Computer Algebra Systems as mathematical optimizing compilers

Aldo Dall’Osso

9 Avenue de Parthenay, 92400 Courbevoie, France

Abstract

The role of computer algebra systems (CAS) is not limited to analyze and solve mathematical and physical problems. They have also been used as tools in the development process of computer programs, starting from the specification and ending with the coding and testing phases. In this way one can exploit their powerful mathematical capacity during the development phases and, by the other way, take advantage of the speed performance of languages as FORTRAN or C in the implementation. Among the mathematical features of CAS there are transformations allowing to optimize the final code instructions. In this paper we show some kind of optimizations that can be done on new or existing algorithms, by extending some techniques that compilers apply currently to optimize the machine code. The results show that the CPU time taken by the optimized code is reduced by a factor that can reach 5. The optimizations are performed with a package built on a well known CAS: Mathematica.

Keywords:
Computer algebra systems, Optimizing compilers, Problem-solving environments, Scientific software development, Code synthesis

1. Introduction

Program optimization is an important feature in software development but it is tedious and not all the optimization possibilities are looked for by the developer. Optimization touches the program development at different stages: the main algorithm definition, the general design of the program, the detailed design of each implemented function. Subprograms at the bottom of the calling tree are usually the most time consuming and optimization is unavoidable for them. But optimize bottom level programs means to introduce architecture depending programming idioms that can compromise readability and portability. For this reason bottom level

Email address: aldo.dallosso@free.fr (Aldo Dall’Osso).

optimization is usually left to compilers. But compilers have not sufficient mathematical knowledge that could permit them to scan all the optimization possibilities (see Section 3.1 for a discussion on this subject). To meet at the same time the requirements of readability, portability and automatization of the optimization process it would be preferable to work with a high level language.

With the advent of code synthesis systems (CSS) and problem-solving environments (PSE), symbolic calculation techniques have been used to automatically generate optimized code. Some of them incorporate their own symbolic analyzer to do the task, as Cdalel [6,7]. This last has been used to develop the weather forecast system Hirlam [4]. Other PSEs take advantage of existing computer algebra systems (CAS), as it is the case of SciNapse [1,2] built on the basis of the well known Mathematica [18]. With CSS and PSE (that in general contains CSS to transform the user directives in executable code) scientific computer codes can be developed through a natural user’s interface using a high level specification language. In general the optimization actions performed by CSSs are those that would have been done by a developer if he programmed manually an algorithm. CSSs spread from those that automatize completely the development process, to those that let the developer to define all of the algorithm steps and give only a mathematical and functional support. This last kind of CSS gives the developer the responsibility to decide where to optimize and what kind of optimization must be applied. We show in the next sections how a CAS can be used as a tool to perform optimizations in a modular way. In order to illustrate the context in which our optimizations operate, we draw the major lines of the software development aided by a CAS (Section 2). Then we show some kind of optimization operations (Section 3). In Section 4 are discussed other possible applications. We conclude in Section 5.

2. Specification language

Any CSS works on a high level specification language. Through it the developer describes the problem with complete abstraction from the coding aspect. It is the CSS itself that takes care of the following phases in the development process, just following some necessary user’s directive. Most CSSs translate the user’s specification in an intermediate language that is further processed by the symbolic analyzer to perform simplifications in order to put it in an optimized form, taking into account the architecture of the target computer and operating system. The translation in the target language is the last task.

In our approach the specification language is the CAS language itself, Mathematica, enriched by specific syntax forms [5]. A top-bottom approach is used in the development process. The developer specifies his initial problem as he had to describe it to a colleague. But differently from PSEs, all the aspects of the problem are defined by the developer. No methodological schemes are proposed by the CAS. The developer specifies how decompose the problem, defines the operations sequence in the
final algorithm, decides the correspondence between the mathematical symbols and the symbols appearing in the code. This aspect might not be viewed as a limitation of CASs with respect to PSEs. These last are devoted to well defined disciplines and for specific problems a development method based on user knowledge is preferable. It is the developer itself that will define new functional operators when he will see that a transformation sequence appears frequently in his development activities. For example, if the problems frequently solved are based on systems of equations with tridiagonal matrix, he will write a module collecting from the symbolic equations the diagonal elements of the matrix stored in three vectors. In this way the developer adds his experience to the CAS, building a collection of functional operators: a package. Thanks to the natural behavior of CASs languages, implementing new functional operators is as simple as explaining to some people how to determine the greatest common divisor of two numbers. The package MathCompile, used all along this paper, has been developed with this principle on the basis of Mathematica. It is loaded by the command:

\[
\text{In[1]:=} \text{\textless\textless MathCompile.m}
\]

Loading MathCompile enables extending Mathematica language with new functions. By invoking them in a Notebook (see below) the developer transforms the equations of his problem in order to reduce it progressively in the final code form.

