Concise, Open-Ended Implementation of Rys Polynomial Evaluation of Two-Electron Integrals

Joseph D. Augspurger, David E. Bernholdt,* and Clifford E. Dykstra†

Department of Chemistry, University of Illinois, Urbana, Illinois 61801

Received 19 December 1989; accepted 22 March 1990

A quadrature-point-driven implementation of the standard Rys polynomial method for computing two-electron repulsion integrals of gaussian basis functions has been found to be both concise and open-ended with respect to the angular momentum of the gaussian functions (i.e., $s,p,d,f,g,\ldots$). These are important features in certain applications, such as molecular properties and property gradients.

INTRODUCTION

The evaluation of two-electron repulsion integrals,

$$(ij\,|\,kl) = \int\int \phi_i(1)\phi_j(1)\frac{1}{r_{12}}\phi_k(2)\phi_l(2)\,d\tau_1\,d\tau_2, \quad (1)$$

where $\phi_i(1) = N_i(x_1 - x_i)^{i_x}(y_1 - y_i)^{i_y}(z_1 - z_i)^{i_z} \cdot \exp(-\alpha_i|\mathbf{r}_1 - \mathbf{r}_i|^2)$, is the first major computational step in any *ab initio* quantum chemical calculation. A general formulation for computing these integrals[1] is

$$(ij\,|\,kl) = \sum_{i=0}^{\lambda} C_i F_i(X), \quad \text{where} \quad (2)$$

$$F_i(X) = \int_0^1 t^{2i}\exp(-Xt^2)\,dt, \quad \text{and} \quad (3)$$

$$X = \rho|\mathbf{P} - \mathbf{Q}|^2, \quad \text{where } \rho = \frac{AB}{A+B} \quad (4)$$

A is the exponent and $\mathbf{P}$ the center of the single Gaussian which results from combining the Gaussians of the i and j basis functions, and B and $\mathbf{Q}$ are the corresponding quantities for the k and l functions. λ is the sum of the powers of the Cartesian components of all the basis functions. The difficulty with this formulation is finding the coefficients C_i. That process involves nested summations over many indices, which become computationally impractical for high angular momentum functions, since the limits are related to the Cartesian powers or angular momentum of the Gaussian functions. King and Dupuis[2-4] introduced an alternative method, based on numerical quadrature using Rys polynomials, to evaluate the repulsion integrals. They make use of the roots (t_α) and weights (W_α) of the Nth-order Rys polynomial to cast the problem into the form of an N-point quadrature where N is any integer greater than $\lambda/2$, although in practice N is chosen to be the smallest integer greater than $\lambda/2$.

$$(ij\,|\,kl) = \int_0^1 P_\lambda(t)\exp(-Xt^2)\,dt = \sum_{\alpha=0}^{N} P_\lambda(t_\alpha)W_\alpha \quad (5)$$

The important improvement of their method is that the costly evaluation of the coefficients C_i is bypassed entirely through recursion relations they developed for evaluating $P_\lambda(t)$.

A powerful, formal feature of the Rys polynomial scheme is the ease of using it beyond standard s,p bases to s,p,d sets, to s,p,d,f sets, and so on. We have sought a particular implementation which exploits this feature to the fullest: a program with no limits on the angular momentum of the Gaussian basis functions ($s,p,d,f,g,h,\ldots$). This is valuable in situations where high-l functions are needed in the basis, for instance in the evaluation of polarizabilities and susceptibilities.[5] It is also valuable in the analytical evaluation of high-order energy gradients, where for instance, the second energy derivative of a wave function in an s,p,d,f basis implicitly requires integrals over h-functions ($l = 5$). Therefore, we sought a general processing of integrals of all l functions. This, it turns out, can be accomplished through a largely quadrature-point-driven loop structure, ensuring that the longest loops for $d,f,g,h,\ldots$ functions are the inner loops.

Obara and Saika[6] and Head-Gordon and Pople[7] have both reported new methods for evaluating repulsion integrals based on recursion relationships. These reports demonstrate a greater effi-

*Current address: Quantum Theory Project, University of Florida, Gainesville, Florida 32611.

†To whom all correspondence should be addressed.

