)\sum_i w_{i\kappa}^{\mathrm{SGTO}}\exp\left(-a_{i\kappa}r_\kappa^2\right),\ -l_\kappa \le m_\kappa \le l_\kappa, \tag{2}$$

or the contracted Cartesian GTOs

$$\chi_\kappa^{\mathrm{CGTO}}(\boldsymbol{r}) = \boldsymbol{r}_\kappa^{\boldsymbol{l}_\kappa}\sum_i w_{i\kappa}^{\mathrm{CGTO}}\exp\left(-a_{i\kappa}r_\kappa^2\right),\ |\boldsymbol{l}_\kappa| = l_\kappa, \tag{3}$$

where $w_{i\kappa}^{\mathrm{SGTO}}$ and $w_{i\kappa}^{\mathrm{CGTO}}$ are the radial contraction coefficients, and each real solid-harmonic or Cartesian GTO in the summation is called primitive GTO with $a_{i\kappa}$ being the orbital exponents. $S_{l_\kappa m_\kappa}(\boldsymbol{r}_\kappa)$ is the real solid-harmonic function, which satisfies the following transformation.[15]

$$S_{l_\kappa m_\kappa}(\boldsymbol{r}_\kappa)\exp\left(-a_{i\kappa}r_\kappa^2\right) = \sum_{|\boldsymbol{l}_\kappa|=l_\kappa} S_{\boldsymbol{l}_\kappa}^{l_\kappa m_\kappa}\boldsymbol{r}_\kappa^{\boldsymbol{l}_\kappa}\exp\left(-a_{i\kappa}r_\kappa^2\right), \tag{4}$$

with $S_{\boldsymbol{l}_\kappa}^{l_\kappa m_\kappa}$ being the transformation coefficients.

Reine et al.[16] have proven that the real solid-harmonic GTOs can also be represented by the Hermite Gaussian functions

$$H_\kappa^{\boldsymbol{l}_\kappa}(\boldsymbol{r}) = (2a_{i\kappa})^{-|\boldsymbol{l}_\kappa|}\partial_{\boldsymbol{R}_\kappa}^{\boldsymbol{l}_\kappa}\exp\left(-a_{i\kappa}r_\kappa^2\right), \tag{5}$$

with the same transformation coefficients of Equation (4)

$$S_{l_\kappa m_\kappa}(\boldsymbol{r}_\kappa)\exp\left(-a_{i\kappa}r_\kappa^2\right) = \sum_{|\boldsymbol{l}_\kappa|=l_\kappa} S_{\boldsymbol{l}_\kappa}^{l_\kappa m_\kappa}H_\kappa^{\boldsymbol{l}_\kappa}(\boldsymbol{r}). \tag{6}$$

This enables us to evaluate both the integrals and their geometrical derivatives on a common footing by using the Hermite Gaussian functions.[13,16]

Take a one-electron operator $\hat{O}(\{\boldsymbol{r}_{C_\alpha}\})$ for example—here $\{\boldsymbol{r}_{C_\alpha}\}$ represent a set of vectors relative to centers $\boldsymbol{C}_\alpha$ (α = 1, 2, ⋯), the geometrical derivatives of its integrals over contracted GTOs $\chi_\kappa(\boldsymbol{r})$ and $\chi_\lambda(\boldsymbol{r})$ are denoted as

$$[\boldsymbol{L}_\kappa\boldsymbol{L}_\lambda\{\boldsymbol{L}_\alpha\}\boldsymbol{l}_\kappa\boldsymbol{l}_\lambda]_{\hat{O}} = \partial_{\boldsymbol{R}_\kappa}^{\boldsymbol{L}_\kappa}\partial_{\boldsymbol{R}_\lambda}^{\boldsymbol{L}_\lambda}\left(\circ_\alpha\partial_{\boldsymbol{C}_\alpha}^{\boldsymbol{L}_\alpha}\right)\int\chi_\kappa(\boldsymbol{r})\hat{O}(\{\boldsymbol{r}_{C_\alpha}\})\chi_\lambda(\boldsymbol{r})\mathrm{d}\boldsymbol{r}, \tag{7}$$

where $\boldsymbol{l}_\kappa$ and $\boldsymbol{l}_\lambda$ are respectively the orbital quantum numbers of contracted GTOs $\chi_\kappa(\boldsymbol{r})$ and $\chi_\lambda(\boldsymbol{r})$, $\left(\circ_\alpha\partial_{\boldsymbol{C}_\alpha}^{\boldsymbol{L}_\alpha}\right) \equiv \partial_{\boldsymbol{C}_1}^{\boldsymbol{L}_1}\partial_{\boldsymbol{C}_2}^{\boldsymbol{L}_2}\cdots$ represents a set of derivatives with respect to the centers $\boldsymbol{C}_\alpha$ (α = 1, 2, ⋯). Replacing $\chi_\kappa(\boldsymbol{r})$ and $\chi_\lambda(\boldsymbol{r})$ with the contracted real solid-harmonic GTOs (2) or the contracted Cartesian GTOs (3), we have

$$\begin{aligned}[\boldsymbol{L}_\kappa\boldsymbol{L}_\lambda\{\boldsymbol{L}_\alpha\}\boldsymbol{l}_\kappa\boldsymbol{l}_\lambda]_{\hat{O}}^{\mathrm{SGTO}} &= \sum_{|\boldsymbol{l}_\kappa|=l_\kappa,|\boldsymbol{l}_\lambda|=l_\lambda} S_{\boldsymbol{l}_\kappa}^{l_\kappa m_\kappa}S_{\boldsymbol{l}_\lambda}^{l_\lambda m_\lambda}\sum_{ij} w_{i\kappa}^{\mathrm{SGTO}}w_{j\lambda}^{\mathrm{SGTO}} \\ &\times(2a_{i\kappa})^{|\boldsymbol{L}_\kappa|}(2b_{j\lambda})^{|\boldsymbol{L}_\lambda|}[\{\boldsymbol{L}_\alpha\},\boldsymbol{L}_\kappa+\boldsymbol{l}_\kappa,\boldsymbol{L}_\lambda+\boldsymbol{l}_\lambda]_{\hat{O}}^{\mathrm{HGTO}},\end{aligned} \tag{8}$$

$$\begin{aligned}[\boldsymbol{L}_\kappa\boldsymbol{L}_\lambda\{\boldsymbol{L}_\alpha\}\boldsymbol{l}_\kappa\boldsymbol{l}_\lambda]_{\hat{O}}^{\mathrm{CGTO}} &= \sum_{ij} w_{i\kappa}^{\mathrm{CGTO}}w_{j\lambda}^{\mathrm{CGTO}} \\ &\times\partial_{\boldsymbol{R}_\kappa}^{\boldsymbol{L}_\kappa}\partial_{\boldsymbol{R}_\lambda}^{\boldsymbol{L}_\lambda}\left(\circ_\alpha\partial_{\boldsymbol{C}_\alpha}^{\boldsymbol{L}_\alpha}\right)\int\boldsymbol{r}_\kappa^{\boldsymbol{l}_\kappa}\mathrm{e}^{-a_{i\kappa}r_\kappa^2}\hat{O}(\{\boldsymbol{r}_{C_\alpha}\})\boldsymbol{r}_\lambda^{\boldsymbol{l}_\lambda}\mathrm{e}^{-b_{j\lambda}r_\lambda^2}\mathrm{d}\boldsymbol{r},\end{aligned} \tag{9}$$

where we have introduced the basic integrals over primitive Hermite Gaussian functions

$$[\{\boldsymbol{L}_\alpha\}\boldsymbol{l}_\kappa\boldsymbol{l}_\lambda]_{\hat{O}}^{\mathrm{HGTO}} = \frac{\partial_{\boldsymbol{R}_\kappa}^{\boldsymbol{l}_\kappa}}{(2a_{i\kappa})^{|\boldsymbol{l}_\kappa|}}\frac{\partial_{\boldsymbol{R}_\lambda}^{\boldsymbol{l}_\lambda}}{(2b_{j\lambda})^{|\boldsymbol{l}_\lambda|}}\left(\circ_\alpha\partial_{\boldsymbol{C}_\alpha}^{\boldsymbol{L}_\alpha}\right)\int\exp\left(-a_{i\kappa}r_\kappa^2\right)\hat{O}(\{\boldsymbol{r}_{C_\alpha}\})\exp\left(-b_{j\lambda}r_\lambda^2\right)\mathrm{d}\boldsymbol{r}. \tag{10}$$

Notice that

$$\begin{aligned}\boldsymbol{r}_\kappa^{\boldsymbol{l}_\kappa+\boldsymbol{e}_\xi}\mathrm{e}^{-a_{i\kappa}r_\kappa^2} &= \boldsymbol{r}_\kappa^{\boldsymbol{l}_\kappa}\left(\frac{\partial_{\boldsymbol{R}_\kappa}^{\boldsymbol{e}_\xi}}{2a_{i\kappa}}\right)\mathrm{e}^{-a_{i\kappa}r_\kappa^2} = \frac{\partial_{\boldsymbol{R}_\kappa}^{\boldsymbol{e}_\xi}}{2a_{i\kappa}}\left(\boldsymbol{r}_\kappa^{\boldsymbol{l}_\kappa}\mathrm{e}^{-a_{i\kappa}r_\kappa^2}\right) - \left(\frac{\partial_{\boldsymbol{R}_\kappa}^{\boldsymbol{e}_\xi}}{2a_{i\kappa}}\boldsymbol{r}_\kappa^{\boldsymbol{l}_\kappa}\right)\mathrm{e}^{-a_{i\kappa}r_\kappa^2} \\ &= \frac{\partial_{\boldsymbol{R}_\kappa}^{\boldsymbol{e}_\xi}}{2a_{i\kappa}}\left(\boldsymbol{r}_\kappa^{\boldsymbol{l}_\kappa}\mathrm{e}^{-a_{i\kappa}r_\kappa^2}\right) + \frac{(\boldsymbol{l}_\kappa)_\xi}{2a_{i\kappa}}\boldsymbol{r}_\kappa^{\boldsymbol{l}_\kappa-\boldsymbol{e}_\xi}\mathrm{e}^{-a_{i\kappa}r_\kappa^2},\end{aligned} \tag{11}$$

we could further transfer $\boldsymbol{l}_\kappa$ to $\boldsymbol{L}_\kappa$ for each integral over primitive Cartesian GTOs in Equation (9), using the following recurrence relation

$$\begin{aligned}&\partial_{\boldsymbol{R}_\kappa}^{\boldsymbol{L}_\kappa}\partial_{\boldsymbol{R}_\lambda}^{\boldsymbol{L}_\lambda}\left(\circ_\alpha\partial_{\boldsymbol{C}_\alpha}^{\boldsymbol{L}_\alpha}\right)\int\boldsymbol{r}_\kappa^{\boldsymbol{l}_\kappa+\boldsymbol{e}_\xi}\mathrm{e}^{-a_{i\kappa}r_\kappa^2}\hat{O}(\{\boldsymbol{r}_{C_\alpha}\})\boldsymbol{r}_\lambda^{\boldsymbol{l}_\lambda}\mathrm{e}^{-b_{j\lambda}r_\lambda^2}\mathrm{d}\boldsymbol{r} \\ &= \frac{1}{2a_{i\kappa}}[\partial_{\boldsymbol{R}_\kappa}^{\boldsymbol{L}_\kappa+\boldsymbol{e}_\xi}\partial_{\boldsymbol{R}_\lambda}^{\boldsymbol{L}_\lambda}\left(\circ_\alpha\partial_{\boldsymbol{C}_\alpha}^{\boldsymbol{L}_\alpha}\right)\int\boldsymbol{r}_\kappa^{\boldsymbol{l}_\kappa}\mathrm{e}^{-a_{i\kappa}r_\kappa^2}\hat{O}(\{\boldsymbol{r}_{C_\alpha}\})\boldsymbol{r}_\lambda^{\boldsymbol{l}_\lambda}\mathrm{e}^{-b_{j\lambda}r_\lambda^2}\mathrm{d}\boldsymbol{r} \\ &+ (\boldsymbol{l}_\kappa)_\xi\partial_{\boldsymbol{R}_\kappa}^{\boldsymbol{L}_\kappa-\boldsymbol{e}_\xi}\partial_{\boldsymbol{R}_\lambda}^{\boldsymbol{L}_\lambda}\left(\circ_\alpha\partial_{\boldsymbol{C}_\alpha}^{\boldsymbol{L}_\alpha}\right)\int\boldsymbol{r}_\kappa^{\boldsymbol{l}_\kappa-\boldsymbol{e}_\xi}\mathrm{e}^{-a_{i\kappa}r_\kappa^2}\hat{O}(\{\boldsymbol{r}_{C_\alpha}\})\boldsymbol{r}_\lambda^{\boldsymbol{l}_\lambda}\mathrm{e}^{-b_{j\lambda}r_\lambda^2}\mathrm{d}\boldsymbol{r}],\end{aligned} \tag{12}$$

and likewise for transferring $\boldsymbol{l}_\lambda$ to $\boldsymbol{L}_\lambda$.