The developing process of scientific programs can be summarized in the following tasks: define the problem in equations, make the necessary approximations if any, solve symbolically the problem, simplify the solution, organize the relationships obtained from the previous solution in the final algorithm, translate it in the target language. The optimizing process corresponds to the simplification phase. The first phases use the CAS language as a model specification language. Their purpose is producing the statements of the algorithm specification language that is automatically optimized and coded in the target language. Model specification language sentences are all applications of the CAS functional operators to solve problems or to make all kind of transformation, as symbolic solvers or simplifiers. Algorithm specification language statements are in general relationships written in transformation rule form as: \(x \to a\) (translated by assignment statements in the target language), or control constructs as If, Do, etc., or external subroutines calls. In our developing approach, model and algorithm specification statements are assigned to CAS symbols as Eq[1], Eq[2], etc. This allows to specify easily the information flow from the first equations of the model to the final instructions of the code.

To show how to specify an algorithm to be implemented in FORTRAN with a CAS, we take a simple example: a sub-program performing a parabolic interpolation between 3 points defined by \(\{x_i, y_i\}\) for \((i=1,2,3)\). In the following discussion we follow a Notebook structure\(^1\) where the user’s commands are preceded by the prompt \texttt{In[...]} and each CAS’s answer is preceded by the prompt \texttt{Out[...]}.

In writing the formulae we use the editing functions of Mathematica user’s interface

\(^1\) This article is a Mathematica Notebook.
allowing introducing subscripts and superscripts. This formalism has the advantage to enhance the readability of the specification.

The interpolation function we want to code is expressed by:

\[ \text{In[2]} := \quad \text{Eq[I]} := \]
\[ y_i = a_0 + a_1 x_i + a_2 x_i^2 \]

and the interpolation equation is \text{Eq[I]} applied to the 3 points \((i=1,2,3)\):

\[ \text{In[3]} := \quad \text{Eq[II]} = \]
\[ \text{Eq[I]} \quad \text{"for"} \{i, 1, 3\} \]
\[ \text{Out[3]} = \{y_1 = a_0 + a_1 x_1 + a_2 x_1^2, y_2 = a_0 + a_1 x_2 + a_2 x_2^2, y_3 = a_0 + a_1 x_3 + a_2 x_3^2\} \]

The solution of \text{Eq[II]} provides the coefficients \(a_i (i=0,1,2)\). It is obtained by the following specification sentence:

\[ \text{In[4]} := \quad \text{Eq[III]} = \text{Solve[Eq[II], \{a_0, a_1, a_2\}] \]
\[ \text{Out[4]} = \left\{ \begin{array}{l}
a_0 \rightarrow -\frac{x_2^2 x_3 y_1 + x_2 x_3^2 y_1 + x_1^2 x_3 y_2 - x_1 x_3^2 y_2 - x_1^2 x_2 y_3 + x_1 x_2^2 y_3}{(x_1 - x_2) (x_1 x_2 - x_1 x_3 - x_2 x_3 + x_3^2)}, \\
a_1 \rightarrow -\frac{x_2^2 y_1 - x_2^2 y_1 - x_1^2 y_2 + x_2 x_3^2 y_2 + x_1^2 y_3 - x_2^2 y_3}{(x_1 - x_2) (x_1 x_2 - x_1 x_3 - x_2 x_3 + x_3^2)}, \\
a_2 \rightarrow -\frac{(-x_1^2 + x_3^2) (y_1 - y_3) + (-x_1 + x_3) (y_1 - y_3)}{-(-x_1^2 + x_3^2) (-x_1 + x_3) + (-x_1 + x_3) (-x_1^2 + x_3^2)} \end{array} \right\} \]

Using the model specification language we have obtained 3 algorithm specification statements. A simplification followed by a common subexpression elimination is needed. The simplification is done by the native command \text{Simplify}. The common subexpression elimination is performed by the MathCompile command \text{CollectExpressions}.

