J. Symbolic Computation (1998) **25**, 367–382

Generation and Verification of Algorithms for Symbolic-Numeric Processing

LADISLAV KOCBACH[†] AND RICHARD LISKA[‡]

[†] *Department of Physics, University of Bergen, Allégaten 55, N-5007 Bergen, Norway*
[‡] *Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague Břehová 7, 115 19 Prague, Czech Republic*

Some large scale physical computations require algorithms performing symbolic computations with a particular class of algebraic formulas in a numerical code. Developing and implementing such algorithms in a numerical programming language is a tedious and error prone task. The algorithms can be developed in a computer algebra system and their correctness can be checked by comparison with build-in facilities of the system so that the system is used as an advanced debugging tool. After that a numerical code for the algorithms is automatically generated from the same source code. The proposed methodology is explained in detail on a simple example. Real applications to calculation of matrix elements of Coulomb interaction and two-centre exchange integrals needed in atomic collision codes, are described. The method makes the developing and debugging of such algorithms easier and faster.

1. Introduction

In certain large scale physical numerical calculations in e.g. quantum physics, one needs to include analytical operations like integration or differentiation of a particular type of algebraic expressions, e.g. products of polynomials and exponentials. Most often such operations are performed manually and the resulting expressions are coded in a language suitable for numerical computations, in some other cases large parts are performed entirely numerically. Recently it has been found advantageous to include some of the symbolic manipulation inside of the numerical codes, see e.g. Hansen (1990). Such procedures might yield higher precision, improve the efficiency and, in particular, result in more general codes.

There exist environments which allow symbolic and numeric algorithms to be used together in compiled code, e.g. AXIOM-XL, the AXIOM Extension Language (Watt *et al.*, 1994), in Axiom (Jenks and Sutor, 1992), but using such an environment would not meet the aims of the applications as it would not produce portable numerical code and its speed would be less than the speed of a purely numerical code, coded in e.g., FORTRAN.

Developing and implementing algorithms for symbolic manipulation in a language such as FORTRAN is a tedious and error prone task, while the computer algebra systems

(CASs) are designed for symbolic manipulation and it is quite straightforward to carry out both purely symbolic and mixed symbolic and numerical evaluations in many of the available CASs.

To distinguish the algorithms which perform symbolic manipulation in a numerical programming language from other methods, we propose calling them *symbolic-numeric* algorithms. These typically use the fixed length integer and floating point representations, and thus lose the absolute precision of CASs, but they evaluate large quantities of similar expressions orders of magnitude faster than CAS and are thus applicable to large scale calculations. By *symbolic* algorithms we mean the algorithms dealing with formulas and using the facilities for formula processing implemented in CASs.

This paper presents a new application of CAS to the development of numerical codes, which might contain a certain amount of symbolic manipulation. In the method presented here, the algorithms suitable for implementation in a *numerical* code are designed by humans and coded in language of a CAS. The algorithms are tested by comparing their results on representative input data sets with the results of the *symbolic* algorithms included in the CAS. The comparison is made at the algebraic level in the CAS. Strictly speaking by this comparision we are not attempting to prove the correctness of the algorithms but we are verifying the correctness of their implementation on finite input data sets. The numerical implementation is then automatically generated from the same source, provided that the CAS in question contains a facility to convert both mathematical functions and its own control language into a code in programming language suitable for the numerical applications. The method uses a CAS for advanced debugging of symbolic-numeric algorithms and also allows comparisons of the algorithms in different floating-point arithmetics. The main advantage of our method is the improvement and speeding up of the development-debugging process. We assert that debugging symbolic-numeric algorithms can be done much more effectively in a CAS than in a numeric programming language.

Using such an approach one can have strong confidence that the numerical code is correct. So here the knowledge from the computer algebra system is used to verify the correctness of the proposed algorithms. In the work reported here, the computer algebra system REDUCE (Hearn, 1995) with the standard code generation package GENTRAN (Gates, 1986) is used to develop codes in FORTRAN.

Generally we deal here with the development of a particular symbolic processing algorithm which is usually used as part of a large numerical code. Typically the algorithm deals only with a special domain of formulas. Many papers (e.g. Cook, 1990; Dewar and Richardson, 1990; Kant, 1993; Steinberg and Roache, 1985; Wang, 1986) have dealt with the code generation of numerical algorithms. Some work has also been reported on program transformation techniques (e.g. Zippel, 1992) and automatic differentiation of numerical codes (Rostaing *et al.* , 1993); however, we are unaware of any work using code generation of symbolic algorithms.

The paper is organized as follows. Section 2 provides a general description of the method presented for the development of verified symbolic-numeric algorithms which is explained in full detail on an elementary example in Section 3. Two particular applications of the method to practical problems in atomic physics are described in detail in Sections 4 and 5. The first application deals with the calculation of the matrix elements of a Coulomb interaction between two bound hydrogenic states and the second deals with the calculation of three-dimensional two-centre exchange integrals with travelling orbitals. Both examples are of interest in the impact-parameter treatment of atom–atom or ion–atom collisions. In these applications the calculations must be repeated for many collision parameters and fast methods for numerical evaluation are essential. At the same time, large amounts of symbolic evaluations are needed in order to set up the formulae used in the numerical work. The test applications of the generated symbolic-numeric codes are discussed briefly in Section 6.

2. Development of Verified Algorithms

We need to implement a particular symbolic processing algorithm $\mathcal{A}$ in a numerical programming language $\mathcal{L}$. In general this implementation and mainly its debugging could be quite tedious, while the algorithm $\mathcal{A}$ can be usually implemented very simply in a computer algebra system (CAS) as the CAS already includes many symbolic algorithms which are typically used as parts of the algorithm $\mathcal{A}$. The algorithm $\mathcal{A}$ is dealing with formulas from the domain $\mathcal{D}$. To enhance the debugging and verify the correctness of the developed algorithm we can use the approach described in general in this section. In this general description we use a CAS and the programming language $\mathcal{L}$ while in the applications we have used the CAS REDUCE and the numerical programming language FORTRAN. For a simple example illustrating this method see Section 3.