Journal of Computational Chemistry, Vol. 11, No. 8, 972–977 (1990)

CCC 0192-8651/90/080972-06$04.00

ciency of evaluating integrals over s, p, and d functions compared to methods based on Rys quadrature. These methods eliminate redundancies in using certain intermediate quantities, but like other schemes based on handling shells of functions,[8-10] require storage of these quantities in memory. The number of integrals per shell increases as the fourth power of the number of functions per shell. Since the number of Cartesian functions goes as $(l + 1)(l + 2)/2$, or as $2l + 1$ if only unique l functions are kept, this intermediate storage requirement scales as l^4, and rapidly becomes quite large. In our implementation of the Rys quadrature approach based on handling individual integrals, the significant memory requirements arise from the roots and weights, and their storage grows roughly as l^2. Clearly, there is a trade-off in efficiency versus keeping the coding general for all values of l. But in addition to the generality or open-endedness of our scheme, it turns out to be very concise, making it subject to overall optimization, vectorization or adaptation to special hardware. This conciseness[11] means that even small, fixed memory computers like the Apple Macintosh can accommodate the program and evaluate integrals for large, high-l bases.

ROOTS AND WEIGHTS

The Rys quadrature evaluation of two-electron integrals requires roots of the Rys polynomials and the appropriate weight factors at the outset. We need the roots and weights of the Rys polynomials, which are even polynomials in t,

$$R_n(X,t) = \sum_{\alpha=0}^{n} d_{n,\alpha} t^{2\alpha} \tag{6}$$

(whose coefficients, d, depend parametrically on X), through order $2l + 1$, where l is the highest value of angular momentum of the functions in the basis, and for all values of X. (Denoting explicit dependence on X and t will be suppressed from this point for brevity.) Roots and weights must be found for all polynomials needed in a given problem. With our program, numerical values that give all these roots and weights are read from a file at the outset of any integrals calculation.

The roots and weights can be obtained and stored in a number of ways. Following King and Dupuis,[2] an asymptotic approximation is used for large X, and otherwise low order Chebyshev polynomials[12] are used for representing the roots and weights as a function of X. Our approach is to fit a root (weight) with a high order Chebyshev polynomial in X, the order being arbitrarily set to 50 to achieve essentially an exact fit. Then, the longest segment of X is found such that truncation of the Chebyshev polynomial to some low order, which we chose to be five, leaves the error strictly limited to a very small preset limit. The coefficients of the Chebyshev polynomials for each segment are then stored sequentially in a one-dimensional array. A separate pointer array maps values of X to the position in the coefficient array for the appropriate Chebyshev polynomial. More specifically, the mapping is accomplished by finding the least common multiple, X_c, of $L_1, L_2, L_3, \ldots L_m$, where the L_i's are the lengths of the fitted segments of X for a particular root (weight). Then the pointer array has 1 in the first L_1/X_c array locations, and then with n as the number of polynomial coefficients, it has $n + 1$ entered in the next L_2/X_c array locations, and so on for each fitted segment. This allows any order of Rys roots or weights to be stored sequentially in just two arrays, plus a short pointer array. Of course, the value of X_c for each root (weight) must be saved to determine the proper element of the mapping array for a given value of X.[13]

The coefficients of the $n + 1$ order Rys polynomial for a given value of X are found from the n and lower order polynomials using the recursion relationship (eq. (55) in [2])

$$\beta_{n+1} R_{n+1} = (t^2 - \beta'_n) R_n - \beta_n R_{n-1} \tag{7}$$

where $\beta_0 = 0$, and

$$\beta_{n+1} = \int_0^1 t^2 R_{n+1} R_n \exp(-Xt^2)\, dt$$

$$= \left(\int_0^1 t^4 R_n^2 \exp(-Xt^2)\, dt - \beta'^2_n - \beta_n^2 \right)^{1/2} \tag{8}$$

$$\beta'_n = \int_0^1 t^2 R_n^2 \exp(-Xt^2)\, dt$$

$$= \sum_{\alpha,\gamma=0}^{n} d_{n,\alpha} d_{n,\gamma} F_{\alpha+\gamma+1}(X) \tag{9}$$