After performing the recurrence relation (12) for each integral over primitive Cartesian GTOs, we will also arrive at the following basic integrals over primitive Hermite Gaussian functions

$$\partial_{\boldsymbol{R}_\kappa}^{\boldsymbol{L}'_\kappa}\partial_{\boldsymbol{R}_\lambda}^{\boldsymbol{L}'_\lambda}\left(\circ_\alpha\partial_{\boldsymbol{C}_\alpha}^{\boldsymbol{L}_\alpha}\right)\int\mathrm{e}^{-a_{i\kappa}r_\kappa^2}\hat{O}(\{\boldsymbol{r}_{C_\alpha}\})\mathrm{e}^{-b_{j\lambda}r_\lambda^2}\mathrm{d}\boldsymbol{r} = (2a_{i\kappa})^{|\boldsymbol{L}'_\kappa|}(2b_{j\lambda})^{|\boldsymbol{L}'_\lambda|}[\{\boldsymbol{L}_\alpha\}\boldsymbol{L}'_\kappa\boldsymbol{L}'_\lambda]_{\hat{O}}^{\mathrm{HGTO}}. \tag{13}$$

Therefore, the prerequisite for calculating integrals over either the contracted real solid-harmonic GTOs or the contracted Cartesian GTOs is the evaluation of integrals (10).

2.3 | Recurrence relations

The evaluation of integrals (10) however depends on the knowledge of explicit form of the operator $\hat{O}(\{\boldsymbol{r}_{C_\alpha}\})$. For instance, integrals of the Cartesian multipole moment operator $\hat{O}(\{\boldsymbol{r}_{C_\alpha}\}) = \boldsymbol{r}_M^{\boldsymbol{m}}$ (dipole origin $\boldsymbol{M}$) can be evaluated using the recurrence relations.[13,16]

$$\begin{aligned}[\boldsymbol{l}_\kappa\boldsymbol{l}_\lambda,\boldsymbol{m}+\boldsymbol{e}_\xi]_{\hat{O}}^{\mathrm{HGTO}} &= \left(\boldsymbol{R}_{\gamma M}\right)_\xi[\boldsymbol{l}_\kappa\boldsymbol{l}_\lambda\boldsymbol{m}]_{\hat{O}}^{\mathrm{HGTO}} + \frac{1}{2p_{ij}}((\boldsymbol{l}_\kappa)_\xi[\boldsymbol{l}_\kappa-\boldsymbol{e}_\xi,\boldsymbol{l}_\lambda\boldsymbol{m}]_{\hat{O}}^{\mathrm{HGTO}} \\ &+ (\boldsymbol{l}_\lambda)_\xi[\boldsymbol{l}_\kappa,\boldsymbol{l}_\lambda-\boldsymbol{e}_\xi,\boldsymbol{m}]_{\hat{O}}^{\mathrm{HGTO}} + \boldsymbol{m}_\xi[\boldsymbol{l}_\kappa\boldsymbol{l}_\lambda,\boldsymbol{m}-\boldsymbol{e}_\xi]_{\hat{O}}^{\mathrm{HGTO}}),\end{aligned} \tag{14}$$

$$[\boldsymbol{l}_\kappa,\boldsymbol{l}_\lambda+\boldsymbol{e}_\xi,\boldsymbol{0}]_{\hat{O}}^{\mathrm{HGTO}} = -\frac{a_{i\kappa}}{b_{j\lambda}}[\boldsymbol{l}_\kappa+\boldsymbol{e}_\xi,\boldsymbol{l}_\lambda\boldsymbol{0}]_{\hat{O}}^{\mathrm{HGTO}}, \tag{15}$$

$$[\boldsymbol{l}_\kappa + \boldsymbol{e}_\xi, \boldsymbol{00}]_{\hat{O}}^{\mathrm{HGTO}} = \left(\boldsymbol{R}_{\gamma\kappa}\right)_\xi [\boldsymbol{l}_\kappa \boldsymbol{00}]_{\hat{O}}^{\mathrm{HGTO}} - \frac{1}{2p_{ij}} \frac{(\boldsymbol{l}_\kappa)_\xi b_{j\lambda}}{a_{i\kappa}} [\boldsymbol{l}_\kappa - \boldsymbol{e}_\xi, \boldsymbol{00}]_{\hat{O}}^{\mathrm{HGTO}}, \quad (16)$$

and starting from

$$[\boldsymbol{000}]_{\hat{O}}^{\mathrm{HGTO}} = \mathrm{e}^{-u_{ij} R_{\kappa\lambda}^2} \int \mathrm{e}^{-p_{ij} r_\gamma^2} \mathrm{d}\boldsymbol{r} = \mathrm{e}^{-u_{ij} R_{\kappa\lambda}^2} \left(\frac{\pi}{p_{ij}}\right)^{\frac{3}{2}}, \quad (17)$$

where

$$p_{ij} = a_{i\kappa} + b_{j\lambda}, \quad (18)$$

$$u_{ij} = \frac{a_{i\kappa} b_{j\lambda}}{p_{ij}}, \quad (19)$$

$$\boldsymbol{R}_\gamma = \frac{a_{i\kappa} \boldsymbol{R}_\kappa + b_{j\lambda} \boldsymbol{R}_\lambda}{p_{ij}}. \quad (20)$$

While for the nuclear attraction potential $\hat{O}(\{\boldsymbol{r}_{C_\alpha}\}) = \partial_{\boldsymbol{C}}^{L_C} r_C^{-1}$, different recurrence relations have to be used.[13]

$$[\boldsymbol{l}_\kappa + \boldsymbol{e}_\xi, \boldsymbol{l}_\lambda \boldsymbol{L}_C; 0]_{\hat{O}}^{\mathrm{HGTO}} = \left(\boldsymbol{R}_{\gamma\kappa}\right)_\xi [\boldsymbol{l}_\kappa \boldsymbol{l}_\lambda \boldsymbol{L}_C; 0]_{\hat{O}}^{\mathrm{HGTO}} - \frac{1}{2p_{ij}} \left(\frac{(\boldsymbol{l}_\kappa)_\xi b_{j\lambda}}{a_{i\kappa}} [\boldsymbol{l}_\kappa - \boldsymbol{e}_\xi, \boldsymbol{l}_\lambda \boldsymbol{L}_C; 0]_{\hat{O}}^{\mathrm{HGTO}} - (\boldsymbol{l}_\lambda)_\xi [\boldsymbol{l}_\kappa, \boldsymbol{l}_\lambda - \boldsymbol{e}_\xi, \boldsymbol{L}_C; 0]_{\hat{O}}^{\mathrm{HGTO}} + [\boldsymbol{l}_\kappa \boldsymbol{l}_\lambda, \boldsymbol{L}_C + \boldsymbol{e}_\xi; 0]_{\hat{O}}^{\mathrm{HGTO}}\right), \quad (21)$$

$$[\boldsymbol{0}, \boldsymbol{l}_\lambda + \boldsymbol{e}_\xi, \boldsymbol{L}_C; 0]_{\hat{O}}^{\mathrm{HGTO}} = \left(\boldsymbol{R}_{\gamma\lambda}\right)_\xi [\boldsymbol{0} \boldsymbol{l}_\lambda \boldsymbol{L}_C; 0]_{\hat{O}}^{\mathrm{HGTO}} - \frac{1}{2p_{ij}} \left(\frac{(\boldsymbol{l}_\lambda)_\xi a_{i\kappa}}{b_{j\lambda}} [\boldsymbol{0}, \boldsymbol{l}_\lambda - \boldsymbol{e}_\xi, \boldsymbol{L}_C; 0]_{\hat{O}}^{\mathrm{HGTO}} + [\boldsymbol{0} \boldsymbol{l}_\lambda, \boldsymbol{L}_C + \boldsymbol{e}_\xi; 0]_{\hat{O}}^{\mathrm{HGTO}}\right), \quad (22)$$

$$[\boldsymbol{00}, \boldsymbol{L}_C + \boldsymbol{e}_\xi; n_0]_{\hat{O}}^{\mathrm{HGTO}} = \left(\boldsymbol{R}_{C_\gamma}\right)_\xi [\boldsymbol{00} \boldsymbol{L}_C; n_0 + 1]_{\hat{O}}^{\mathrm{HGTO}} + (\boldsymbol{L}_C)_\xi [\boldsymbol{00}, \boldsymbol{L}_C - \boldsymbol{e}_\xi; n_0 + 1]_{\hat{O}}^{\mathrm{HGTO}}, \quad (23)$$

and

$$[\boldsymbol{000}; n_0]_{\hat{O}}^{\mathrm{HGTO}} = \mathrm{e}^{-u_{ij} R_{\kappa\lambda}^2} \frac{2\pi}{p_{ij}} \left(-2p_{ij}\right)^{n_0} F_{n_0}\left(p_{ij} R_{C_\gamma}^2\right), \quad (24)$$

where $F_{n_0}\left(p_{ij} R_{C_\gamma}^2\right)$ is the n_0-th order Boys function.[15]

Generally, one has to manually program different source codes or prepare different recurrence-relation code generators for the recurrence relations of different electron operators. However, as will be shown in the following section, it becomes possible to evaluate different recurrence relations of various electron operators with our developed "general recurrence-relation generation scheme".

3 | SCHEME, DATA STRUCTURE AND ALGORITHM

Before presenting the new scheme, we give a more formal definition of our interested recurrence relations—q-indexed recurrence relation of order $(t_1, \cdots, t_k, \cdots, t_q)$ as.[17]

$$[\boldsymbol{i}_1, \cdots, \boldsymbol{i}_k + \boldsymbol{e}_\xi, \cdots, \boldsymbol{i}_q] = \sum_{r_1=-t_1}^{t_1} \cdots \sum_{r_k=0}^{t_k-1} \cdots \sum_{r_q=-t_q}^{t_q} \boldsymbol{a}_\xi^{r_1,\cdots r_q} \left[\boldsymbol{i}_1 + \sum_{\zeta_1} \boldsymbol{e}_{\zeta_1}^{\pm}, \cdots, \boldsymbol{i}_k - \sum_{\zeta_k} \boldsymbol{e}_{\zeta_k}, \cdots, \boldsymbol{i}_q + \sum_{\zeta_q} \boldsymbol{e}_{\zeta_q}^{\pm}\right], \quad (25)$$

where

$$\boldsymbol{e}_{\zeta_p}^{\pm} = \boldsymbol{e}_{\zeta_p} \text{ or } -\boldsymbol{e}_{\zeta_p}, \quad (26)$$

$$t_k \geq 1, \ \sum_{\zeta_k} \left|\boldsymbol{e}_{\zeta_k}\right| = r_k, \quad (27)$$

$$t_p \geq 0, \ \sum_{\zeta_p} \left|\boldsymbol{e}_{\zeta_p}^{\pm}\right| = r_p, \ (p = 1, \cdots, k-1, k+1, \cdots, q), \quad (28)$$

and $\boldsymbol{a}_\xi^{r_1,\cdots r_q}$ are coefficients of the right-hand-side (RHS) terms, which can be constants or variables depending on the indices. We further name the index $\boldsymbol{i}_k$ as "output index", indices $\boldsymbol{i}_1$ to $\boldsymbol{i}_{k-1}$ are called "inner indices", and $\boldsymbol{i}_{k+1}$ to $\boldsymbol{i}_q$ are "outer indices".

For instance, the recurrence relation (14) is a 3-indexed recurrence relation of order (1, 1, 2), and the output index is $\boldsymbol{m}$, the inner indices are $\boldsymbol{l}_\kappa$ and $\boldsymbol{l}_\lambda$, and there is no outer indices.

The integral evaluation using different multi-indexed recurrence relations of different orders can fall into two steps: a top-down procedure and followed by a bottom-up procedure, which will be described in the following two subsections respectively.