2.1. symbolic implementation

The algorithm $\mathcal{A}$ is implemented in a CAS. The formulas from $\mathcal{D}$ are represented in the CAS as its standard formulas. The implementation and debugging is usually quite simple. For debugging and verification the CAS offers much better tools than the language $\mathcal{L}$. The symbolic implementation is assumed to be error free. We believe that we can assume this as the symbolic implementaion is really very simple, see Fig. 1 and Equations (4.1) and (5.4) showing formulas implemented in practical applications.

2.2. data representation

To implement the algorithm $\mathcal{A}$ in the language $\mathcal{L}$ we have to choose the representation $\mathcal{R}$ of formulas from the domain $\mathcal{D}$ in the data structures of the language $\mathcal{L}$. For this method we further need that the used data structures and the control commands (e.g. loops and conditions) are also supported by the CAS and that the CAS supports the code generation of these structures and control commands in the language $\mathcal{L}$. Typically these structures include only integers, floats and their arrays, e.g. a polynomial in one variable can be represented by an array of its coefficients. To represent the multivariate polynomials, that appear in parts of the processed formulas, we use either sparse representation, i.e. storing coefficients and degrees, or dense representation, i.e. storing all coefficients. These representations have been chosen so that the implementation of the symbolic-numeric algorithms using them might be quite simple.

We should note here that the representation $\mathcal{R}$ does not usually need to be absolutely precise, i.e. including big integers, as the developed symbolic-numeric algorithm will be finally used in the numerical code which does not require absolute precision. However we have to be aware of possible rounding effects during the development of a symbolic-numeric algorithm, e.g. testing whether a number is zero when a rational number is replaced by a float number.

2.3. SYMBOLIC-NUMERIC IMPLEMENTATION

The algorithm $\mathcal{A}$ is implemented again in the CAS, however now we use for the formulas from the domain $\mathcal{D}$ the representation $\mathcal{R}$ and use only the operations and semantics supported by the language $\mathcal{L}$. It may appear strange to represent in the CAS formulas by the representation $\mathcal{R}$, e.g. polynomials by arrays of their coefficients, but this is precisely what is needed for verification of the symbolic-numeric implementation. The symbolic-numeric implementation is necessarily much lengthier and complicated than the symbolic implementation. This is because many subalgorithms of the algorithm $\mathcal{A}$ are known to the CAS, e.g. the addition of two polynomials, and can be directly used in the symbolic implementation while these subalgorithms have to be coded in the symbolic-numeric implementation. In other words, the two implementations are using different tools. In the symbolic implementation, the CAS is used with all the facilities it supports, while in the symbolic-numeric implementation, though coded in the CAS language, only the facilities (data structures, semantics, algorithms) supported by the language $\mathcal{L}$ can be used.

2.4. VERIFICATION

Now we have two implementations of the algorithm $\mathcal{A}$, the symbolic implementation and the symbolic-numeric implementation, both implemented in the CAS. The symbolic implementation is assumed to be correct so we can verify the symbolic-numeric implementation by comparing the results of both on a representative set of input data to algorithm $\mathcal{A}$. If the two implementations produce different results, a bug from the symbolic-numeric implementation has to be removed. The comparision is done in the precise arithmetics of the CAS. If needed in critical cases one can check the numerical quality of symbolic-numeric implementation by comparing in the CAS the results in two rounded arithmetics of different precisions. At the end of this step we have confidence that the symbolic-numeric implementation is error free.

It might be argued that the same type of error might somehow propagate into both the symbolic and the symbolic-numeric representation. Since two entirely different approaches are used, the chance that an accidental error would lead to a fortutious agreement over a wide range of parameters seems to be very small indeed. On the other hand, the method cannot provide a rigorous proof of the correctness, only demonstrate the validity in the tested domain. It does, however, represent a major improvement over the methods usually used in verification of numerical codes. One further advantage is that if the algorithms are to be implemented in a new application with certain change of scope, the testing can be performed once more with stress on the new features. This is shortly discussed below in Section 6.

Usually the verification is done over input data from a direct product of small integer sets. However the results also depend on some other parameters, e.g. R and a in Section 3. If we are checking the symbolic-numeric implementation in the CAS these parameters are, as parts of formulas, treated symbolically while if we were to debug these algorithms in a numerical environment we would also need to check the results for many numerical values of these parameters. This is one of the points which makes the development-debugging process easier by the presented method.

2.5. CODE GENERATION

By using a code generation facility of the CAS the required implementation of the algorithm $\mathcal{A}$ in the language $\mathcal{L}$ is automatically generated from the verified symbolic-numeric implementation. The final result is the verified source code in language $\mathcal{L}$ implementing the symbolic-numeric algorithm $\mathcal{A}$.

The numerical procedures obtained and tested by the described method might not be optimal, neither from the point of speed nor accuracy (e.g. avoiding accumulation of round-off errors), but the code is, in principle, error-free in our sense. It can thus serve as a base for further optimalization which can be performed by changing the symbolic-numeric implementation with regard to these aspects. The existence of an error-free code when developing new versions will be recognized as a great advantage by anybody who has worked on similar problems. It should also be mentioned that some special evaluation techniques can be encoded in the symbolic-numeric implementation and thus be automatically verified by the described method (as, e.g., the Horner scheme evaluation of polynomials).