The integral on the right hand side of eq. (8) may be evaluated by a sum analogous to that in eq. (9), except that the index on $F(X)$ must then be $\alpha + \gamma + 2$, not $\alpha + \gamma + 1$. For low order polynomials, evaluating the β coefficients by these summations is fast and accurate. However, the coefficients for high order Rys polynomials become large with alternating sign, and the situation arises where a number on the order of one results from the sum of large numbers of alternating sign. An alternative method is to perform the integrations over t in eqs. (8) and (9) numerically; that does not require explicitly multiplying the Rys polynomial coefficients. On a VAX 11/780, using double precision (word length of 8 bytes) provides about 15 decimals accurately and we found that the summations began to lose

accuracy for the Rys polynomials of order 10. Performing the integrations numerically instead is cumbersome, but of course the polynomial representation of the roots and weights, however achieved, is only done once. Again, it is a process that is external to actually evaluating the two-electron integrals.

COMPUTATIONAL APPROACH

The key task in evaluating the electron repulsion integrals is efficient evaluation of the polynomials in eq. (5) at the Rys polynomial roots. Rys, Dupuis, and King[4] presented relationships for evaluating the polynomials by writing them as (eq. (30) in reference 4).

$$P_\lambda(t_\alpha) = 2\sqrt{\rho/\pi}\, I_x(t_\alpha) I_y(t_\alpha) I_z(t_\alpha) \tag{9}$$

where the individual terms I_x, I_y, and I_z, which represent two-dimensional integrals over the coordinates of electrons 1 and 2, are generated recursively. Each individual term depends on the angular momentum of the four basis functions as well as t_α,

$$I_x(t_\alpha) \equiv I_x(i_x, j_x, k_x, l_x, t_\alpha) \tag{10}$$

but, for the following discussion the dependence on t_α will be suppressed. The first step is to evaluate

$$I_x(0,0,0,0) = \frac{\pi}{\sqrt{AB}} \exp\left(-\frac{\alpha_i \alpha_j}{\alpha_i + \alpha_j}(x_i - x_j)^2 - \frac{\alpha_k \alpha_l}{\alpha_k + \alpha_l}(x_k - x_l)^2 \right) \tag{11}$$

Next, three coefficients must be defined, B_{00}, B_{10}, C_{00}, which depend on t_α, A, B, P_x, Q_x, and the coordinates of the basis function centers.

$$B_{00} = \frac{1}{2(A+B)} t_\alpha^2 \tag{12}$$

$$B_{10} = \frac{1}{2A} - \frac{B}{2A(A+B)} t_\alpha^2 \tag{13}$$

$$C_{00} = (P_x - x_i) + \frac{B(Q_x - P_x)}{A+B} t_\alpha^2 \tag{14}$$

The following can be used to generate $I_x(i_x + j_x, 0, k_x + l_x, 0)$ recursively, (eqs. (40, 44) in reference 4)

$$I_x(n+1,0,m,0) = nB_{10} I_x(n-1,0,m,0) + mB_{00} I_x(n,o,m-1,0) + C_{00} I_x(n,0,m,0) \tag{15}$$

and

$$I_x(n,0,m+1,0) = mB'_{10} I_x(n,0,m-1,0) + nB_{00} I_x(n-1,0,m,0) + C'_{00} I_x(n,0,m,0) \tag{16}$$

where the primed coefficients are those obtained upon transposing A, P_x, and x_i with B, Q_x, and x_k in eqs. (13) and (14).

The final step in evaluating $I_x(i_x, j_x, k_x, l_x)$ from $I_x(i_x + j_x, 0, k_x + l_x, 0)$ is to "shift" the dependence on just the two functions to the proper dependence on all four. Rys, King, and Dupuis derived recursion formulas (eqs. (52) and (53) of reference 4) for this

$$I_x(i_x, j_x, m, 0) = I_x(i_x + 1, j_x - 1, m, 0) + (x_i - x_j) I_x(i_x, j_x - 1, m, 0) \tag{17}$$

$$I_x(i_x, j_x, k_x, l_x) = I_x(i_x, j_x, k_x + 1, l_x - 1) + (x_k - x_l) I_x(i_x, j_x, k_x, l_x - 1) \tag{18}$$

However, instead of following this directly, since it requires collecting intermediate values, we elect to use the following summation.

$$I_x(i_x, j_x, m, 0) = \sum_{n=0}^{j_x} q_n I_x(i_x + n, 0, m, 0), \quad \text{where} \quad q_n = \binom{j_x}{n} (x_j - x_i)^{(j_x - n)} \tag{19}$$