\[ \text{In[5]} := \quad \text{Eq[IV]} = \text{CollectExpressions[Simplify[Eq[III]], q \_]} \]
\[ \text{Out[5]} = \left\{ \begin{array}{l}
q_1 \rightarrow x_1^2, \quad q_2 \rightarrow x_2^2, \quad q_3 \rightarrow x_3^2, \quad q_4 \rightarrow x_1 - x_3, \quad q_5 \rightarrow \frac{1}{q_3 (x_1 - x_2) (x_2 - x_3)}, \\
a_0 \rightarrow q_5 (-q_4 x_1 x_3 y_2 + x_2 (-q_2 y_1 + q_1 y_3) + q_3 (x_3 y_1 - x_1 y_3)), \\
a_1 \rightarrow q_5 (q_2 (y_1 - y_2) + q_1 (y_2 - y_3) + q_3 (-y_1 + y_3)), \\
a_2 \rightarrow q_5 (x_3 (-y_1 + y_2) + x_2 (y_1 - y_3) + x_1 (-y_2 + y_3)) \end{array} \right\} \]

The 3 statements have been transformed in 8 statements whose CPU performance is higher, having reduced the number of floating point operations. To complete the algorithm we must transform the interpolation function defined in \text{Eq[I]} from \text{Equal form} to \text{Rule form}, change the symbols' names and apply the \text{HornerRule} in order to reduce the number of multiplications:

\[ \text{In[6]} := \quad \text{Eq[V]} = \text{ToRules[Eq[I]]} /. \{x_i \rightarrow x_v, \ y_i \rightarrow y_v\} /. \text{HornerRule}[x_v] \]
\texttt{Out[6]} = \{y \rightarrow a_0 + x y (a_1 + a_2 x y)\}

The complete algorithm is obtained assembling all of these statements:

\texttt{In[7]}: = \textbf{ParabolicInterpolation} := \{\text{Eq[IV]}, \text{Eq[V]}\}

The final interpolating sub-program is obtained by invocation of the MathCompile command \texttt{FortranProgramUnit}, where the developer defines the program unit type ("S": subroutine), the name, the input, output and local data with the FORTRAN type specification, the sequence of statements and the correspondence table between symbols used in the specification and symbols to be used in the coding phase. Although this could be surprising in a specification language, giving the user the possibility to choose the symbols names, even for temporary variables (in this case \texttt{q}, whose dimension is deduced by taking the higher subscript value appearing in \texttt{Eq[IV]}, i.e. 5), allows to analyze easily the code during execution with debug tools, in case of executions problems (overflows, numerical instabilities).

\texttt{In[8]}: = \texttt{FortranProgramUnit[S, parint,}

\begin{verbatim}
    \{(x, real, \{3\}), \{y, real, \{3\}\}, \{x_v, real\},\n    \{(y_v, real\},\n    \{\{a, real, \{0, 2\}\}, \{q, real, \{5\}\}\},
    \textbf{ParabolicInterpolation},
    \{x_v \rightarrow \texttt{xv}, y_v \rightarrow \texttt{yv}\}\}
\end{verbatim}

The notebook section starting from \texttt{In[2]} and ending with \texttt{In[8]} can be seen as the true source of subroutine \texttt{parint}, a kind of self documented source code. The FORTRAN form can be considered as a temporary file, which could be removed after compilation, because it can be generated at any time running the Notebook with Mathematica.

Package MathCompile enables translating either in FORTRAN 77 or 90 but here we use FORTRAN 77 as target language. The coded subroutine is shown in Fig. 1.

In this example, the key relationships have been generated by the CAS symbolic solver. In general the developer can define itself the relationships he wants to code. He can use the algorithm specification language to write a complete algorithm and the CAS is useful to optimize it and to code it in the target language. Writing the code in this high level language enables introducing in the code the optimization techniques described in the following sections.
Subroutine parint(x,y,xv,yv)
implicit none
real x(3)
real y(3)
real xv
real yv
real a(0:2)
real q(5)
q(1) = x(1) ** 2
q(2) = x(3) ** 2
q(3) = x(2) ** 2
q(4) = x(1) - x(3)
q(5) = 1 / (q(4) * (x(1) - x(2)) * (x(2) - x(3)))
\( a(0) = q(5) * (-q(4) * x(1) * x(3) * y(2)) + x(2) * (-q(2) * y(1)) + q(1)) \)
\( a(1) = q(5) * (q(2) * y(1) - y(2)) + q(1) * (y(2) - y(3)) + q(3)) \)
\( a(2) = q(5) * (q(3) * y(1) - y(2)) + q(1) * y(2) + q(3)) \)
\( yv = a(0) + xv * (a(1) + xv * a(2)) \)
end

Fig. 1. FORTRAN77 subroutine resulting from the translation of the interpolating function specification.