3. Simple Example

For better understanding of the development method described in the previous section, a very simple example, for which the method will be presented in detail, is included here. The problem considered is to implement in FORTRAN the program which calculates the integral

$$I(R,n,a) = \int_0^R x^n e^{-ax}\,\mathrm{d}x, \tag{3.1}$$

for input parameters R, n, a where $n \geq 0$ is integer and $R \geq 0$, $a \geq 0$ are floating point numbers. The integral can be numerically integrated, however for any n it can be evaluated to

$$I(R,n,a) = e^{-aR}\sum_{i=0}^{n} C_i R^{j_i} + A \tag{3.2}$$

which would give a faster and more precise routine. We will develop this routine by applying our method.

3.1. SYMBOLIC IMPLEMENTATION

The REDUCE program for calculation of integral (3.1), presented in Figure 1, is really simple and does not need any comments.

Note that in this example we could proceed by calculating (3.1) for let say $n = 0, \dots, 20$ with R, a as parameters and then generate the FORTRAN routine including the results in the form (3.2). However, such approach has disadvantages, e.g. later we need to calculate $I(R, 25, a)$ and have to make another code generation, and it is impossible to apply such an approach to more complicated cases where the number of necessary formulas can be very large (e.g. of the order 10^4 as in Section 4). So we need to perform the manipulation with formulas on a numerical level in FORTRAN. The actual evaluation of integral (3.1) suitable for the numerical work is done in the alternative way described below.

```
procedure integ(r,n,a);
%  Calculates the definite integral
%    int_0^r x^n exp(-a x) d x
%  Input: r,n,a  - parameters of the integral, n has to be non-negative integer
%  Output: value of the procedure - the definite integral
begin
  scalar y;
  y := int(x^n*e^( - a*x),x);
  return (sub(x=r,y) - sub(x=0,y));
end;
```

Figure 1. Symbolic implementation of (3.1), file `integ.alg`

3.2. DATA REPRESENTATION

All formulas needed for calculating (3.1) have the form (3.2) which we need to take with particular values of parameters n, a keeping R as variable. Such formula will be represented by two floats `a` $= a$, `abs` $= A$, an array of integer exponents `oexp(i)` $= j_{i+1}$ and an array of floating-point coefficients `ocof(i)` $= C_{i+1}$, where we have made the shift by 1 in indices so that the FORTRAN arrays will begin from the standard index 1.

3.3. SYMBOLIC-NUMERIC IMPLEMENTATION

To implement the calculation of integral (3.1) in terms of array representation described in the previous section without the use of the REDUCE operator for integration `int`, we need to derive explicit formula (3.2) for calculation of (3.1). Applying several times integration per partes we get

$$I(R,n,a) = \int_0^R x^n e^{-ax}\,\mathrm{d}x = -\frac{1}{a}R^n e^{-ax} - \frac{n}{a^2}R^{n-1}e^{-ax} - \frac{n(n-1)}{a^3}R^{n-2}e^{-ax} - \cdots - \frac{n!}{a^{n+1}}e^{-ax} + \frac{n!}{a^{n+1}},$$

(the last term $n!/a^{n+1}$ comes from the zero limit of the integral) from which we can deduce the recurrence relations for the degrees j_i and coefficients C_i and a formula for the absolute term A in (3.2):

$$\begin{aligned} j_0 &= n, \quad j_i = n - i, \\ C_0 &= -\frac{1}{a}, \quad C_i = C_{i-1}\frac{n+1-i}{a}, \quad i = 1,\ldots,n, \\ A &= \frac{n!}{a^{n+1}}. \end{aligned} \tag{3.3}$$

Recurrence relations appear regularly in symbolic-numeric implementations. The symbolic-numeric implementation based on (3.3) is shown on Figure 2. Note that the procedure `pinteg` could be split into two procedures, one implementing (3.3) and the other (3.2), where the first procedure has to be called only after the change of n or a. For comments on declarations `scalar`, `operator`, `literal` and `declare` see Appendix A.

```
procedure fact(n);
begin
  literal"c  Calculates Factorial of n                                   ",cr!*;
  declare
    <<fact:function;
      fact,f:real*8;
      n,i:integer>>;
  f:= if n=0 then 1
       else for i:=1:n product i;
  return f
end;

procedure pinteg(r,n,a);
begin
  scalar abs,res;
  operator ocof,oexp;
  literal"c  Calculates the definite integral                            ",cr!*;
  literal"c    int_0^r x^n exp(-a x) d x                                 ",cr!*;
  literal"c  Input: r,n,a  - parameters of the integral                  ",cr!*;
  literal"c                  n has to be non-negative integer            ",cr!*;
  literal"c  Output: value of the procedure - the definite integral  ",cr!*;
  declare
    <<pinteg:function;
      n,i,oexp(50):integer;
      pinteg,ocof(50),r,a:real*8 >>;
  literal"c  Symbolic calculation of the integral                        ",cr!*;
  oexp(1) := n;
  ocof(1) := -1/a;
  for i:=2:n+1 do
    <<oexp(i) := n + 1 - i;
      ocof(i) := ocof(i-1)*(n+2-i)/a >>;
  abs := fact(n)/a^(n+1);
  literal"c  Evaluation of the integral                                  ",cr!*;
  res := 0;
  for i:=1:n+1 do res := res + ocof(i)*r^oexp(i);
  res := e^(-a*r)*res + abs;
  return res;
end;
```

Figure 2. Symbolic-numeric implementation of (3.1) based on (3.3), file `integ.pro`

3.4. VERIFICATION

The verification of the symbolic-numeric implementation (see Figure 2) has been performed by comparing its results with symbolic implementation (see Figure 1). The verification code is presented on Figure 3. Its last line gives the result zero proving that the symbolic-numeric implementation is correct for $n = 0, \dots, 50$ from which we assume it to be correct for all non-negative integer n.