An analogous relationship holds for finding the final $I_x(i_x, j_x, k_x, l_x)$ from $I_x(i_x, j_x, k_x + n, 0)$. The coefficients, q_n, depend only on binomial coefficients and the geometrical coordinates of the functions. (If the basis is a contracted set where all primitives of a contracted function have the same center, then the q_n coefficients need to be constructed just once for each integral over contracted functions.) This formulation eliminates finding many intermediate values. Using eqs. (17) and (18) requires finding $I_x(n, 0, m, 0)$ for all n and m, where $i_x \le n \le i_x + j_x$ and $k_x \le m \le k_x + l_x$, to compute the final $I_x(i_x, j_x, k_x, l_x)$. This requires $j_x(j_x + 1)/2$ applications of eq. (17), to find a value of $I_x(i_x, j_x, m, 0)$, while there are only $j_x + 1$ terms in the summation of eq. (19). Comparison between use of eqs. (17) and (18) versus eqs. (19) in the test calculations described later showed that the time required for this shifting process (see the subroutine SHIFT in the Appendix) is reduced by a factor of about two and a half to three using eq. (19).

After substituting eq. (9) into eq. (5), the repulsion integral becomes

$$(ij|kl) = 2\sqrt{\rho/\pi} \sum_{\alpha=0}^{N} I_x(t_\alpha) I_y(t_\alpha) I_z(t_\alpha) W_\alpha \tag{20}$$

Evaluating I_x, I_y, and I_z for each root is the slowest operation, but the efficiency is enhanced by inverting the loop structure so that the values of I_x are found for all roots and then the values for I_y and then for I_z. Also, since the loop over α will often be the longest loop, this structure automatically takes partial advantage of vector pro-

Table I. Timing comparisons between Macintosh II, VAX 11/780 and FPS M65/34 (in seconds[a]).

Basis[b]	N[b]	Mac II	VAX	Mac/VAX	FPS	VAX/FPS
s	1	57	18	3.2	14	1.3
p	3	233	66	3.5	22	3.0
d	5	494	170	2.9	31	5.5
f	7	1083	384	2.8	50	7.7
g	9	2139	762	2.8	79	9.6
h	11	3799	1353	2.8	123	11.0
i	13	6201	2187	2.8	182	12.0
j	15	9483	3334	2.8	262	12.7

[a]The Macintosh II and FPS timings are clock time, while the VAX timings are processor times. The calculations on the Macintosh and FPS were run alone, so that the timings would be accurate for comparison.
[b]Each basis set consisted of 20 functions, all of the same l (see text). N is the required order of Rys polynomials needed for the integral evaluation.

cessing capability. This would be manifested as an improving vector/scalar advantage with increasing l of the basis because the α-loops get longer. To test this, we carried out calculations with s, p, d, f, g, h, i, and j functions, in each case with just the purely x component: x, xx, xxx, $xxxx$, and so on. For each calculation, the basis consisted of 20 Gaussians, with four Gaussians placed on five centers.[14] The Gaussian exponents, chosen to span representative exponent values, were 10, 2.5, 0.5, and 0.1. Calculations were carried out on a Macintosh II, a VAX 11/780 and an FPS M65/34 Array Processor. The M65/34 has a pipeline architecture, with performance characteristics of a (short) vector processor. The results given in Table I show the ratio of Macintosh-to-Vax timings to be relatively independent of the order of Rys polynomials; it is about 2.8. Comparing the VAX and FPS timings shows an improving vector-to-scalar timing ratio with increasing order of Rys polynomials. The plateau that is approached partly reflects saturation of the M65/34 pipeline, as N becomes large (>13). The Appendix shows the concise, essential coding for the whole procedure.[15]

The conciseness of this open-ended implementation of the Rys polynomial scheme, achieved by a quadrature-point-driven loop structure, opens up this part of a large scale *ab initio* calculation to very small computers, while extending the l-limit of a basis as far as desired.