3.1 | Top-down procedure

Take the evaluation of the Cartesian multipole moment integrals $[\boldsymbol{l}_\kappa \boldsymbol{l}_\lambda \boldsymbol{m}] = [\boldsymbol{123}]$ for example—hereafter we abbreviate $[\cdots]_{\hat{O}}^{\mathrm{HGTO}}$ as $[\cdots]$, the first step is a top-down procedure as shown in Figure 1, where the recurrence relations (14)–(16) are used to find their RHS terms of the target integrals on the left hand side (LHS). For instance, according to the recurrence relation (14), the target integrals [**111**] connects to the RHS terms [**110**], [**010**] and [**100**] by solid arrows as shown in Figure 1.

3.1.1 | Jagged Array

From the graph theory,[18] all integral terms of a recurrence relation can be readily described by the so-called "directed acyclic graph". Rák et al. have proposed a method to map an integral into a thread of a parallel architecture, where a directed graph has been used for the computation of the integral expressed as a summation.[19]

Different from the invention by Rák et al., we care more about the key information that can be delivered to and guide the next step—the practical integral computations. More exactly, we are interested in finding:

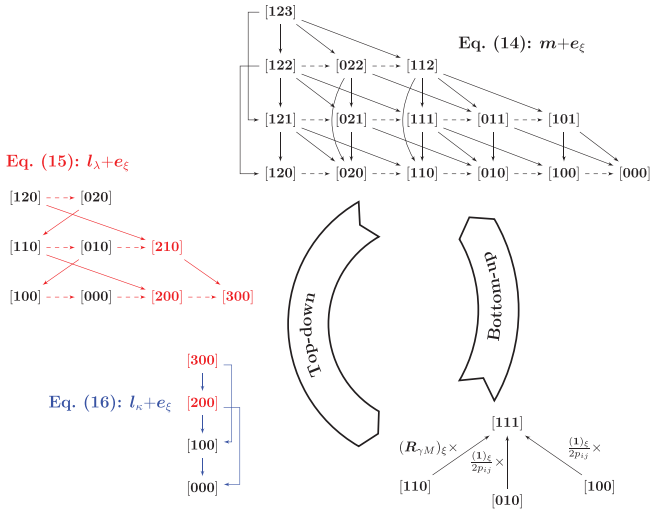

FIGURE 1 Top-down procedure: finding all necessary intermediate integrals and mapping information between the LHS and RHS terms for the Cartesian multipole moment integrals $[l_\kappa l_\lambda m]$ = [**123**]. Bottom-up procedure: computing all the intermediate integrals and the final results by following the arithmetic operations defined in the recurrence relations and starting from the integral [**000**] [Color figure can be viewed at wileyonlinelibrary.com]

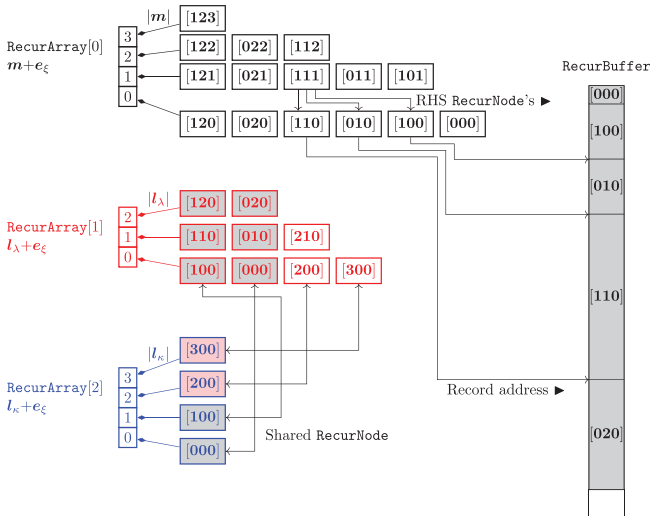

FIGURE 2 Illustration of the data structure `RecurArray` (jagged array) used for different recurrence relations. The example shown here is to compute the Cartesian multipole moment integrals $[l_\kappa l_\lambda m]$ = [**123**]. Intermediate integrals are stored (physically) in another data structure `RecurBuffer` (coarse-grained circular buffer) and their addresses are recorded in the corresponding `RecurNode`'s [Color figure can be viewed at wileyonlinelibrary.com]

- all integral terms $[\cdots]$ needed for the recurrence relations, and
- relationships among these integral terms as described by the recurrence relations.

Therefore, instead of the directed acyclic graph, we have chosen the other simpler data structure—jagged array to represent a recurrence relation. The implementation of the jagged array—named as `RecurArray`—has been made in our recently developed tIntegral library,[20] and C++ programming language has been chosen for the implementation. The tIntegral library is released under the Mozilla Public License (version 2.0) and a development version (version 1.0.0) is available at https://gitlab.com/tglue-chemistry/tintegral.

The rationale behind the design of the data structure `RecurArray` can be better understood from the evolvement of recurrence relations in the top-down procedure:

1. As shown in Figure 2, several `RecurArray`'s will be created during the top-down procedure that can be readily put into a sequence container like `std::array` in C++ programming language.
   These jagged arrays are arranged following the top-down manner, that is, the first jagged array represents the recurrence relation for the final molecular integrals while the last jagged array for the recurrence relation starting from the integrals $[\mathbf{0}\cdots\mathbf{0}]_O^{\text{HGTO}}$.
2. Each integral term of a recurrence relation is implemented by a class `RecurNode`, which contains information such as orders of indices, address of integrals in the other data structure coarse-grained circular buffer (named as `RecurBuffer` and will be discussed later in the current section) and RHS `RecurNode`'s.
   As shown in Figure 1, integral terms of each recurrence relation can be arranged into different levels—those connected by the dashed arrows have the same order of the output index, and can therefore be put into the same member array.
   Each member array can be realized by the sequence container `std::vector` in C++ programming language, which will gradually grow whenever new integral terms are found with the corresponding order of the output index.
3. Also as revealed in Figures 1 and 2, there are integral terms appearing in both the last member array of a jagged array and the succeeding jagged array. For instance, integral terms—from [**120**] to [**000**] of the first jagged array also appear in the second jagged array for the recurrence relation of l_λ.
   To efficiently handle the above "shared ownership" of integral terms, we have used the C++ smart shared pointer `std::shared_ptr` to manage each integral term, which has also been used for the management of RHS `RecurNode`'s of each integral term.

3.1.2 | Coarse-grained circular buffer

Now let us focus on the other important data structure—coarse-grained circular buffer, or `RecurBuffer` that will be used to store (intermediate) integrals physically in computer memory. The following requirements have to be considered for an efficient and non-conflicting use of the computer memory:

1. The amount of the required computer memory should be as less as possible, and whenever a `RecurNode` will not be involved in the

integral evaluation, the memory that it has used before should be released or be used by other `RecurNode`'s;
2. No conflicting use of the computer memory during the integral evaluation, that is, any `RecurNode` and its RHS `RecurNode`'s should use different parts of the computer memory;
3. Integral terms appearing in both the last member array of a jagged array and the succeeding jagged array need special consideration because their integrals can be viewed as the "input" of the former jagged array (recurrence relation) and the "output" of the succeeding recurrence relation; In other words, these integrals have to stay in the computer memory during the evaluation of both recurrence relations.

As will be discussed in the next subsection, the bottom-up procedure will be performed by following the order of the output index for each member array—starting from the first order up to a maximum order. Such a strategy and the aforementioned memory-usage requirements lead us to the following decisions for the design of the class `RecurBuffer`:

1. `RecurNode`'s of the same member array (i.e., with the same order of the output index) will use a consecutive segment of the computer memory;
2. `RecurNode`'s of different member arrays will usually use different segments to avoid conflicting usage of the memory—in particular those to be involved together in the corresponding recurrence relation;
3. However, a same segment of the computer memory could be used by member arrays that are not at the same time involved in the corresponding recurrence relation, which can reduce the amount of required memory;
4. For output terms of a recurrence relation, their used segments of the computer memory need to be reserved as the input of the preceding recurrence relation.

By following the above decisions and noticing the bottom-up procedure will be performed order by order, we can design the `RecurBuffer` as a coarse-grained version of the known data structure "circular buffer". That means, as illustrated in Figure 3, the `RecurBuffer` divides a portion of the computer memory into several segments (separated by solid lines), and each of them contains the molecular integrals of `RecurNode`'s in the same member array.

Let the maximum order of the output index $\boldsymbol{i}_k$ be $\max_{\text{order}}(\mathbf{i}_k)$ for a recurrence relation (25). The number of needed segments for the recurrence relation can be determined by

$$N_{\text{segments}} = \min[\max_{\text{order}}(\boldsymbol{i}_k), t_k] + 1, \tag{29}$$

which is not greater than the number of member arrays (=$\max_{\text{order}}(\mathbf{i}_k)$ + 1) so that less computer memory is required.

Meanwhile, we can ensure each `RecurNode` and all its RHS `RecurNode`'s will use different segments by using the computer memory in a cyclic manner, that is, only the segment used by a member array with the least order of the output index will be released or be used by another member array with greater order of the output index. As illustrated in Figure 3, the segements used by integral terms [**210**], [**200**] and [**300**] will be used again by the final results [**123**] when the former will not be involved in the evaluation of the recurrence relation.

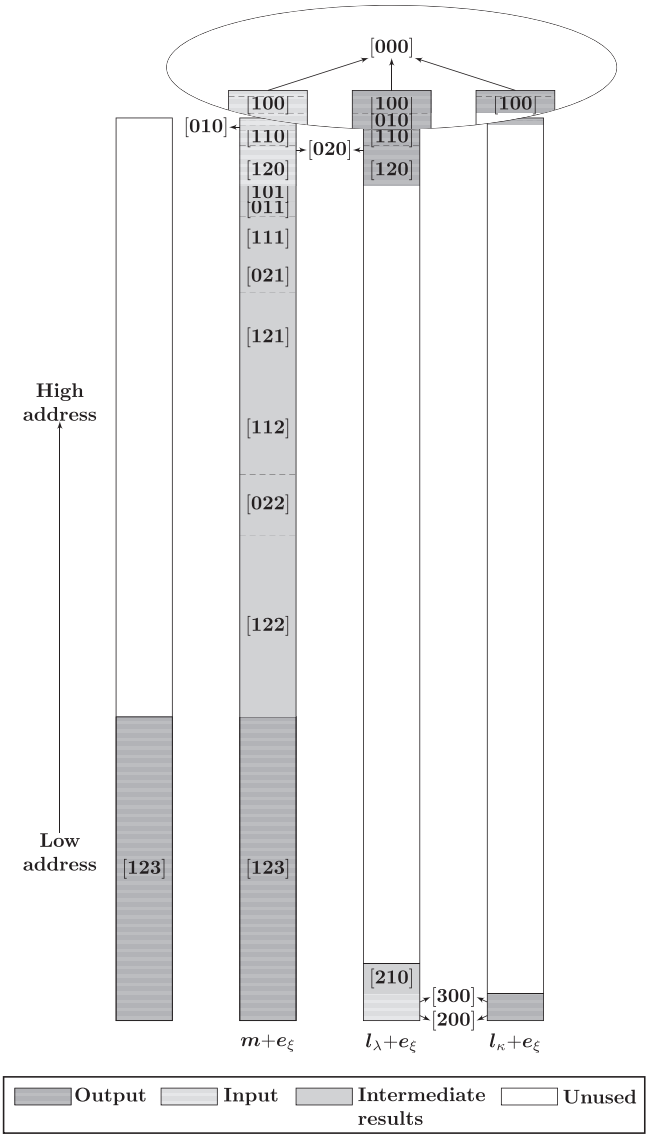

FIGURE 3 Coarse-grained circular buffer used for storing the final and intermediate integrals during recurrence relations, where integrals of `RecurNode`'s belonging to the same member array are stored in the same segment of the computer memory, and separated by dashsed lines

It therefore means that the `RecurBuffer` is well-suited as a first-in-first-out buffer and also serves well for one of our objectives—the first computed intermediate integrals could be firstly "kicked out".

Last but not least, to reserve the segment of output terms of a recurrence relation, we have implemented an important feature in the data structure `RecurBuffer`—the direction of data storage in the computer memory. Still take the evaluation of the Cartesian multipole

moment integrals [$l_\kappa l_\lambda m$] = [**123**] for example. The final integrals are stored in the computer memory from the lowest address of the `RecurBuffer` as shown in Figure 3, which should be reserved (untouched) when performing the recurrence relation (14) on ***m***.

A straightforward solution is to assign segments to integral terms in an opposite way for the recurrence relation (14), i.e., starting from the highest address of the `RecurBuffer`. As illustrated in Figure 3, `RecurNode`'s [**000**] to [**120**] therefore occupy the computer memory with the highest address, and are also arranged from higher address to lower one. The same procedure continues so that the direction of data storage always takes the opposite of the preceding recurrence relation.