3.5. CODE GENERATION

Having verified the symbolic-numeric implementation in the file `integ.pro` we can generate the FORTRAN code directly from this file. The code generation commands shown in Figure 4 are really very simple. Note that we generate the code directly from

```
in "genproc.red"; % to read and eliminate DECLARE and LITERAL
in "integ.pro";   % definition of pinteg - in terms of array-operators
in "integ.alg";   % definition of integ  - algebraic algorithm

% testing of procedures in integ.pro by comparing with symbolic calculation

for n:=0:50 sum abs(integ(r,n,a) - pinteg(r,n,a));
```

Figure 3. Verification of symbolic-numeric implementation, file `integ.tst`

```
in "genproc.red";  % loads gentran, defines switch genproc

on genproc;

gentranout "integ.f";
in "integ.pro";
gentranshut "integ.f";
```

Figure 4. Code generation of FORTRAN symbolic-numeric implementation, file `integ.gen`

the file `integ.pro`. To be able to use exactly the same file without any modifications which could introduce errors, we have introduced a new switch, `genproc`, described in Appendix A. This switch and the associated actions are responsible for code generation by interfacing the code generation package GENTRAN (Gates, 1986). The generated FORTRAN code implementing the evaluation of the integral (3.1) is presented in Figure 5. This code is guaranteed (if we have not made an error in the few lines shown in Figure 1) to be error free for $N \leq 50$. For evaluating of (3.1) it uses the developed symbolic-numeric implementation. Limitation on the maximum n given by the array bounds (here 50) can be avoided by increasing the array bounds. Then the algorithm is also assumed to be error free also for $N > 50$.

4. Matrix Elements of Coulomb Interaction

The methodology outlined in Section 2 and demonstrated on the simple example in the previous section has been applied for preparation of numerical FORTRAN code for calculating the matrix elements of Coulomb interaction of two bound hydrogenic states. These calculations can be separated into radial and angular parts. For the actual calculations codes both parts are constructed with the assistance of the CAS REDUCE, however here we discuss only the radial parts where the described techniques are used.

The radial matrix elements are given by

$$M_{l_1 l_2}^{n_1 n_2}(l) = \frac{1}{R^{l+1}} \int_0^R r^{l+2} R_{n_1 l_1}(r) R_{n_2 l_2}(r)\, \mathrm{d}r + R^l \int_R^\infty \frac{1}{r^{l-1}} R_{n_1 l_1}(r) R_{n_2 l_2}(r)\, \mathrm{d}r, \quad (4.1)$$

where n_1, n_2, l_1, l_2 are quantum numbers, $l_1 - l_2 \leq l \leq l_1 + l_2$ is the transferred angular

```
      REAL*8 FUNCTION FACT(N)
      REAL*8 F
      INTEGER N,I
c  Calculates Factorial of n
      IF (N.EQ.0.0) THEN
          F=1.0
      ELSE
          F=1
          DO 25001 I=1,N
              F=F*I
25001     CONTINUE
      ENDIF
      FACT=F
      RETURN
      END
      REAL*8 FUNCTION PINTEG(R,N,A)
      INTEGER N,I,OEXP(50)
      REAL*8 OCOF(50),R,A
c  Calculates the definite integral
c    int_0^r x^n exp(-a x) d x
c  Input: r,n,a  - parameters of the integral
c                 n has to be non-negative integer
c  Output: value of the procedure - the definite integral
c  Symbolic calculation of the integral
      OEXP(1)=N
      OCOF(1)=-(1.0/A)
      DO 25002 I=2,N+1
          OEXP(I)=N+(1-I)
          OCOF(I)=OCOF(I-1)*((N+(2.0-I))/A)
25002 CONTINUE
      ABS=FACT(N)/A**(N+1)
c  Evaluation of the integral
      RES=0.0
      DO 25003 I=1,N+1
          RES=RES+OCOF(I)*R**OEXP(I)
25003 CONTINUE
      RES=EXP(REAL(-(A*R)))*RES+ABS
      PINTEG=RES
      RETURN
      END
```

Figure 5. FORTRAN code (file `integ.f`) generated by GENTRAN from the file `integ.pro` (Figure 2) is the resulting implementation of the symbolic-numeric algorithm in FORTRAN.

momentum quantum number and $R_{nl}(r)$ are radial hydrogenic functions

$$R_{nl}(r) = \frac{\bar{R}_{nl}(r)}{\sqrt{\int_0^\infty \bar{R}_{nl}^2(r) r^2 \, \mathrm{d}r}}, \quad \text{where} \quad \bar{R}_{nl}(r) = L_{2l+1}^{n+l}\left(\frac{2r}{n}\right) r^l e^{-r/n}, \tag{4.2}$$

where $L_j^k(r)$ are generalized Laguere polynomials

$$L_j^k(r) = \frac{\mathrm{d}^j}{\mathrm{d}r^j}\left(e^r \frac{\mathrm{d}^k r^k e^{-r}}{\mathrm{d}r^k}\right). \tag{4.3}$$

4.1. SYMBOLIC IMPLEMENTATION

The symbolic implementation for calculating the matrix elements (4.1) is done by a few lines of code implementing formulas (4.1), (4.2) and (4.3) using the operators performing differentiation and integration.

4.2. DATA REPRESENTATION

The limitation on l and properties of the radial hydrogenic functions $R_{nl}(r)$ guarantee that any matrix element (4.1) can be expressed as

$$M_{l_1 l_2}^{n_1 n_2}(l) = \frac{1}{R^{l+1}} \int_0^R P_1(r) e^{-ar}\,\mathrm{d}r + R^l \int_R^\infty P_2(r) e^{-ar}\,\mathrm{d}r, \tag{4.4}$$

where $a = 1/n_1 + 1/n_2$ and $P_i(r), i = 1, 2$ are polynomials in r. The matrix elements (4.4) result in the formula of the form

$$M_{l_1 l_2}^{n_1 n_2}(l) = e^{-aR} \sum_{i=0}^{n} C_i R^{j_i} + C_a R^{j_a} \tag{4.5}$$

which is very similar to (3.2). So for all processing we need to represent polynomials in one variable which we represent as in Section 3.2 by integer array of exponents j_i and floating point array of coefficients C_i.