3. Optimization techniques

In this section we show some kinds of optimization techniques that can be applied on programming sequences written in algorithm specification language. Their main aim is to avoid loop overhead by code replication. The resulting code is bigger in size but faster when executed. From the readability point of view it must be emphasized that the result of the optimization is to be considered as an intermediate code. The true source code is the model specification. In this way we keep the program clear. In our approach each optimization is performed only on developer directives. This way of proceeding is equivalent to the choice of the optimization level, accessible on compilers. If, by one hand, this approach is more constraining because it demands fine decisions to the developer, by the other hand, it permits to avoid conflicting optimizing strategies.

3.1 Basic optimizing operators

After the execution of mathematical transformations it is always necessary to perform other final transformations in order to reduce the complexity of the results. If these post transformations are useful but not necessary when the main mathematical transformations are performed by hand, they become unavoidable if they are performed by a CAS. In fact, with the support of a CAS, we can perform transformations that we would never dare to perform without it. Complex transformations lead to very complex results that the functional operators of the CAS can simplify. Between the simplification operations we can cite factorizations, applications of mathematical rules, etc. The built in simplification command of the majority of the existing CASs executes this kind of operations. But to reduce drastically the complexity of a big expression, operations as common subexpression elimination
(CSE) are of great efficiency. This kind of optimization is today executed by many compilers but CAS can do it on expressions of bigger size. Moreover CSE executed by a CAS is more efficient than the one executed by compilers. The expression processing done by a CAS is not a simple pattern matching replacement. In doing the common subexpression search it takes into account mathematical rules. Hence, if among a series of expressions it finds subexpressions as \( x^2, x^3, \frac{1}{x^2} \), it recognizes that they can be replaced by \( a, x, \frac{1}{x} \), where \( a \) is a temporary variable set to \( x^2 \). Other kinds of optimizations which compilers perform, as loop unfolding, loop unrolling, Do-If blocks interchange, inlining, code replication, can be specified by the developer through a CAS.

### 3.2 Optimization through operations expansion

Programming languages offer many varieties of iteration blocks. If this kind of control construct is the major feature offered by computers, its use is not always the best choice in algorithm implementation. Optimizing compilers are aware of this kind of problem and many strategies are built to avoid that control constructs penalize the computation time. The most common between them is loop unrolling or loop flattening. With CAS this is very easy to do. Let us take the following matrix operation:

\[
\text{In}[9] := \text{op} = \quad g \rightarrow a \cdot b; \]

where \( a, b, \) and \( q \) are rank 4 matrices. The most natural way to program this operation in FORTRAN is the following:

```fortran
do i = 1, 4
   do j = 1, 4
      q(i, j) = 0
      do k = 1, 4
         q(i, j) = q(i, j) + a(i, k) * b(k, j)
      end do
   end do
end do
```

FORTRAN 90 allows writing it in more concise way, thanks to the functional operator `matmul`:

```
g = matmul(a, b)
```

This programming style can be considered satisfactory when the rank of matrices is not known but it is data of the problem. It is not optimal when, as in our case, the rank is known. In this case it is better to write explicitly all the elementary operations composing the matrix product. For this purpose CASs are very useful.

The expansion of the above matrix operation `op` can be done in applying the rule `SubscriptedMatrix` of package MathCompile permitting to explicit a matrix
through its elements, and threading the rule symbol $\rightarrow$ over the resulting matrix elements:

$$In[10]:= \text{explop} = \text{op/.SubscriptedMatrix}\{\{a, b, g\}, 4, 4\}/.\text{aRule} :> \text{Thread}\{a\}$$