To briefly summarize, the top-down procedure therefore needs (a) to generate a series of `RecurArray`'s for given recurrence relations and (b) to assign segments of a `RecurBuffer` to all member arrays. From the point of view of recurrence-relation performing, these tasks only require to manipulate different indices algebraically, and there is no integral computation happened. Therefore, the top-down procedure is general for all different recurrence relations and the same algorithm and source codes can therefore be developed.

3.2 | Bottom-up procedure

Still taking the evaluation of the Cartesian multipole moment integrals [$l_\kappa l_\lambda m$] = [**123**] for example, the second step of the recurrence-relation evaluation is a bottom-up procedure. For instance, the target integrals [**111**] are computed from already calculated RHS terms [**110**], [**010**] and [**100**] as shown in Figure 1.

The bottom-up procedure is carried out by backtracking `RecurNode`'s from the last member array of each `RecurArray`, and from the last `RecurArray` to the first one.

3.2.1 | Triangle-based recurrence relations

All integrals in the bottom-up procedure are treated following a triangle-based scheme in our current contribution. As shown in Figure 4, different Cartesian components—for instance, of GTOs and different derivatives—are arranged in a triangle with the $X\cdots X$, $Y\cdots Y$ and $Z\cdots Z$ components in the corners. Each integral term—[**111**], [**110**], [**010**] and [**100**]—actually contains integrals of a direct product of the corresponding triangles. The bottom-up procedure therefore does not manipulate a single number, but multiple components of the direct product of triangles.

Two problems in this step are (a) the ordering of these components, that is, how to arrange them in linear storage like the segment of the `RecurBuffer` and (b) the noninjective and surjective relationship between higher-order components and lower-order ones. As shown in Figure 5(a), although each second-order component can be calculated from *at least* one first-order component ("surjective"), there is not a *one-to-one* relationship ("injective") between them—from the value of the component *X*, one can calculate values of components

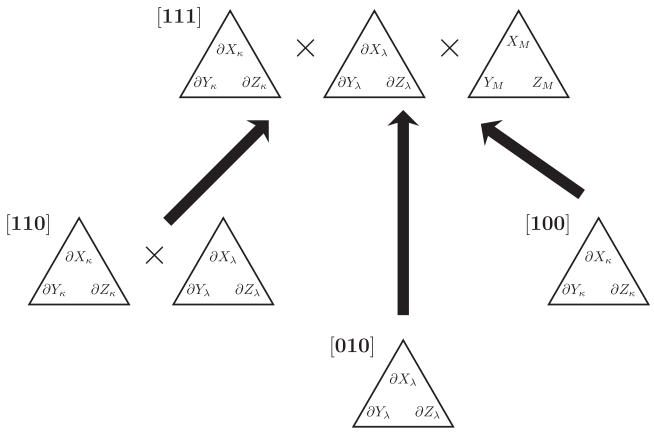

FIGURE 4 Triangle-based recurrence relations to get the Cartesian multipole moment integrals [**111**], where we have abbreviated $\frac{\partial^{l+m+n}}{\partial X^l \partial Y^m \partial Z^n}$ as $\partial X^l Y^m Z^n$

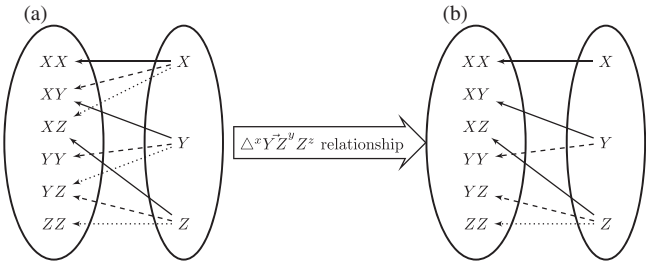

FIGURE 5 Noninjective and surjective relationship between components of the second- and first-order, where the solid, dashed and dotted arrows stands for recurrence relations along *x*, *y* and *z* directions respectively

XX, *XY* and *XZ* by performing recurrence relations along *x*, *y* and *z* directions respectively.

The ordering issue and the noninjective behavior are problematic for programming recurrence relations. All integral codes have chosen their own ordering and *one-to-one* relationship for practical implementation. For instance, the following component ordering and one-to-one recurrence relationship can be used together for programming:

Descending *XY*-major order where consecutive components along the $\vec{YZ}$ edge of a triangle are contiguous in memory;
$\triangle^x \vec{YZ}^y Z^z$ **recurrence relationship** where recurrence relations are first performed for all components in the lower-order triangle along the *x* direction, then along the *y* direction for components at the $\vec{YZ}$ edge of the lower-order triangle (components $Y\cdots Y$ to $Z\cdots Z$), and finally along the *z* direction for the component $Z\cdots Z$, as illustrated in Figure 5(b).

In the tIntegral library,[20] we have considered several combinations of the triangle-component ordering and its possible one-to-one recurrence relationship(s) as shown in Table 1. Such a combination can be provided as additional information to our general recurrence-

relation compiler (which will be introduced afterwards). It thus becomes straightforward to generate integral codes for different host computational-chemistry programs by choosing the appropriate combination or by implementing a new combination of the triangle-component ordering and its possible one-to-one recurrence relationship.

3.2.2 | Converting to loops

After choosing a combination in Table 1, we can focus on how to perform the bottom-up procedure for given recurrence relations. An efficient way is to convert the bottom-up procedure (of recurrence relations) to different loops:

1. Loop over different `RecurArray`'s starting from the last one.
2. Loop over different member arrays of a `RecurArray`, from one with the first order of the output index up to that with the maximum order (that with the zeroth order contains already calculated integrals).For instance, the loop will start from the first order to the third order of the output index $\boldsymbol{m}$ in the `RecurArray[0]` of Figure 2.
3. Loop over different `RecurNode`'s of a member array.
4. Loop over *XYZ* components of different indices by following a given triangle-component ordering and its one-to-one recurrence relationship.

The first three loops are much more straightforward to program than the last one, because the last loop usually depends on the exact form of the corresponding recurrence relation. Instead of manually programming, we will in the current contribution develop a general recurrence-relation compiler—`RecurCompiler` to automatically generate source codes of the above four loops for different multi-index recurrence relations given in the form of (25), and with the order (t_1, $\cdots$, t_k, $\cdots$, t_q) where $0 \le t_p \le 2$ ($p \ne k$) and $1 \le t_k \le 2$.

TABLE 1 Triangle-component ordering and possible one-to-one recurrence relationship(s) implemented in the tIntegral library (version 1.0.0)[20]

Component ordering	Contiguous components in memory	One-to-one recurrence relationship(s)
Descending *XY*-major order	Components along the $\vec{YZ}$ edge	$\triangle^{x}\vec{YZ}^{y}Z^{z}$
Descending *XZ*-major order	Components along the $\vec{ZY}$ edge	$\triangle^{x}\vec{ZY}^{z}Y^{y}$
Descending *Y X*-major order	Components along the $\vec{XZ}$ edge	$\triangle^{y}\vec{XZ}^{x}Z^{z}$
Descending *Y Z*-major order	Components along the $\vec{ZX}$ edge	$\triangle^{y}\vec{ZX}^{z}X^{x}$
Descending *ZX*-major order	Components along the $\vec{XY}$ edge	$\triangle^{z}\vec{XY}^{x}Y^{y}$
Descending *ZY*-major order	Components along the $\vec{YX}$ edge	$\triangle^{z}\vec{YX}^{y}X^{x}$

The restriction on the order (t_1, $\cdots$, t_q) should not affect the use of our recurrence-relation compiler, because most one- and two-electron integrals needed in computational-chemistry calculations can be evaluated with recurrence relations of order ≤2.[13,16] Furthermore, such a restriction can be simply extended as will be shown in the following discussion of loops of different indices.

The input of the `RecurCompiler` is recurrence relation(s), or more exactly the right hand side that is given in the form of (25). We have implemented an abstract symbolic class `RecurSymbol` and a few derived classes in the tIntegral library[20] as shown in Table 2. These derived classes can be used to (rapidly) construct the right hand side of a recurrence relation (25).

3.2.3 | Loops of different indices

Now let us look into the most challenging part of the `RecurCompiler`—the generation of loops over *XYZ* components of different indices $\boldsymbol{i}_1$, $\cdots$, $\boldsymbol{i}_k$, $\cdots$, $\boldsymbol{i}_q$ with $\boldsymbol{i}_k$ the output index, and the order (t_1, $\cdots$, t_k, $\cdots$, t_q). As discussed in the triangle-based recurrence relations, the generation of such loops over *XYZ* components are mostly decided by the chosen triangle-component ordering and the one-to-one recurrence relationship. The order (t_1, $\cdots$, t_k, $\cdots$, t_q) can also affect how the loops over *XYZ* components will be carried out.

Apparently, the generated loops over *XYZ* components should ensure:

1. Each component on the left hand side of a recurrence relation will be visited once so that its value can be computed from the recurrence relaiton;
2. Only contributing components on the right hand side will be visited, which will be used to compute the corresponding LHS component along either *x*, *y* or *z* direction;

TABLE 2 Derived classes of `RecurSymbol` implemented in the tIntegral library (version 1.0.0)[20]

Derived classes	Description
`RecurTerm`	A RHS term $\left[\boldsymbol{i}_1 + \sum_{\zeta_1} \boldsymbol{e}^{\pm}_{\zeta_1}, \cdots, \boldsymbol{i}_k - \sum_{\zeta_k} \boldsymbol{e}_{\zeta_k}, \cdots, \boldsymbol{i}_q + \sum_{\zeta_q} \boldsymbol{e}^{\pm}_{\zeta_q}\right]$ of (25).
`RecurNumber`	A constant.
`RecurScalarVar`	A scalar variable.
`RecurCartesianVar`	A Cartesian variable.
`RecurScalarVec`	A vector of scalars.
`RecurCartesianVec`	A vector of Cartesian variables.
`RecurIdxOrder`	Order of an index.
`RecurAddition`	An addition.
`RecurSubtraction`	A subtraction.
`RecurMultiplication`	A multiplication.
`RecurDivision`	A division.
`RecurParentheses`	An expression in parentheses.

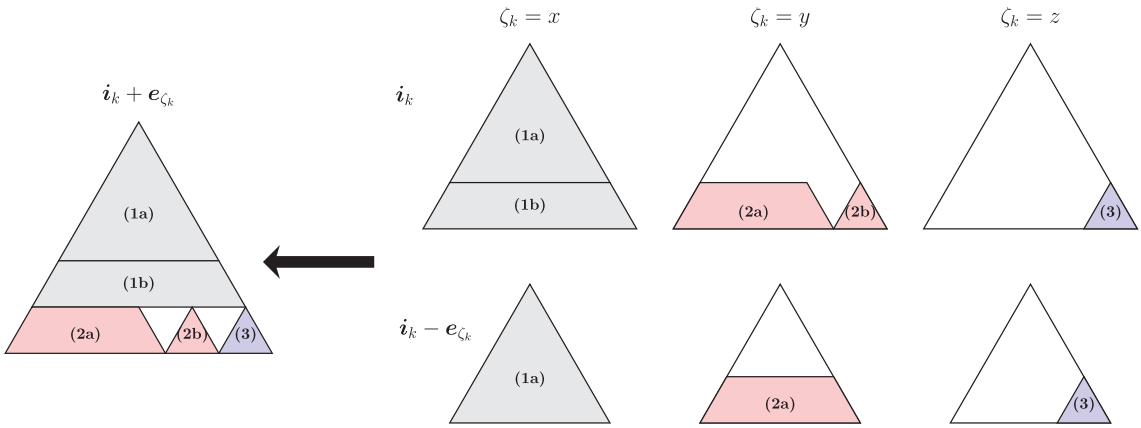

FIGURE 6 Loops over *XYZ* components of the output index i_k for the evaluation of the left hand side ($i_k + e_{\zeta_k}$) of a recurrence relaiton from contributing components on the right hand side (i_k and $i_k - e_{\zeta_k}$) with the descending *XY*-major order and the $\triangle^x\overrightarrow{YZ}^y Z^z$ recurrence relationship being chosen. The loops are first performed for parts (1a) and (1b) along *x* direction, then for (2a) and (2b) along *y* direction, and finally for (3) along *z* direction [Color figure can be viewed at wileyonlinelibrary.com]