4.3. SYMBOLIC-NUMERIC IMPLEMENTATION

For the symbolic-numeric implementation, calculating the radial hydrogenic functions $R_{nl}(r)$, the formula

$$R_{nl}(r) = \frac{2^{l+1}}{n^{l+2}} \sqrt{\frac{(n-l-1)!}{(n+l)!^3}} e^{-r/n} \sum_{i=0}^{n-l-1} (-1)^{i+1} \left(\frac{2}{n}\right)^i \frac{(n+l)!^2}{(n-l-i-1)!(2l+i+1)!i!} r^{l+i} \tag{4.6}$$

has been derived. The integration of a polynomial multiplied by e^{-ar} which is needed in (4.4) is transformed into a linear combination of the integrals (3.1) which are evaluated by (3.3) (actually a generalization of (3.3) working with polynomials has been developed). In addition the subalgorithms for polynomial addition, multiplication and calculation of the absolute term of a polynomial, which are too long to be reproduced here, have been implemented in the array representation. Finally the algorithm in the array representation for calculation of the matrix elements (4.1) has been built from all the foregoing subalgorithms.

4.4. VERIFICATION AND CODE GENERATION

The symbolic-numeric implementation has been verified by comparison of its results with the symbolic implementation. The matrix elements (4.1) for the quantum numbers $0 \le n_1 \le 6, 0 \le n_2 \le n_1$ (formulas are symmetric in n_1, n_2) $0 \le l_1 < n_1, 0 \le l_2 < n_2, l_1 - l_2 \le l \le l_1 + l_2$ (these restrictions are physical limitations on quantum numbers) have been calculated identically by both implementations. The typical quantum numbers used in the applications are small, usually the greatest one is around 4, so our verification test has covered most of the relevant region of quantum numbers. Only the verification of the whole algorithm has been done over this range of very small integers while the verification of all used subalgorithms which were mentioned above has been done by the same method up to the degree 20. Again as in the case of the simple example in Section 3.5 the same source file which includes the symbolic-numeric implementation has been used for the generation of a FORTRAN symbolic-numeric implementation. In such a way we have constructed the FORTRAN program for the analytical calculation of matrix elements (4.1). The algorithms used in the code have been verified.

5. Exchange Integrals of Heavy-particle Collisions

The three-dimensional overlap exchange integrals in the impact-parameter treatment of heavy-particle collisions have the form (McDowell and Coleman, 1970)

$$I(n_1, l_1, m_1, n_2, l_2, m_2) = \int \psi^*_{n_1 l_1 m_1}(\mathbf{r}_1)\, e^{i\mathbf{a}\cdot\mathbf{r}_1 + i\mathbf{b}\cdot\mathbf{r}_2} \psi_{n_2 l_2 m_2}(\mathbf{r}_2)\, \mathrm{d}\mathbf{r}_1 \tag{5.1}$$

where the star denotes complex conjugation and the hydrogenlike wavefunctions ψ_{nlm} with the quantum numbers n, l, m have the form

$$\psi_{nlm}(\mathbf{r}) = R_{nl}(r) Y_{lm}(\mathbf{r}), \qquad r = |\mathbf{r}|, \tag{5.2}$$

where R_{nl} are radial hydrogen functions (4.2) and Y_{lm} are spherical harmonics functions. The position vectors $\mathbf{r}_1$,$\mathbf{r}_2$ measured from the two centers are related by $\mathbf{r}_2 = \mathbf{r}_1 - \mathbf{R}$, where $\mathbf{R}$ is the vector connecting the two centers.

The wavefunctions can be expressed in cartesian coordinates as [†]

$$\psi_{nlm}(\mathbf{r}) = e^{-\alpha r} \sum_j C_j r^{n_j} x^{l_{xj}} y^{l_{yj}} z^{l_{zj}}. \tag{5.3}$$

Substituting this into (5.1) we get the exchange integral (5.1) as a linear combination of integrals

$$\begin{aligned} &i(n_a, l_{ax}, l_{ay}, l_{az}, n_b, l_{bx}, l_{by}, l_{bz}) = \\ &\int r_1^{n_a-1} x_1^{l_{ax}} y_1^{l_{ay}} z_1^{l_{az}} r_2^{n_b-1} x_2^{l_{bx}} y_2^{l_{by}} z_2^{l_{bz}} e^{i\mathbf{a}\cdot\mathbf{r}_1 + i\mathbf{b}\cdot\mathbf{r}_2 - \alpha r_1 - \beta r_2}\, \mathrm{d}\mathbf{r}_1. \end{aligned} \tag{5.4}$$

Using the method in Shakeshaft (1975) the integrals (5.4) can be transformed into one-dimensional integrals

$$\begin{aligned} &i(n_a, \mathbf{l}_a, n_b, \mathbf{l}_b) = 2\pi(-i)^{(\mathbf{l}_a+\mathbf{l}_b)\cdot\mathbf{1}} \\ &\int_0^1 \left[\left(-\frac{\partial}{\partial\alpha}\right)^{n_a} \left(-\frac{\partial}{\partial\beta}\right)^{n_b} \nabla_{\mathbf{a}}^{\mathbf{l}_a} \nabla_{\mathbf{b}}^{\mathbf{l}_b} \frac{e^{i\mathbf{x}\cdot\mathbf{R} - R\sqrt{y}}}{\sqrt{y}} \right] \mathrm{d}w. \end{aligned} \tag{5.5}$$

where we have used notation

$$\begin{aligned} \mathbf{x} &= \mathbf{b}w - \mathbf{a}(1-w), \\ y &= \alpha^2(1-w) + \beta^2 w + (\mathbf{a}+\mathbf{b})^2 w(1-w), \\ \mathbf{l} &= (l_x, l_y, l_z), \end{aligned}$$

† Here $n_j, l_{xj}, l_{yj}, l_{zj}$ are integer degrees, not quantum numbers.

$$\begin{aligned}\nabla_{\mathbf{a}}^{\mathbf{k}} &= \nabla_{(a_x,a_y,a_z)}^{(k_x,k_y,k_z)} = \left(\frac{\partial}{\partial a_x}\right)^{k_x} \left(\frac{\partial}{\partial a_y}\right)^{k_y} \left(\frac{\partial}{\partial a_z}\right)^{k_z} \\ \mathbf{1} &= (1,1,1)\end{aligned}$$

The exchange matrix elements are obtained by numerical evaluation of the one-dimensional integrals (5.5). The symbolic-numeric methodology is applied to the evaluation of the integrands.