APPENDIX

Following are segments of the subroutines which find the values of $I_x(i_x, j_x, k_x, l_x)$. These routines are called for I_x, I_y, and I_z separately. The DO loop statements in bold are the quadrature-point-driven loops. NDIM, the key dimension value in a calculation, must be greater than $2l + 1$. LAMBDA is the order of the required Rys polynomial. I and K are the Cartesian powers of the i and k functions, for the appropriate Cartesian direction, i.e., i_x and k_x. N and M are the sums of the Cartesian powers of the i, j functions and k, l functions, respectively, i.e., $N = i_x + j_x$ and $M = k_x + l_x$. The array G holds the $I_x(n, 0, m, 0)$ values given by eqs. (15), (16), the array GG holds the intermediate values found with eq. (19), e.g., $I_x(i_x, j_x, m, 0)$; and the array VV holds the final values, $I_x(i_x, j_x, k_x, l_x)$. VAL initially holds the weight factors, and after the three calls of the subroutines holds the values of $I_x(t_\alpha)I_y(t_\alpha)I_z(t_\alpha)W_\alpha$. The arrays XXX and YYY hold the coefficients, q_n, defined in eq. (19), which depend on XX ($= x_i - x_j$) and YY ($= x_k - x_l$), and the binomial coefficients stored in NBIN.

```
      SUBROUTINE RECUR(N, M, LAMBDA, NDIM, A, B, XA, XB, XI, XK, ROOT,
     1                 BBB, B10, B01, CCC, CPP, G)
      DIMENSION  BBB(LAMBDA), B10(LAMBDA), CCC(LAMBDA),B01(LAMBDA)
      DIMENSION  CPP(LAMBDA), ROOT(LAMBDA), G(LAMBDA,NDIM,NDIM)
      XAB = XA - XB                                  XAB is -(Qx - Px) in Eqn. (14)
      XXA = XA - XI                                  XXA is Px - xi in Eqn. (14)
      XXB = XB - XK                                  XXB is Qx - xk in Eqn. (14)
      BXBA = -B * XAB
      AXAB =  A * XAB
      AB = 1.0D 0 / (A + B)
      DO 20 I=1,LAMBDA
         ROOTAB = ROOT(I) * AB
         CCC(I) = XXA + BXBA * ROOTAB                CCC is C00 in Eqn. (14)
         BBB(I) = 0.5D 0 * ROOTAB                    BBB is B00 in Eqn. (12)
         B10(I) = (0.5D 0 - BBB(I) * B) / A          B10 is B10 in Eqn. (13)
         B01(I) = (0.5D 0 - BBB(I) * A) / B          B01 is B10' in Eqn. (13)
         CPP(I) = XXB + AXAB * ROOTAB                CPP is C00' in Eqn. (14)
         G(I,2,1) = CCC(I)                                        Eqn. (15)
         G(I,1,2) = CPP(I)                                        Eqn. (16)
   20 CONTINUE
      IF (N .LE. 1) GO TO 50
```

```
      DO 40 I=1,LAMBDA
         BN = 0.0
         DO 40 NN = 1, N-1
            BN = BN + B10(I)
            G(I,NN+2,1) = BN * G(I,NN,1) + CCC(I) * G(I,NN+1,1)          Eqn. (15)
   40 CONTINUE
   50 IF (M .LE. 1) GO TO 80
      DO 70 I=1,LAMBDA
         BM = 0.0
         DO 70 MM = 1, M-1
            BM = BM + B01(I)
            G(I,1,MM+2) = BM * G(I,1,MM) + CPP(I) * G(I,1,MM+1)          Eqn. (16)
   70 CONTINUE
   80 IF (N .EQ. 0 .OR. M .EQ. 0) RETURN
      DO 90 I=1,LAMBDA
         BN = 0.0
         DO 90 NN = 1, N
            BN = BN + BBB(I)
            G(I,NN+1,2) = BN * G(I,NN,1) + CPP(I) * G(I,NN+1,1)          Eqn. (15)
            IF(M .LE. 1) GO TO 100
            BM = 0.0
            DO 90 MM = 1, M-1
               BM = BM + B01(I)
               G(I,NN+1,MM+2) = BM * G(I,NN+1,MM) + BN * G(I,NN,MM+1)
     1                        + CPP(I) * G(I,NN+1,MM+1)                 Eqn. (16)
   90 CONTINUE
      END


      SUBROUTINE SHIFT(LAMBDA, NDIM, I, K, N, M, G, GG, VV, VAL, XX, YY)
      DIMENSION  XXX(NDIM), YYY(NDIM), VAL(LAMBDA), VV(LAMBDA)
      DIMENSION  GG(LAMBDA,NDIM), G(LAMBDA,NDIM*NDIM)
      DATA  XXX(1)  /  1.0 D 0 /,   YYY(1)   /  1.0D 0   /
      NPOLY = N - I + 1
      IF(NPOLY .LT. 2)  GO TO 15
      TEMP = 1.0D 0
      DO 10 KP=2,NPOLY
         TEMP = TEMP * XX
         XXX(KP) = TEMP * NBIN(KP,NPOLY)
   10 CONTINUE
   15 MPOLY = M - K + 1
      IF(MPOLY .LT. 2) GO TO 25
      TEMP = 1.0D 0
      DO 20 KP=1,MPOLY
         TEMP = TEMP * YY
         YYY(KP) = TEMP * NBIN(KP,MPOLY)
   20 CONTINUE
   25 IPOINT = (K - 1) * NDIM
      DO 50 IB=K+1,M+1
         IPOINT = IPOINT + NDIM
         IPOLY = NPOLY
         DO 50 IA=I+1,N+1
            C = XXX(IPOLY)                          C is qn in Eqn. (19)
            IPOLY = IPOLY - 1
            IP = IPOINT + IA
            DO 50 IL=1,LAMBDA
               GG(IL,IB) = GG(IL,IB) + G(IL,IP) * C                      Eqn. (19)
   50 CONTINUE
      DO 80 IJK=K+1,M+1
         C = YYY(MPOLY)                             C is qn in Eqn. (19)
         MPOLY = MPOLY - 1
         DO 80 IL=1,LAMBDA
            VV(IL) = VV(IL) + C * GG(IL,IJK)                           Eqn. (19)
   80 CONTINUE
      DO 90 IL=1,LAMBDA
         VAL(IL) = VAL(IL) * VV(IL)                                    Eqn. (9)
   90 CONTINUE
      END
```