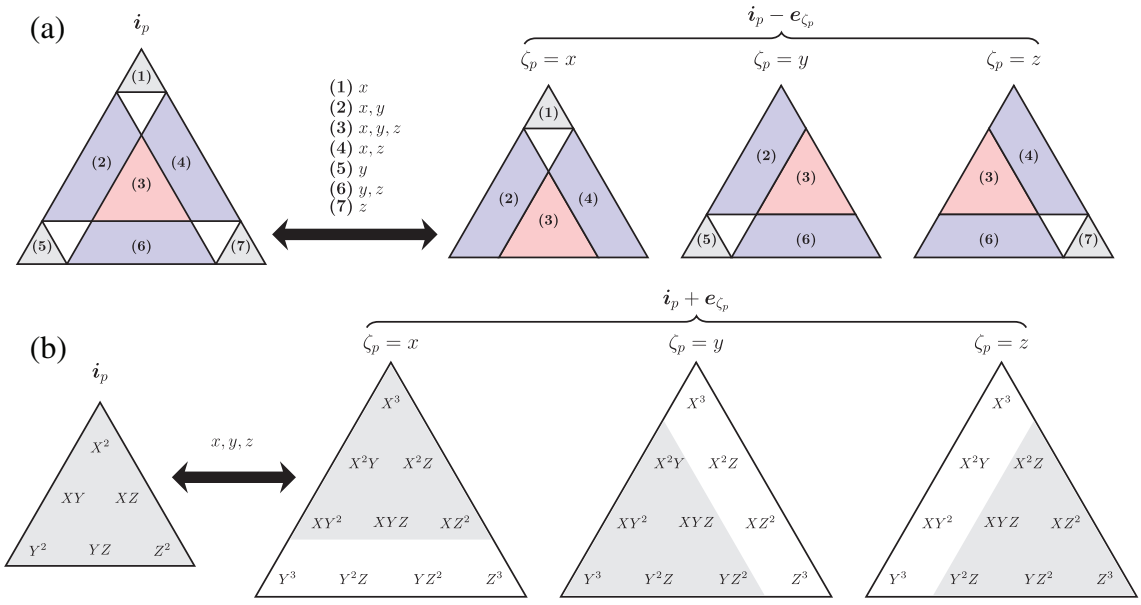

FIGURE 7 Contributing RHS components of inner and outer indices for the evaluation of LHS components of the triangle i_p, with the increment (a) $e^{\pm}_{\zeta_p} = -e_{\zeta_p}$ and (b) $e^{\pm}_{\zeta_p} = e_{\zeta_p}$, and with the descending *XY*-major order and the $\triangle^x\overrightarrow{YZ}^y Z^z$ recurrence relationship being chosen. In (a), an LHS component in one of the 7 parts may have a contributing RHS component along some direction(s) as given on top of the black arrow. In (b), all LHS components have their contributing RHS components along *x*, *y* and *z* directions as marked in gray color [Color figure can be viewed at wileyonlinelibrary.com]

3. Noncontributing RHS components will be skipped in an appropriate manner.

Let us first consider the output index i_k. Take the descending *XY*-major order and the $\triangle^x\overrightarrow{YZ}^y Z^z$ recurrence relationship for example, the loops over *XYZ* components of the output index i_k can be performed as described in Figure 6, where we have considered the cases of the order $t_k = 1$ and $t_k = 2$. In the former, RHS terms of a recurrence relation can only take the form $[\cdots, i_k, \cdots]$, while in the latter we need to consider both the form $[\cdots, i_k, \cdots]$ and $[\cdots, i_k - e_{\zeta_k}, \cdots]$.

For other combinations of the triangle-component ordering and the one-to-one recurrence relationship in Table 1, the generation of loops over *XYZ* components of the output index can be performed by following similar procedures to that of Figure 6. The restriction on the order of t_k ($1 \le t_k \le 2$) can also be removed by developing slightly different procedures for the cases of $t_k > 2$.

The generation of loops over *XYZ* components of inner and outer indices requires a different consideration. During the loops, one needs to figure out:

- For an inner index, which of its *XYZ* component(s) on the right hand side will contribute to the recurrence relation along a direction *x*, *y* or *z* that was set during the loops of the output index; and

- For an outer index, along which direction(s) *x*, *y* and/or *z*, one of its *XYZ* components on the right hand side will contribute to the recurrence relation.

Actually, the above slightly different statements for inner and outer indices require the same and important information for the generation of their loops over *XYZ* components—contributing RHS components along direction(s) *x*, *y* and/or *z*. For any inner or outer index $\boldsymbol{i}_p$, the contributing RHS components of the increment $\sum_{\zeta_p}\left|\boldsymbol{e}^{\pm}_{\zeta_p}\right| = 0$ are obvious—each LHS component $X^l_p Y^m_p Z^n_p$ $(l+m+n = |\boldsymbol{i}_p|)$ has the contributing RHS component $X^l_p Y^m_p Z^n_p$ regardless of the direction.

In Figure 7, we present contributing RHS components of inner and outer indices with the descending *XY*-major order and the $\triangle^x\overrightarrow{YZ}^y Z^z$ recurrence relationship for the increment (a) $\boldsymbol{e}^{\pm}_{\zeta_p} = -\boldsymbol{e}_{\zeta_p}$ and (b) $\boldsymbol{e}^{\pm}_{\zeta_p} = \boldsymbol{e}_{\zeta_p}$. The loops over *XYZ* components of inner and outer indices can therefore be performed by looping the LHS components and determined contributing RHS components from given direction(s), which also holds for any other increment $\sum_{\zeta_p}\left|\boldsymbol{e}^{\pm}_{\zeta_p}\right| \geq 2$.

We also note that there are noncontributing RHS components for the increment $\boldsymbol{e}^{\pm}_{\zeta_p} = \boldsymbol{e}_{\zeta_p}$ that need to be skipped during loops. For example, RHS components of the last row—from Y^3 to Z^3 as shown in Figure 7(b)—will not contribute to the recurrence relation along *x* direction and should be skipped.

A more general form of contributing and noncontributing RHS components is given in Figure 8 for an increment $\sum_{\zeta_p}\boldsymbol{e}^{\pm}_{\zeta_p} \equiv \boldsymbol{\tau}_p$. Take the increment $\boldsymbol{e}^{\pm}_{\zeta_p} = \boldsymbol{e}_{\zeta_p}$ for example, which gives $\tau_{px} = 1$ and $\tau_{py} = \tau_{pz} = 0$ along *x* direction, so that $|\boldsymbol{i}_p| + 2$ noncontributing components should be skipped after loops.

After converting the bottom-up procedure to different loops, the left and the only step that one needs to manually program is the evaluation of integrals $[\boldsymbol{0}\cdots\boldsymbol{0}]^{\text{HGTO}}_{\text{O}}$, which is usually trivial compared with

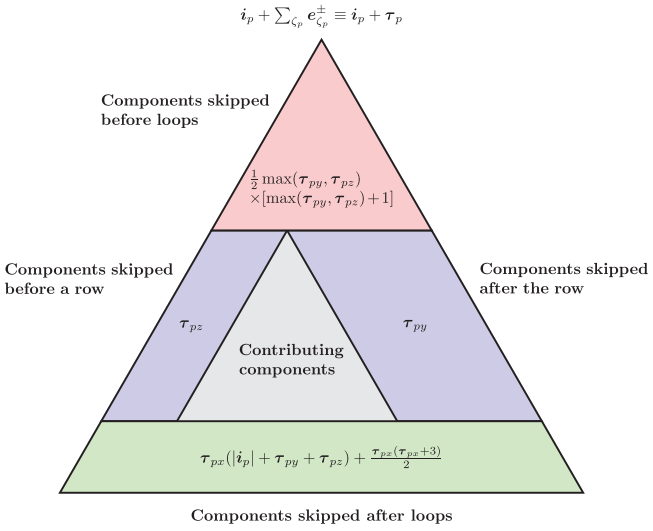

FIGURE 8 Contributing and noncontributing RHS components of inner and outer indices for the evaluation of LHS components with an increment $\sum_{\zeta_p}\boldsymbol{e}^{\pm}_{\zeta_p} \equiv \boldsymbol{\tau}_p$, and with the descending *XY*-major order and the $\triangle^x\overrightarrow{YZ}^y Z^z$ recurrence relationship being chosen [Color figure can be viewed at wileyonlinelibrary.com]

the implementation of the aforementioned different loops. Therefore, our developed automatic programming approach can work for almost all different molecular integrals in computational-chemistry calculations. Furthermore, by considering the recently developed just-in-time compilation technique[21] in computer science, users can even introduce new electron operators and evaluate their integrals during runtime. This will become quite useful for developers to quickly test their new idea in computational chemistry.

4 | DESIGN AND IMPLEMENTATION

The previous section has presented our two key steps for integral evaluation from an algorithmic view, in which the proposed scheme, data structures and algorithms can in principle be served to guide the practical implementation—the development of the tIntegral library.[20]

More exactly, what we have presented so far can fall into either software requirements (which functionalities of the tIntegral library we expect), or software construction (coding, data structures and algorithms) in the software engineering discipline.

One important software development process between the software requirements and the software construction is software design. A considered design can help one develop modularized, reusable, maintainable and extensible software. We have therefore followed standards of the software design in the development of the tIntegral library. In particular, we have applied well-known design patterns[22] to solve design problems we have encountered, and we have also employed unified modeling language (UML) to describe and to help us understand how our chosen design works both structurally and behaviorally.

We will in the next two subsections present our software design for integral computation and integral code generation using the tIntegral library. The object-oriented programming has been chosen as our programming paradigm and C++ programming language for the implementation.

4.1 | Integral computation

In Figure 9, we collect (important) classes of the tIntegral library (version 1.0.0) for one-electron integral computation as an example. The overall structure can be divided into (a) one-electron operators, (b) basis functions, (c) integration classes, and (d) a set of low-level classes for the execution of the top-down and bottom-up procedures—including the classes `RecurArray`, `RecurNode` and `RecurBuffer`.

The one-electron operators are derived from the base class `OneElecOperator` that is defined in the other library tSymbolic (https://gitlab.com/tglue-mathematics/tsymbolic). The reason of introducing the tSymbolic library is that it can take care of different symbolic operations, in particular the symbolic differentiation with respect to different (external) perturbations that is required for our developed open-ended response theory library OpenRSP (https://github.com/openrsp/openrsp). As such, one can directly send different (one-)electron operators to the tSymbolic and OpenRSP libraries,

and a seamless integration of symbolic operations and numerical evaluation can be expected for the response theory calculations.

Similarly, we define classes `GaussianFunction` (for primitive Hermite Gaussian functions) and `ContractedGTO` (for contracted real solid-harmonic GTOs or contracted Cartesian GTOs) derived from the base class `Symbol` so that they can also be used for symbolic operations in the response theory calculations.

For ordinary users, it is advisable to use the template class `OneElecGTOIntegration` for computing integrals of different one-electron operators with either contracted real solid-harmonic GTOs or contracted Cartesian GTOs. The template parameter `RealType` will be specified during compile time that determines the type of floating point numbers.

The class `OneElecGTOIntegration` actually works as the skeleton of integral computations. As illustrated in Figure 10, it will construct appropriate concrete integration classes during runtime according to the types of one-electron operator and basis functions on bra and ket centers. When the member method `integrate` is called, the class `OneElecGTOIntegration` will invoke these concrete integration classes one by one to perform the top-down and the bottom-up procedures.

Here, we follow the tradition of object-oriented programming by first introducing an abstract integration class `RecurIntegration`, and define all concrete integration classes as its derived class. The class `RecurIntegration` specifies the common interface that its derived classes should implemnet for the top-down and the bottom-up procedures. As illustrated in the deduction of Equation (10) and its following recurrence relations according to the operator $\hat{O}(\{\boldsymbol{r}_{C_\alpha}\})$, the integration classes for one-electron operators fall into two categories:

1. Integration of different one electron operators with primitive Hermite Gaussian functions. For instance, the classes `CartMultMomentHGTOIntegration` and `NucAttractPotentialHGTOIntegration` respectively take care of the integration of Cartesian multipole moment operator and nuclear attraction potential operator.
2. Transformation of integrals between primitive Hermite Gaussian functions and contracted Cartesian GTOs (by the class `ContractedCGTOIntegration`) or contracted real solid-harmonic GTOs (by the class `ContractedSGTOIntegration`).

Except for the class `ContractedSGTOIntegration`, all other concrete integration classes are automatically generated from the general recurrence-relation compiler `RecurCompiler`, whose design and implementation will be presented in the next subsection.