5.1. symbolic implementation

The expression for the integrand in formula (5.5) can be directly implemented symbolically which allows us, having also implemented calculating the wavefunctions (5.2) in the form (5.3), to calculate the exchange integrals (5.1).

5.2. symbolic-numeric implementation and data representation

Here prior to proposing the data representation which will be used in the symbolic-numeric implementation we have developed a new method for calculating the derivatives in (5.5). In Kocbach and Liska (1994) it has been shown that (5.5) can be written in the closed form

$$\begin{aligned} i(n_a,\mathbf{l}_a,n_b,\mathbf{l}_b) &= 2\pi(-i)^{(\mathbf{l}_a+\mathbf{l}_b)\cdot\mathbf{1}}(-1)^{n_a+n_b} \\ &\sum_{\mathbf{k}_a=(0,0,0)}^{\mathbf{l}_a} \sum_{\mathbf{k}_b=(0,0,0)}^{\mathbf{l}_b} \binom{\mathbf{l}_a}{\mathbf{k}_a}\binom{\mathbf{l}_b}{\mathbf{k}_b} \int_0^1 (-iw_1\mathbf{R})^{\mathbf{k}_a}(iw\mathbf{R})^{\mathbf{k}_b} e^{i\mathbf{x}\cdot\mathbf{R}} \qquad (5.6)\\ &\sum_{m_a=0}^{\lfloor n_a/2\rfloor} \sum_{m_b=0}^{\lfloor n_b/2\rfloor} \sum_{\mathbf{m}=(0,0,0)}^{\lfloor \mathbf{M}/2\rfloor} D_{m_a}^{n_a} D_{m_b}^{n_b} D_{\mathbf{m}}^{\mathbf{M}} (2\alpha w_1)^{n_a-2m_a}(2w_1)^{m_a}(2\beta w)^{n_b-2m_b} \\ &(2w)^{m_b}[2(\mathbf{a}+\mathbf{b})ww_1]^{\mathbf{M}-2\mathbf{m}}(2ww_1)^{\mathbf{m}\cdot\mathbf{1}} \frac{e^{-tR}}{2^N t^{2N+1}} \sum_{j=0}^{N} A_j^N t^j \,\mathrm{d}w\,, \end{aligned}$$

where the notation

$$\begin{aligned} t &= \sqrt{y}, \qquad w_1 = 1-w, \\ \mathbf{M} &= \mathbf{l}_a+\mathbf{l}_b-\mathbf{k}_a-\mathbf{k}_b, \qquad N = n_a+n_b+(\mathbf{M}-\mathbf{m})\cdot\mathbf{1}, \\ \mathbf{v}^{\mathbf{m}} &= (v_x,v_y,v_z)^{(m_x,m_y,m_z)} = v_x^{m_x}v_y^{m_y}v_z^{m_z}, \\ \lfloor\mathbf{M}\rfloor &= (\lfloor M_x\rfloor, \lfloor M_y\rfloor, \lfloor M_z\rfloor), \\ \binom{\mathbf{k}}{\mathbf{m}} &= \binom{(k_x,k_y,k_z)}{(m_x,m_y,m_z)} = \binom{k_x}{m_x}\binom{k_y}{m_y}\binom{k_z}{m_z}, \\ D_{\mathbf{m}}^{\mathbf{k}} &= D_{m_x}^{k_x}D_{m_y}^{k_y}D_{m_z}^{k_z}, \\ \sum_{\mathbf{k}=\mathbf{j}}^{\mathbf{n}} &= \sum_{\mathbf{k}=(j_x,j_y,j_z)}^{n_x,n_y,n_z} = \sum_{k_x=j_x}^{n_x}\sum_{k_y=j_y}^{n_y}\sum_{k_z=j_z}^{n_z} \end{aligned}$$

is used ($\lfloor n/2\rfloor$ denotes truncated integer part of $n/2$, i.e. the greatest integer $\le n/2$) and where the coefficients D_j^n and A_j^n are defined by the recurrence relations

$$D_0^1 = 1,$$

$$D_0^n = 1, \quad D_j^n = (n+1-2j)D_{j-1}^{n-1} + D_j^{n-1}, \quad n \geq 2,\ j = 1, \ldots, \lfloor n/2 \rfloor, \tag{5.7}$$
$$A_0^1 = -1, \quad A_1^1 = -R,$$
$$A_0^n = A_0^{n-1}(-2n+1), \quad A_j^n = A_j^{n-1}(j-2n+1) - RA_{j-1}^{n-1}, \quad ,n \geq 2,\ j = 0, \ldots, n.$$

The closed form formula (5.6) contains a 12-tuple summation, however, in any particular case most of the sums reduce to a single term. Keeping only the variables w, w_1, t which depend on the integration variable w the exchange integral (5.1) can be, by using (5.6), written in the closed form as

$$I(n_1, l_1, m_1, n_2, l_2, m_2) = \int_0^1 e^{-tR} \sum_{j=0}^{j_{max}} \sum_{k=0}^{k_{max}} \sum_{l=l_{min}}^{l_{max}} C_{jkl} w^j w_1^k t^l \, \mathrm{d}w. \tag{5.8}$$

To represent the wavefunctions (5.3) we use a floating-point array for coefficients C_j, four integer arrays for degrees $n_j, l_{xj}, l_{yj}, l_{zj}$ and of course a number of terms in the sum (α is represented by a special way by other physical quantities). The polynomial in w, w_1, t in the resulting formula (5.8) is represented by the three-dimensional floating-point array storing the coefficients C_{jkl} and by the degree bounds $j_{max}, k_{max}, l_{min}, l_{max}$.