This work was supported, in part, by the Chemical Physics Program of the National Science Foundation (Grant No. CHE-8721467).

References

1. H. Taketa, S. Huzinaga, and O. Ohata, *J. Phys. Soc. Jap.*, **21,** 2313 (1966).
2. H. F. King and M. Dupuis, *J. Comp. Phys.*, **21,** 144 (1976).
3. M. Dupuis, J. Rys, and H. F. King, *J. Chem. Phys.*, **65,** 111 (1976).
4. J. Rys, M. Dupuis, and H. F. King, *J. Comp. Chem.*, **4,** 154 (1983).
5. C. E. Dykstra, J. D. Augspurger, B. Kirtman, and D. J. Malik, in "Reviews in Computational Chemistry," K. Lipkowitz and D. Boyd, Ed., Vol.1, VCH Publishers, New York, 1990, p. 83.
6. S. Obara and A. Saika, *J. Chem. Phys.*, **84,** 3963 (1986); *J. Chem. Phys.*, **89,** 1540 (1989).
7. M. Head-Gordon and J. A. Pople, *J. Chem. Phys.*, **89,** 5777 (1989).

8. J.A. Pople and W.J. Hehre, *J. Comp. Phys.*, **27,** 161 (1978).
9. V.R. Saunders, in "Methods in Computational Molecular Physics," G.H.F. Diercksen and S. Wilson, Ed., Reidel, Dordrecht, 1983.
10. L.E. McMurchie and E.R. Davidson, *J. Comp. Phys.*, **26,** 218 (1978).
11. The program requires about 600 lines of FORTRAN code exclusive of comments, and only about 75 of those lines (as shown in the Appendix) are responsible for most of the computation, by an amount that increases with the order of Rys polynomial (70% for $N = 3$, 85% for $N = 5$, and over 95% for $N = 9$).
12. W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, Numerical Recipes in FORTRAN, (Cambridge University Press, New York, 1986).
13. Other methods of storage of the roots (weights) could be used. The method chosen is economical in the amount of data required in memory as well as look-up time.
14. The four function centers were placed at the following (x,y,z) coordinates (in Å): $(0,0,0)$, $(2,0,0)$, $(0,1,0)$, $(1,1,1)$, and $(-1.5,0,-1)$.
15. The program is module I of the UNIVERSE 2000 program system.