One may notice that the construction of an appropriate concrete integration requires the knowledge of either the one-electron operator class or the basis function class. A possible solution is to resort to

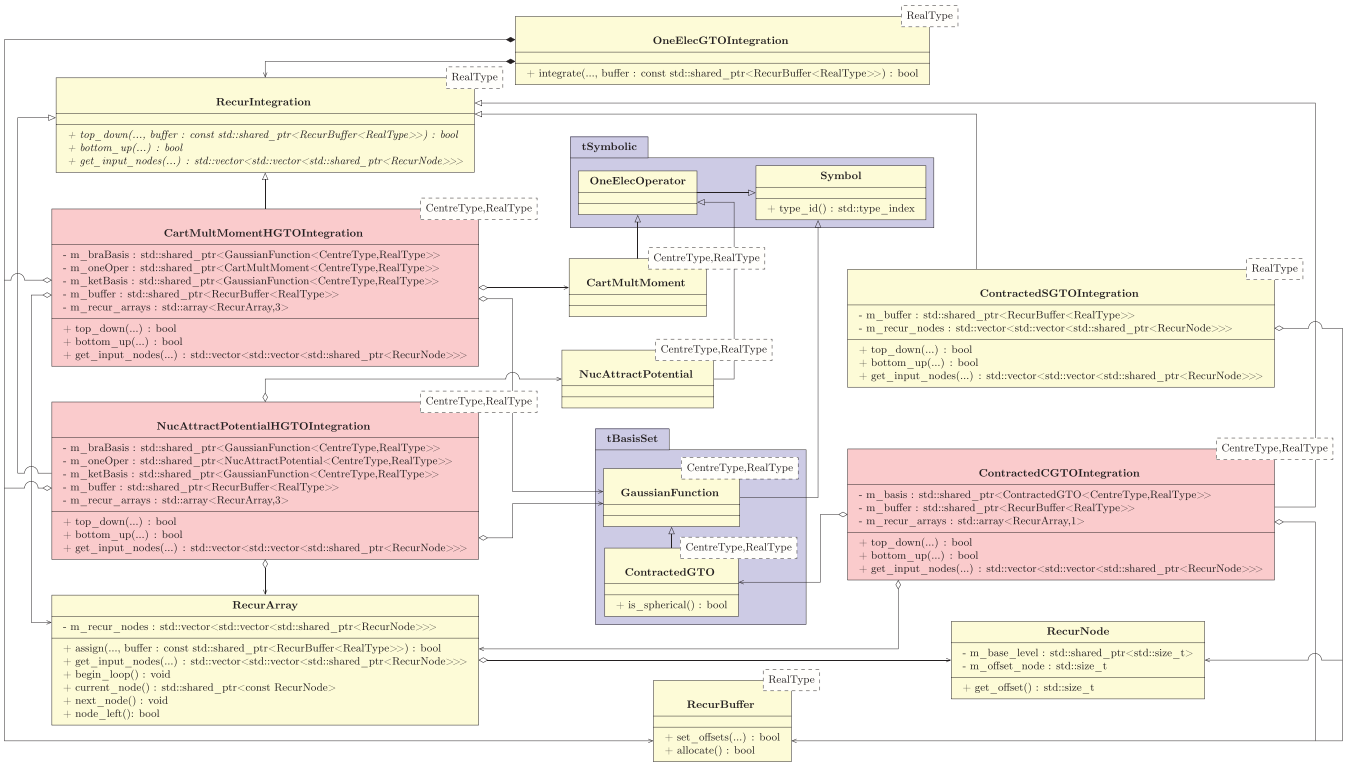

FIGURE 9 UML class diagram of the tIntegral library (version 1.0.0)[20] for one-electron integral computation with Gaussian type orbitals. Classes of basis functions and the abstract one-electron operator class `OneElecOperator` are respectively from our other developed libraries tBasisSet (https://gitlab.com/tglue-chemistry/tbasis-set) and tSymbolic (https://gitlab.com/tglue-mathematics/tsymbolic). Integration classes marked in red color are automatically generated from the general recurrence-relation compiler `RecurCompiler` (see Figure 11) [Color figure can be viewed at wileyonlinelibrary.com]

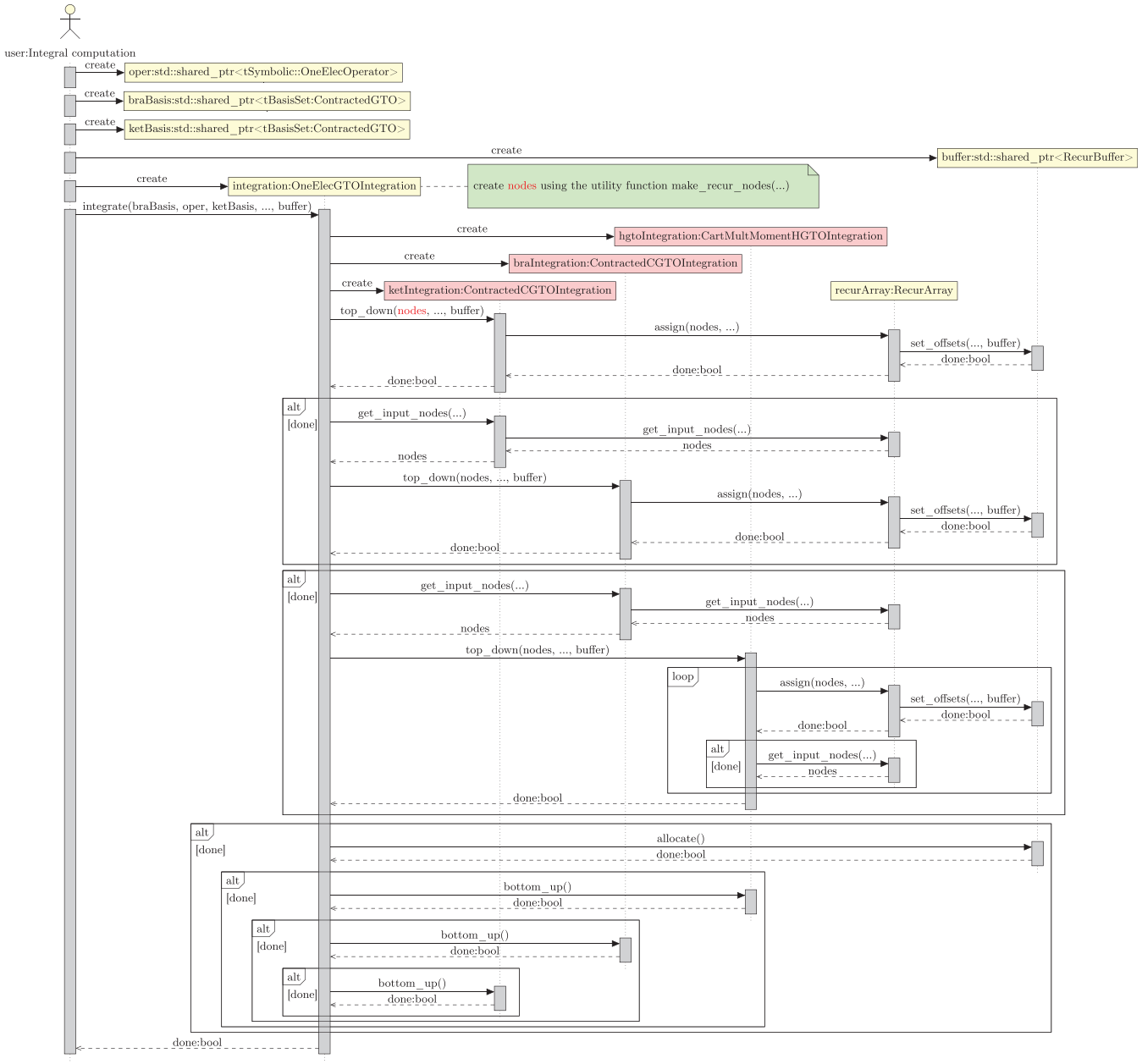

FIGURE 10 UML sequence diagram of the tIntegral library (version 1.0.0)[20] for computing integrals of Cartesian multipole moment operator with contracted Cartesian GTOs [Color figure can be viewed at wileyonlinelibrary.com]

the so-called visitor pattern.[22] However, it is not trivial to introduce any new (one-)electron operator or basis function class using the visitor pattern.

For the time being, we employ the member function `type_id` of the base class `Symbol` (see Figure 9) to create the correct concrete integration in the class `OneElecGTOIntegration` during runtime. This function returns the runtime type informaiton (RTTI) of an object (a given one-electron operator or basis function).

Nevertheless, the use of RTTI is not a reasonable choice within the framework of object-oriented programming, and the code structure of the class `OneElecGTOIntegration` is also a bit "brutal" that has many conditional statements to check each RTTI. We have considered the other probably better solution—pattern matching[23] for the implementation of the class `OneElecGTOIntegration`, which may become available in the next release of the tIntegral library.

Last but not least, we would like to point out that one can also directly use, for instance the derived class `CartMultMomentHGTOIntegration` or `NucAttractPotentialHGTOIntegration` to compute integrals with primitive Hermite Gaussian functions. It can be useful for some computational chemistry programs that have their own routines for the transformation to contracted Cartesian GTOs or contracted real solid-harmonic GTOs.

4.2 | Integral code generation

Figure 11 shows the UML class diagram of integral code generation using the tIntegral library (version 1.0.0). The key class is the `RecurCompiler`, with which users can interact to generate different concrete integration classes derived from the base one `RecurIntegration`.

Following the discussion of the bottom-up procedure, the main task of the `RecurCompiler` is to analyze the RHS of a given recurrence relation (in the form of the class `RecurSymbol`), in particular to figure out the increment of each index in each RHS term. Afterwards, the loops of different indices can be generated by following a given combination of the triangle-component ordering and the one-to-one recurrence relationship in Table 1, together with the schemes presented in Figures 6–8. Finally, the code to compute each individual LHS component needs to be generated within the innermost loop by following the formula of the recurrence relation.

Based on the above task analysis, we can divide the general recurrence-relation compiler into the following (categories of) classes:

1. `RecurTraversal` is a simple class for recording which combination of the triangle-component ordering and the one-to-one recurrence relationship in Table 1 is chosen.
2. `RecurExpression` (marked in red color in Figure 11) contains detailed information of a recurrence relation that the `RecurCompiler` accepts, including the RHS and indices involved into the recurrence relation. It will also create a `RecurAnalyzer` object for the recurrence relation during runtime.

   The `RecurCompiler` takes a vector of `RecurExpression` objects as one input argument for the integral code generation. Currently, we manually prepare these `RecurExpression` objects for different one-electron operators and basis functions in a class `RecurRelation` (see Figure 12). However, our long-term objective is to "teach" the class `RecurRelation` to automatically generate the `RecurExpression` objects for given electron operators