The symbolic-numeric implementation includes subalgorithms for calculating the wavefunctions (5.3) and several stages for calculation of the coefficients C_{jkl} and degree bounds from (5.8) based on (5.6), (5.7) and expressing (5.1) after substituting (5.3) as the linear combination of (5.4).

5.3. VERIFICATION AND CODE GENERATION

The symbolic-numeric implementation, based mainly on (5.6), has been compared with the symbolic implementation, based mainly on (5.4). For the quantum numbers $0 \leq n_1 \leq 4, 0 \leq n_2 \leq n_1, 0 \leq l_1 < n_1, 0 \leq l_2 \leq \min(n_2 - 1, l_1), -l_1 \leq m_1 \leq l_1, -l_2 \leq m_2 \leq \min(l_2, m_1)$ (formulas are symmetric in n_1, n_2, in l_1, l_2 and in m_1, m_2, restrictions on l_i in terms of n_i and on m_i in terms of l_i are physical limitations on quantum numbers) we have obtained the same resulting formulas (5.8) where for the checking α, β and components of $\mathbf{a}, \mathbf{b}, \mathbf{R}$ have remained as parameters. Note that here these variables except $\mathbf{R}$ have to remain as parameters because the symbolic implementation performs derivatives with respect to them, while in the symbolic-numeric implementation these variables can already have numerical values from the beginning of the calculation. Verification has also been done for a number of randomly chosen quantum numbers with $n_1 > 4$ or $n_2 > 4$. The typical quantum numbers used in applications are small, usually the greatest one is around 4, so our verification test has covered most of the relevant region of quantum numbers. Again as in the previous section on matrix elements the whole algorithm is composed of several subalgorithms all of which have been independently verified up to degree 20 of polynomials.

To show the complexity of the symbolic processing involved in this problem we present here few numbers. The calculation of $I(3, 2, 1, 3, 1, -1)$ by the symbolic implementation took 25 s with the resulting formula which include the sum of 799 terms and occupy 400 lines in a dense, machine readable format. The whole verification described above took almost 2 hours of CPU time on a recent workstation.

From the same source file which defines the symbolic-numeric implementation we have generated the FORTRAN source file implementing the symbolic-numeric implementation. This code has two top level routines, the first one calculates for given quantum numbers $n_1, l_1, m_1, n_2, l_2, m_2$ and vectors $\mathbf{a}, \mathbf{b}, \mathbf{R}$ the value of the coefficients C_{jkl} and the degree bounds from (5.8), while the second one only calculates for given w (w_1 and t are functions of w) the numerical value of the integrated function from (5.8) in this point and is used by the numerical integrator to calculate the integral in (5.8).

6. On the Applications of the Generated Code

Both the codes generated for the described examples have been embedded in a broader application test program, evaluating sets of the relevant matrix elements. The less complex case, described in Section 4, has been combined with other mathematical objects (spherical harmonics and Clebsch–Gordan coefficients) in a straightforward manner.

The more complex case in section 5 is discussed in more detail. To keep the complexity low we described above only the development for overlap exchange integrals (5.1) however, we have actually developed a more general case which also includes two potential exchange integrals which differ from (5.1) by including $1/r_j, j = 1, 2$ in the integral. The performance of the generated code for all three types of exchange integrals has been compared with the currently widely used code of J.P. Hansen and collaborators (Hansen, 1990; Hansen and Dubois, 1992; Nielsen *et al.* , 1990). Although the numerical procedures differ at many points, agreement better than six significant digits has been obtained. The comparison has verified both the correctness of our method and that of Hansen's code. The calculational speed was in some cases higher, in other cases lower. This depends on the vectorization of the calculations (the symbolic evaluations in the code of Hansen *et al.* are structured differently). The evaluation of the matrix elements of section 4 has also been compared with the existing routines used in the codes of Hansen *et al.*

The method will presented here will thus be valuable in future revisions and applications of this type of atomic-collision code. To be more specific, the new applications might include work with several active electrons, special regions of quantum numbers in specialized versions of the codes, etc. It should be mentioned that the quantities whose evaluation is discussed here, are usually precalculated in a matrix typically $30 \times 30 \times 100$ for calculations which are typically repeated 50 to 100 times for a given collision. In a particular study some 20 to 100 model collisions might be investigated. The code which uses the symbolic-numeric procedures is, in principle, general, so that once debugged, its compiled version can be used for all such production calculations.

The routines developed by Hansen and collaborators have been tested in several periods during several years by two to four workers (including in several short periods also one of the authors, L.K.), and every extension of functionality required a new shorter or longer debugging period. In contrast to this, the coding work reported here has taken several weeks, including the development of the techniques. Furthermore, the existing encoding of the formulae can be used to generate new versions of the code with special features in the future.

7. Conclusion

We have presented a new methodology for constructing verified symbolic-numeric algorithms manipulating algebraic expressions of a special kind in numerical programming languages. The algorithms are implemented in a general CAS, here REDUCE, and verified by comparison with algorithms contained inside the system. The code used for verification is the same as that used for generating the numerical code performing the required manipulations so that the generated numerical code is also verified.

The methodology is discussed in full detail for an elementary case and then applied to the design of an algorithm for calculating the matrix elements of the Coulomb interaction of two bound hydrogenic states and the exchange integrals in the impact-parameter treatment of heavy-particle collisions.