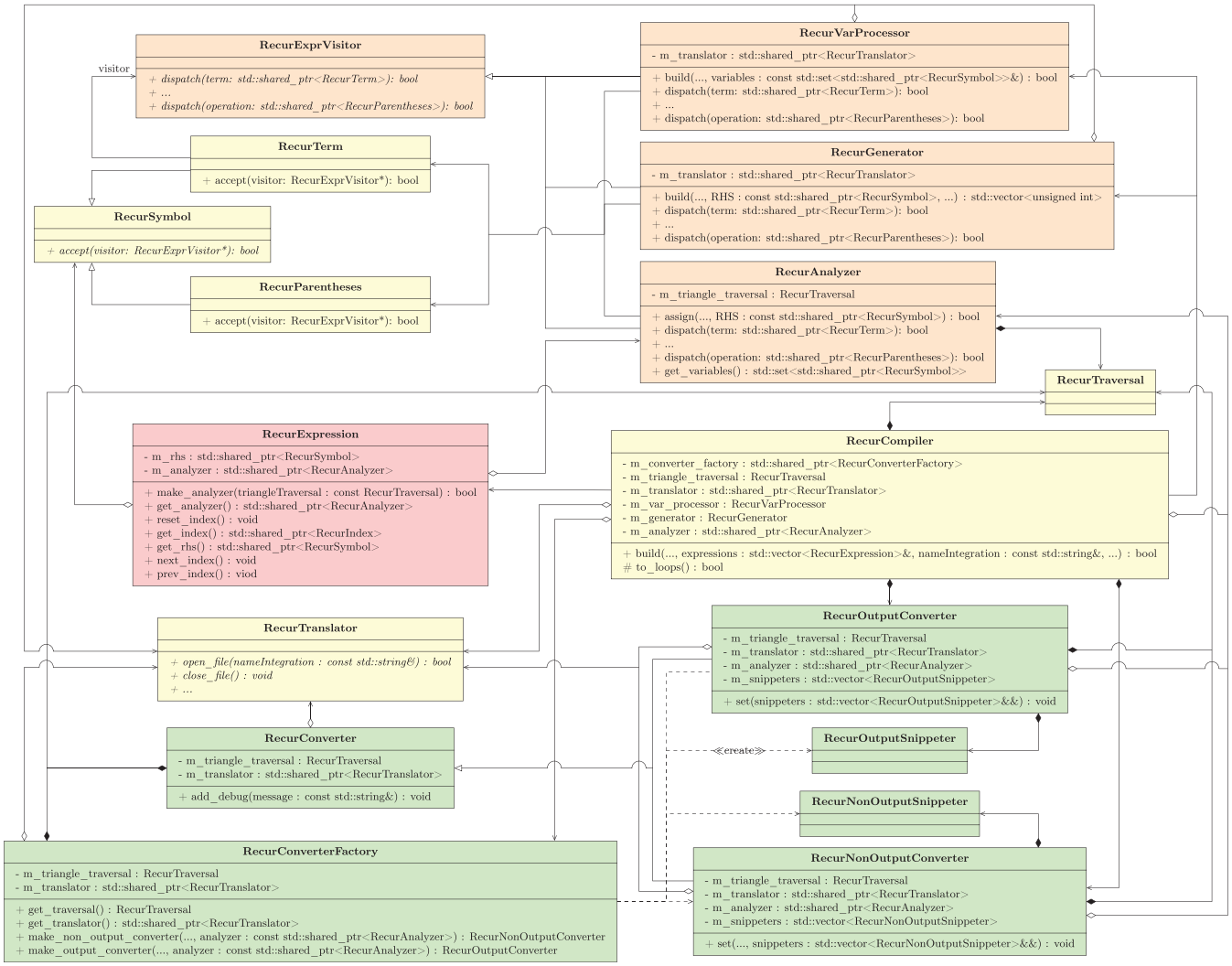

FIGURE 11 UML class diagram of the tIntegral library (version 1.0.0)[20] for integral code generation. See main text for detailed explanation of classes in different colors and design patterns used [Color figure can be viewed at wileyonlinelibrary.com]

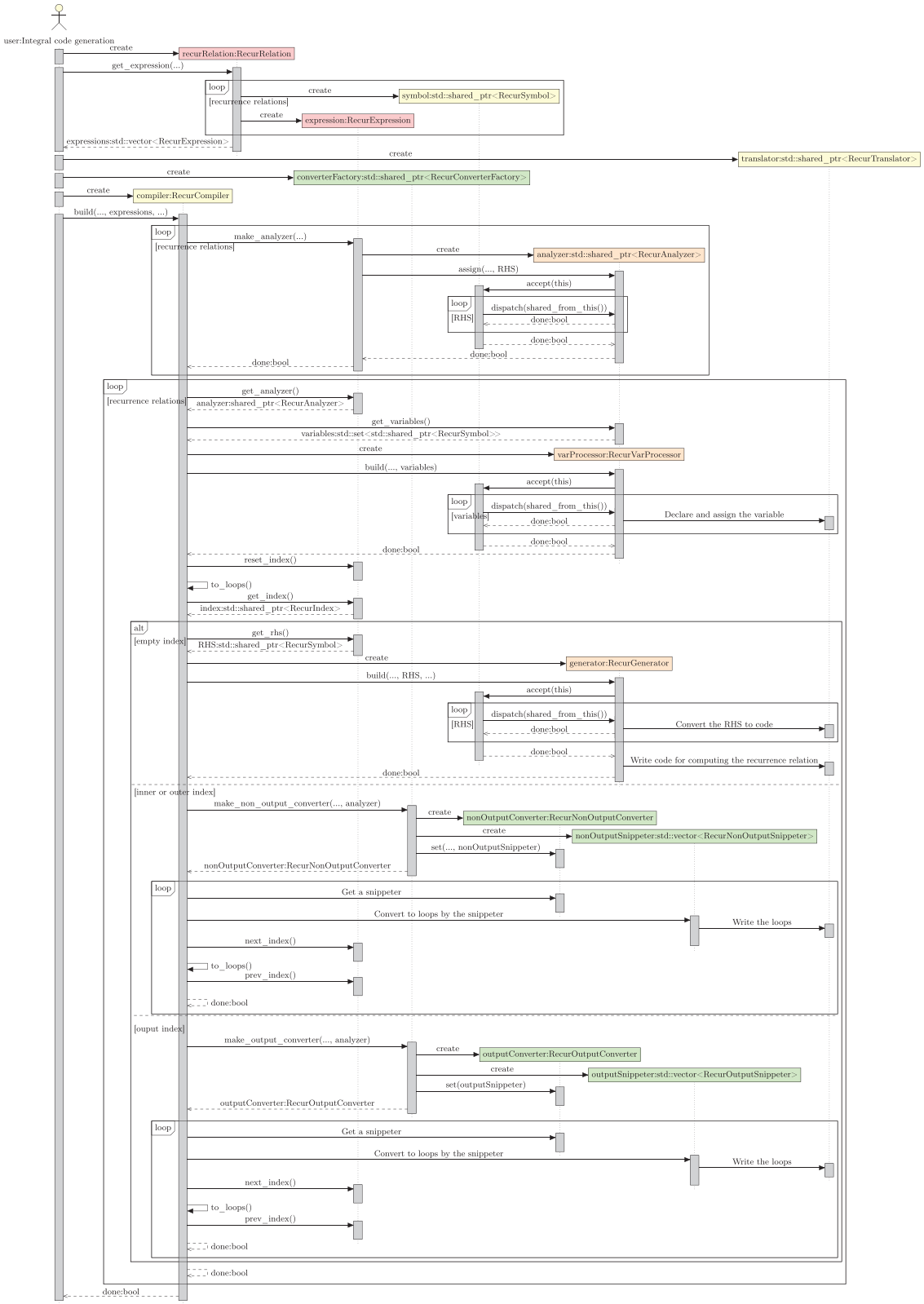

FIGURE 12 UML sequence diagram of the tIntegral library (version 1.0.0)[20] for integral code generation. See main text for detailed explanation of the use of the general recurrence-relation compiler `RecurCompiler` [Color figure can be viewed at wileyonlinelibrary.com]

and basis functions so that the manual coding work can be further reduced.

3. The classes `RecurAnalyzer`, `RecurVarProcessor` and `RecurGenerator` (marked in orange color in Figure 11) are derived from the base class `RecurExprVisitor`. They are respectively responsible for the analysis of the RHS of a recurrence relation, for the declaration and assigment of different variables of a recurrence relation, and for the generation of the code to compute each individual LHS component within the innermost loop.
   We have used the visitor pattern[22] for the above three classes to process different `RecurSymbol` derived classes that are used to construct the (RHS) of a recurrence relation. Within the visitor pattern, each `RecurSymbol` derived class needs to implement a dispatching operation `accept` that dispatches a request to the base class `RecurExprVisitor` as shown in Figure 11.
   Meanwhile, each class derived from the `RecurExprVisitor` needs to implement several `dispatch` operations to make sure all different `RecurSymbol` derived classes can be visited, which is a double dispatch approach.[22]
4. `RecurOutputConverter` and `RecurNonOutputConverter` (marked in green color in Figure 11) respectively convert a recurrence relation to loops for the output index and nonoutput (inner and outer) indices. Both of them are derived from the class `RecurConverter`, which takes care of a few common tasks in the loop generation, for example, adding debug information.
   Because the loops of each index are converted in a few steps (see Figures 6 and 7), we further introduce two other classes `RecurOutputSnippeter` and `RecurNonOutputSnippeter`, which generate a snippet for a single step of the loop converting for the output and nonoutput indices respectively.
   Furthermore, we have employed the abstract factory pattern[22] for creating `RecurOutputConverter` and `RecurNonOutputConverter` objects during runtime. As illustrated in Figure 11, the `RecurCompiler` refers to the class `RecurConverterFactory`'s member functions `make_output_converter` and `make_non_output_converter` for creating the "converter" objects, which also construct corresponding vectors of `RecurOutputSnippeter` and `RecurNonOutputSnippeter` objects for the converter objects.
5. `RecurTranslator` is a base class that any other class can and should use to generate the final source codes, and for the time being we have a derived class `RecurCppTranslator` that can be used to generate source codes in C++ programming language.It is straightforward to implement new derived classes for the generation of source codes in other programming languages like Fortran and GPU computing language.

A typical procedure for the integral code generation using the tIntegral library (version 1.0.0) is illustrated in Figure 12, where the key operation for generating the integral code is the `RecurCompiler`'s member function `to_loops`. This function will call itself in a recursive manner and move to the next index to be processed at the same time. When all indices are processed (i.e., an empty index got from the class `RecurExpression`), the `RecurGenerator` object will execute to generate the source code for the computation of each individual LHS component within the innermost loop.

The general recurrence-relation compiler can be used to automatically generate different concrete integration classes as introduced in the previous subsection. The only left manual work is to write a member function to compute the integrals $[\mathbf{0}\cdots\mathbf{0}]_{\hat{O}}^{\mathrm{HGTO}}$ with the zeroth order Hermite Gaussian functions, which is much less and simpler than writing such recurrence relation codes manually.

5 | EXAMPLES AND DISCUSSIONS

To test the performance of the tIntegral library, we have chosen our previous implementation—Gen1Int library (version 0.2.1)[24] for reference. In Table 3, we present the CPU time used by these libraries with different one-electron operators and Hermite Gaussian functions.

Different from the general recurrence-relation generation scheme proposed here, all recurrence relation codes were manually converted

TABLE 3 CPU time (millisecond) used by Gen1Int (version 0.2.1)[24] and tIntegral (version 1.0.0)[20] libraries with different one-electron operators, and Hermite Gaussian functions $(2a_\kappa)^{-|\boldsymbol{l}_\kappa|}\partial_{\mathbf{R}_\kappa}^{\boldsymbol{l}_\kappa}\exp(-a_\kappa r_\kappa^2)$ on bra and $(2b_\lambda)^{-|\boldsymbol{l}_\lambda|}\partial_{\mathbf{R}_\lambda}^{\boldsymbol{l}_\lambda}\exp(-b_\lambda r_\lambda^2)$ on ket centers

	$\langle \boldsymbol{l}_\kappa \vert \boldsymbol{l}_\lambda\rangle$	$\langle \boldsymbol{s}\vert\boldsymbol{s}\rangle$	$\langle \boldsymbol{p}\vert\boldsymbol{p}\rangle$	$\langle \boldsymbol{d}\vert\boldsymbol{d}\rangle$	$\langle \boldsymbol{f}\vert\boldsymbol{f}\rangle$	$\langle \boldsymbol{g}\vert\boldsymbol{g}\rangle$	$\langle \boldsymbol{h}\vert\boldsymbol{h}\rangle$	$\langle \boldsymbol{i}\vert\boldsymbol{i}\rangle$	$\langle \boldsymbol{j}\vert\boldsymbol{j}\rangle$
$\hat{O}=1$	Gen1Int	0.0001	0.0003	0.0007	0.0013	0.0022	0.0040	0.0067	0.0113
	tIntegral	0.002	0.006	0.009	0.012	0.015	0.018	0.023	0.028
$\hat{O}=\boldsymbol{r}_M$	Gen1Int	0.0002	0.0004	0.0012	0.0028	0.0059	0.0111	0.0200	0.0337
	tIntegral	0.004	0.011	0.015	0.020	0.024	0.031	0.040	0.054
$\hat{O}=\boldsymbol{r}_M^2$	Gen1Int	0.0002	0.0016	0.0038	0.0080	0.0166	0.0309	0.0543	0.1045
	tIntegral	0.005	0.015	0.023	0.031	0.042	0.057	0.080	0.114
$\hat{O}=\boldsymbol{r}_M^3$	Gen1Int	0.0004	0.0031	0.0072	0.0175	0.0436	0.0726	0.1287	0.2142
	tIntegral	0.006	0.018	0.031	0.048	0.073	0.109	0.161	0.237
	Top-down	0.005	0.012	0.017	0.021	0.028	0.031	0.034	0.037

Note: All computations were performed on the Linux cluster Stallo at UiT The Arctic University of Norway, which has 2.60 GHz Intel Xeon E5 2,670 processor. The Gen1Int library was built with GNU Fortran (GCC) 7.3.0, and the tIntegral library was built with GCC 7.3.0, and both at level `-O3`. The CPU time is the average of 10,000 times.

into loops in the Gen1Int library. Instead of explicitly constructing jagged arrays for different recurrence relation terms, the loops of different indices in the Gen1Int library were prepared by specifying their maximum and minimum orders. As such, there may be extra and unnecessary integrals computed using the Gen1Int library in some cases.