Note that the method is not based on proving the correctness of symbolic-numeric algorithms. In all presented cases the algorithms has to be valid for all natural numbers (actually over direct product of several ones) so one might think about the possibility of constructing an automatic prover based on mathematical induction. However for real applications proving would not be an easy task, see e.g. formulas (5.5) and (5.6) which should be proven to be equal for all $n_a, \mathbf{l}_a, n_b, \mathbf{l}_b$ (i.e. over N^8) in the application described in Section 5.

Finally, we mention another aspect of the presented methodology. At present, the theoretical treatment of collisions with so called Rydberg atoms (e.g. Lundsgaard *et al.* (1995)) which have extremely high quantum numbers ($n \approx 30$) is of great interest. While the presented methods cannot be used directly, they will be helpful in developing new approximate methods to approach these problems.

Acknowledgements

We would like to thank J.P. Hansen, A. Dubois and S.E. Nielsen for valuable discussions and interest in this work and to the referees for valuable suggestions concerning presentation of this work.

This work has been supported in part by the Czech Grant Agency grant 201/94/1209.

Appendix A. New REDUCE Switch `genproc`

The new REDUCE switch `genproc` which allows us to use the same REDUCE source file `integ.pro` with symbolic-numeric implementation for verification and code generation has been implemented. The switch redefines the parsing routine for the REDUCE command `procedure`. If the switch is `off` then the `procedure` command works by the standard way, i.e. defines the REDUCE procedure, only the `literal` and `declare` declarations are removed from within the `begin-end` block inside the body of the `procedure` command. If the switch is `on` then the procedure is translated by GENTRAN (Gates, 1986) and not defined in REDUCE, i.e. REDUCE performs the same action as if the `procedure` command were to be preceded by the `gentran` command (actually as if the word `procedure` were to be replaced by the string `gentran procedure`) in the input. Further if the switch is `on` the `scalar, integer, real` and `operator` declarations are removed from within the `begin end` block inside the body of the `procedure` command.

The file `genproc.red` contains the implementation of the switch `genproc`. The special actions of this switch are required by the need to use really the same file `integ.pro` for the verification and code generation stages of the method described in this paper so that no intermediate file editing between this stages is necessary. One stage needs one type of declarations while the other one needs another declarations. The source files `integ.pro` contains both declarations and this switch takes care of removing the unnecessary declarations in each of the two stages.

The same approach as described above for the simple example from Section 3 with the source file `integ.pro` has been used in the two applications presented in Sections 4 and 5 with appropriate symbolic-numeric source files. All the files are available on request from `ladi@hpatom.fi.uib.no` or `liska@siduri.fjfi.cvut.cz`.

References

Cook, G. O. J. (1990). ALPAL: a program to generate physics simulation codes from natural descriptions. *Int. J. Mod. Phys. C*, **1**(1), 1–51.

Dewar, M. C., Richardson, M. G. (1990). Reconciling symbolic and numeric computation in a practical setting. In A. Miola (ed.), *Design and Implementation of Symbolic Computation Systems, DISCO'90, Lecture Notes in Computer Science*, **429**, pp. 195–204, Springer: Berlin.

Gates, B. L. (1986). A numerical code generation facility for REDUCE. In B. W. Char (ed.), *SYMSAC '86*, pp. 94–99, Waterloo, ACM.

Hansen, J. (1990). General subroutines for the calculation of atomic and molecular two-centre integrals. *Comput. Phys. Comm.*, **58**, 217–221.

Hansen, J., Dubois, A. (1992). Procedures for analytical and numerical calculation of coulombic one- and two-centre integrals. *Comput. Phys. Comm.*, **67**, 456–464.

Hearn, A. C. (1995). REDUCE user's manual, version 3.6. Technical Report RAND Publication CP 78 (Rev. 7/95), RAND, Santa Monica.

Jenks, R., Sutor, R. (1992). *AXIOM, the Scientific Computation System* . Springer, New York.

Kant, E. (1993). Synthesis of mathematical-modeling software. *IEEE Software*, **10**(3), 30–41.

Kocbach, L., Liska, R. (1994). Closed form formula for the exchange integrals in the impact-parameter treatment of heavy-particle collisions. *J. Phys. B: At. Mol. Opt. Phys.*, **27**, L619–L624.

Lundsgaard, M., Chen, Z., Lin, C., Toshima, N. (1995). Electron capture from circular Rydberg states. *Phys. Rev.* A, **51**, 1347–1350.

McDowell, M., Coleman, J. (1970). *Introduction to the Theory of Ion–Atom Collisions*. North-Holland, Amsterdam.

Nielsen, S., Hansen, J., Dubois, A. (1990). Propensity rules for orientation in singly-charged ion-atom collisions. *J. Phys. B: Atom. Molec. Phys.*, **23**, 2595–2612.

Rostaing, N., Dalmas, S., Galligo, A. (1993). Automatic differentiation in odyssée. *Tellus*, **45A**, 558–568.

Shakeshaft, R. (1975). A note on the exchange integrals in the impact-parameter treatment of heavy-particle collisions. *J. Phys. B: Atom. Molec. Phys.*, **8**, L134–136.

Steinberg, S., Roache, P. J. (1985). Symbolic manipulation and computational fluid dynamics. *J. Comput. Phys.*, **57**, 251–284.

Wang, P. S. (1986). FINGER: A symbolic system for automatic generation of numerical programs in finite element analysis. *J. Symbolic Computations*, **2**(3), 305–316.

Watt, S. M., Broadbery, P., Dooley, S., Iglio, P., Morrison, S., Steinbach, J., Sutor, R. (1994). *AXIOM Library Compiler, Users Guide* . NAG, Oxford.

Zippel, R. (1992). Symbolic/numeric techniques in modeling and simulation. In Donald, B., Kapur, D., Mundy, J., editors, *Symbolic and Numerical Computation in Artificial Intelligence* . Academic Press.

Originally received 1 September 1994
Accepted 12 February 1997