Another pitfall of the Gen1Int library is that it only accepts one pair of exponents on bra and ket centers at a time, different from the tIntegral library that can take multiple exponents on both centers. Nevertheless, we only use one pair of exponents a_κ and b_λ for our comparison in Table 3.

It is not surprised that the computations using the tIntegral library take more CPU time than those of the Gen1Int library, in particular in the cases of lower order Hermite Gaussian functions (till *f* shell) where the CPU time is dominated by the top-down procedure as illustrated for the operator $\hat{O} = \boldsymbol{r}_M^3$ in Table 3.

However, we need to point out that the integrals with *s* and *p* shells are directly calculated in the Gen1Int library without any loop of indices, whereas the top-down procedure is always performed and the jagged arrays are always constructed in the current implementation of the tIntegral library. It can be improved by adding conditional statements in the concrete integration classes so that Hermite Gaussian functions with lower orders (such as *s*, *p* and *d* shells) will be computed directly.

We would also like to argue that the current implementation of the top-down procedure can be further optimized in the tIntegral library. For example, we currently compare orders of all indices when trying to find matching RHS nodes, which can be performed only for indices involved in the recurrence relation.

Furthermore, it is worth mentioning that the comparison in Table 3 is carried out with only one pair of exponents (a_κ and b_λ). The percentage of the CPU time used for the top-down procedure will become less if there are multipole exponents on bra and/or ket centers, which is usually the case in computational chemistry calculations.

One should also observe that, for higher order Hermite Gaussian functions, the CPU time used by the tIntegral library becomes more and more comparable with that of the Gen1Int library. Moreover, by only considering the bottom-up procedure, the tIntegral library has used less CPU time than that of the Gen1Int library in the cases of *i* and *j* shells as revealed in Table 3. It can be explained from the aforementioned fact that the loops of different indices in the Gen1Int library are carried out from a given minimum order to a maximum one, so that there may be extra and unnecessary integrals computed and more CPU time can be taken.

We have also compared the performance of the tIntegral library to the Libint library (version 2.1.0).[25] The CPU time is given in Table 4 for different one-electron operators and Cartesian Gaussian functions. We have chosen the horizontal recurrence relation for the order of the Cartesian multipole moment operator and the Obara-Saika recurrence relations for the angular momenta of Cartesian Gaussian functions[15] in the tIntegral library.

The Libint library generates integral codes for a given maximum angular momentum and (intermediate) integrals are explicitly addressed without any loop. As such, the codes are highly efficient as revealed in Table 4. But the pitfall is that the Libint library can not handle arbitrary orders of (geometrical) derivatives and angular momenta without regenerating codes. In contrast, our proposed scheme in the current paper and the tIntegral library can compute different integrals as well as their (geometrical) derivatives to arbitrary order, which is vital for (high-order) response theory calculations.

We have also measured the CPU time with more than one Gaussian function on bra and ket centers. In Table 4, we present the CPU time used for computing integrals of $\hat{O} = 1$ and $\hat{O} = \boldsymbol{r}_M^3$ with 5 Cartesian Gaussian functions on bra and ket centers. Interestingly, the performance of the tIntegral library becomes comparable with that of the Libint library, and even better for $\hat{O} = \boldsymbol{r}_M^3$ and Gaussian functions with

TABLE 4 CPU time (millisecond) used by Libint (version 2.1.0)[25] and tIntegral (version 1.0.0)[20] libraries with different one-electron operators, and Cartesian Gaussian functions $\boldsymbol{r}_\kappa^{l_\kappa}\exp(-a_\kappa r_\kappa^2)$ on bra and $\boldsymbol{r}_\lambda^{l_\lambda}\exp(-b_\lambda r_\lambda^2)$ on ket centers

	$\langle \boldsymbol{l}_\kappa \vert \boldsymbol{l}_\lambda \rangle$	$\langle \boldsymbol{s} \vert \boldsymbol{s} \rangle$	$\langle \boldsymbol{p} \vert \boldsymbol{p} \rangle$	$\langle \boldsymbol{d} \vert \boldsymbol{d} \rangle$	$\langle \boldsymbol{f} \vert \boldsymbol{f} \rangle$	$\langle \boldsymbol{g} \vert \boldsymbol{g} \rangle$	$\langle \boldsymbol{h} \vert \boldsymbol{h} \rangle$	$\langle \boldsymbol{i} \vert \boldsymbol{i} \rangle$
$\hat{O} = 1$	Libint	0.0001	0.0001	0.0001	0.0003	0.0004	0.0007	0.0015
	tIntegral	0.003	0.007	0.012	0.017	0.025	0.039	0.063
$\hat{O} = 1$ (a)	Libint	0.0014	0.0017	0.0024	0.0042	0.0086	0.0159	0.0273
	tIntegral	0.005	0.012	0.019	0.038	0.085	0.187	0.378
$\hat{O} = \boldsymbol{r}_M$	Libint	0.0001	0.0001	0.0003	0.0006	0.0022	0.0048	0.0084
	tIntegral	0.007	0.011	0.018	0.025	0.038	0.058	0.093
$\hat{O} = \boldsymbol{r}_M^2$	Libint	0.0001	0.0002	0.0005	0.0031	0.0066	0.0139	0.0295
	tIntegral	0.010	0.018	0.025	0.036	0.055	0.087	0.161
$\hat{O} = \boldsymbol{r}_M^3$	Libint	0.0001	0.0004	0.0020	0.0063	0.0160	0.0348	0.0623
	tIntegral	0.013	0.023	0.033	0.051	0.083	0.161	0.272
$\hat{O} = \boldsymbol{r}_M^3$ (a)	Libint	0.0021	0.0064	0.0416	0.1493	0.3514	0.8514	1.4442
	tIntegral	0.020	0.038	0.078	0.172	0.369	0.735	1.368

Note: All computations were performed on the Linux cluster Stallo at UiT The Arctic University of Norway, which has 2.60 GHz Intel Xeon E5 2,670 processor. Both the Libint and the tIntegral libraries were built with Intel(R) C++ Compiler 16.0.3 and at level `-O3`. The CPU time is the average of 10,000 times. (*a*) CPU time was measured with five Cartesian Gaussian functions on bra and ket centers.

angular momenta *h* and *i*. It may be due to the Libint library places the loops of Gaussian functions outside recurrence relations, while the tIntegral library puts these loops inside recurrence relations and at the deepest level–which is more efficient.

To briefly summarize, it is worthy of using the current version the tIntegral library in practical calculations, especially for the computations of different one-electron integrals. Except for further improvement of the library itself, users can consider to generate all possible jagged arrays in advance so that they can be (re)used during runtime. As such, the total CPU time used for the top-down procedure will become trivial.

6 | CONCLUSIONS

A general recurrence-relation generation scheme has been proposed in the current contribution, and its implementation in the recently developed tIntegral library[20] has also been discussed. In particular, the application of software design patterns[22] has been highlighted for developing a modularized, reusable, maintainable and extensible library.

This new scheme and the tIntegral library are able to program different molecular integrals automatically and thus significantly reduces the programming and maintaining effort for the integral evaluation in computational chemistry. Our chosen software designs have also made the tIntegral library be able to work with the just-in-time compilation technique. For instance, it can be straightforward to use the tIntegral library in the recently developed interactive C++ interpreter—Cling[21] (which is built on top of the LLVM compiler infrastructure[26]) so that one can generate source codes for any new electron operator and immediately evaluate its integrals during runtime.

Our current focus is on the further improvement and development of the currently proposed scheme and the tIntegral library. More explicitly, we have considered and begun to work on the following issues:

1. Evaluation of integrals using London atomic orbitals[27] and their derivatives with respect to the external electric and magnetic fields are important for molecular property calculations, which will require new recurrence relations and can be fitted into the current proposed scheme.
2. Our general recurrence-relation generation scheme can also be used for programming two-electron integrals, but the top-down procedure needs to be further optimized or discarded so that loops of indices will be performed for a given range of orders.
3. Although the `RecurCompiler` class has provided an automatic, generic and simple way to program various recurrence relations, one still needs to prepare the right hand side of a recurrence relation using the `RecurSymbol` derived classes, which usually requires tens of lines or a few hundred lines of coding work.
   To further reduce human's coding work, we can "teach" the class `RecurRelation` to automatically generate the right hand side of a recurrence relation based on the tSymbolic library (https://gitlab.com/tglue-mathematics/tsymbolic).

ACKNOWLEDGMENTS

This work was partially supported by the Research Council of Norway through its Centers of Excellence scheme, project number 262695. This work also received support from the Norwegian Supercomputer Program (NOTUR) through a grant of computer time (Grant No. NN4654K).

ORCID

Bin Gao https://orcid.org/0000-0003-1092-7785

REFERENCES

[1] S. Obara, A. Saika, *J. Chem. Phys.* **1985**, *84*, 3963.
[2] S. Obara, A. Saika, *J. Chem. Phys.* **1988**, *89*, 1540.
[3] L. E. McMurchie, E. R. Davidson, *J. Comp. Phys.* **1978**, *26*, 218.
[4] M. Dupuis, J. Rys, H. F. King, *J. Chem. Phys.* **1976**, *65*, 111.
[5] A. J. Thorvaldsen, K. Ruud, K. Kristensen, P. Jørgensen, S. Coriani, *J. Chem. Phys.* **2008**, *129*, 214108.
[6] C. W. Kern, M. Karplus, *J. Chem. Phys.* **1965**, *43*, 415.
[7] P. Chandra, R. J. Buenker, *J. Chem. Phys.* **1983**, *79*, 358.
[8] P. M. W. Gill, R. S. John, C. Z. Michael, *Advances in Quantum Chemistry*, Vol. 25, Academic Press, San Diego **1994**, p. 141.
[9] R. Ahlrichs, *Phys. Chem. Chem. Phys.* **2006**, *8*, 3072.
[10] P. Schwerdtfeger, H. Silberbach, *Phys. Rev. A* **1988**, *37*, 2834.
[11] P. Schwerdtfeger, H. Silberbach, *Phys. Rev. A* **1990**, *42*, 665.
[12] M. J. Smit, *Int. J. Quantum Chem.* **1999**, *73*, 403.
[13] B. Gao, A. J. Thorvaldsen, K. Ruud, *Int. J. Quantum Chem.* **2010**, *111*, 858.
[14] X. S. Raymond, *Elementary introduction to the theory of pseudodifferential operators*, CRC Press, Boca Raton, Florida **1991**.
[15] T. Helgaker, P. Jørgensen, J. Olsen, *Molecular Electronic-Structure Theory*, John Wiley & Sons Ltd, Chichester **2000**.
[16] S. Reine, E. Tellgren, T. Helgaker, *Phys. Chem. Chem. Phys.* **2007**, *9*, 4771.
[17] A. Youssef, *Proceedings of the 2nd European Parallel and Distributed Systems Conference*, IASTED/ACTA Press, Vienna, Austria **1998**, p. 325.
[18] N. Christofides, *Graph Theory: An Algorithmic Approach, Computer Science and Applied Mathematics*, Academic Press Inc, New York **1975**.
[19] A. Rák, G. Feldhoffer, G. Soós, T. Höltzl, B. Oroszi, and G. Cserey, WO Patent App. PCT/HU2013/000,051; **2014**.
[20] B. Gao, *tIntegral development version 1.0.0*, library for both integral computations and integral code generation for different electron operators in computational chemistry, and is released under the Mozilla Public License, version 2.0; **2020**. https://gitlab.com/tglue-chemistry/tintegral
[21] V. Vasilev, P. Canal, A. Naumann, P. Russo, *J. Phys. Conf. Ser.* **2012**, *396*, 052071.
[22] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, New York **1995**.
[23] Y. Solodkyy, G. Dos Reis, B. Stroustrup, *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, ACM, New York, NY **2013**, p. 97.
[24] B. Gao and A. J. Thorvaldsen, *Gen1Int version 0.2.1*, gen1Int is a library to evaluate the derivatives of one-electron integrals with respect to the geometry perturbation and external electric and magnetic fields at zero fields with contracted London atomic orbitals, and is released under the GNU Lesser General Public License; **2012**. https://gitlab.com/bingao/gen1int

[25] E. F. Valeev, *Libint version 2.1*., Libint - a library for the evaluation of molecular integrals of many-body operators over Gaussian functions; **2016**. https://github.com/evaleev/libint
[26] C. Lattner, V. Adve, *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, Santa Clara, CA **2004**.
[27] F. London, *J. Phys. Radium* **1937**, *8*, 397.

How to cite this article: Gao B. General recurrence-relation generation scheme for molecular integral evaluation. *J Comput Chem*. 2020;41:2722–2739. https://doi.org/10.1002/jcc.26425