$$Out[10]= \{\{g_{1,1} \rightarrow a_{1,1} b_{1,1} + a_{1,2} b_{2,1} + a_{1,3} b_{3,1} + a_{1,4} b_{4,1},
\quad g_{1,2} \rightarrow a_{1,1} b_{1,2} + a_{1,2} b_{2,2} + a_{1,3} b_{3,2} + a_{1,4} b_{4,2},
\quad g_{1,3} \rightarrow a_{1,1} b_{1,3} + a_{1,2} b_{2,3} + a_{1,3} b_{3,3} + a_{1,4} b_{4,3},
\quad g_{1,4} \rightarrow a_{1,1} b_{1,4} + a_{1,2} b_{2,4} + a_{1,3} b_{3,4} + a_{1,4} b_{4,4}\},
\quad \ldots ,
\quad \{g_{4,1} \rightarrow a_{4,1} b_{1,1} + a_{4,2} b_{2,1} + a_{4,3} b_{3,1} + a_{4,4} b_{4,1},
\quad g_{4,2} \rightarrow a_{4,1} b_{1,2} + a_{4,2} b_{2,2} + a_{4,3} b_{3,2} + a_{4,4} b_{4,2},
\quad g_{4,3} \rightarrow a_{4,1} b_{1,3} + a_{4,2} b_{2,3} + a_{4,3} b_{3,3} + a_{4,4} b_{4,3},
\quad g_{4,4} \rightarrow a_{4,1} b_{1,4} + a_{4,2} b_{2,4} + a_{4,3} b_{3,4} + a_{4,4} b_{4,4}\}\}$$

This sequence of operations can be directly translated in FORTRAN via the functional operators of package MathCompile. Here is the translation command containing all the necessary information to declare the data.

$$In[11]:= \text{FortranProgramUnit}\{S, \text{mpexp4},
\quad \{\{a, \text{real}, \{4, 4\}\}, \{b, \text{real}, \{4, 4\}\}\},
\quad \{\{g, \text{real}, \{4, 4\}\}\},
\quad \{}, \text{explop}, \{a_{i\ldots j} \rightarrow a[i, j]\}\}$$

The automatically coded subroutine is shown in Fig. 2.

The subroutine obtained has a bigger size than the one corresponding to the conventional programming sequence like the one shown above, but its performance is higher. If we indicate with 100 the CPU time used by the loop programming, the time used by the expanded subroutine is about 13. The CPU time saving is reduced if we use an optimizing option of the compiler but it remains not negligible. The comparison between the CPU times for an Absoft FORTRAN compiler on a MacIntosh G3 233 MHz processor (with two optimisation levels) is shown in Table 1.
Subroutine mexp4(a, b, c)
    implicit none
    real a(4, 4)
    real b(4, 4)
    real c(4, 4)
    g(1, 1) = a(1, 1) * b(1, 1) + a(1, 2) * b(2, 1) + a(1, 3) * b(3, 1) + a(1, 4) * b(4, 1)
    g(1, 2) = a(1, 1) * b(1, 2) + a(1, 2) * b(2, 2) + a(1, 3) * b(3, 2) + a(1, 4) * b(4, 2)
    g(1, 3) = a(1, 1) * b(1, 3) + a(1, 2) * b(2, 3) + a(1, 3) * b(3, 3) + a(1, 4) * b(4, 3)
    g(1, 4) = a(1, 1) * b(1, 4) + a(1, 2) * b(2, 4) + a(1, 3) * b(3, 4) + a(1, 4) * b(4, 4)
    g(2, 1) = a(2, 1) * b(1, 1) + a(2, 2) * b(2, 1) + a(2, 3) * b(3, 1) + a(2, 4) * b(4, 1)
    g(2, 2) = a(2, 1) * b(1, 2) + a(2, 2) * b(2, 2) + a(2, 3) * b(3, 2) + a(2, 4) * b(4, 2)
    g(2, 3) = a(2, 1) * b(1, 3) + a(2, 2) * b(2, 3) + a(2, 3) * b(3, 3) + a(2, 4) * b(4, 3)
    g(2, 4) = a(2, 1) * b(1, 4) + a(2, 2) * b(2, 4) + a(2, 3) * b(3, 4) + a(2, 4) * b(4, 4)
    g(3, 1) = a(3, 1) * b(1, 1) + a(3, 2) * b(2, 1) + a(3, 3) * b(3, 1) + a(3, 4) * b(4, 1)
    g(3, 2) = a(3, 1) * b(1, 2) + a(3, 2) * b(2, 2) + a(3, 3) * b(3, 2) + a(3, 4) * b(4, 2)
    g(3, 3) = a(3, 1) * b(1, 3) + a(3, 2) * b(2, 3) + a(3, 3) * b(3, 3) + a(3, 4) * b(4, 3)
    g(3, 4) = a(3, 1) * b(1, 4) + a(3, 2) * b(2, 4) + a(3, 3) * b(3, 4) + a(3, 4) * b(4, 4)
    g(4, 1) = a(4, 1) * b(1, 1) + a(4, 2) * b(2, 1) + a(4, 3) * b(3, 1) + a(4, 4) * b(4, 1)
    g(4, 2) = a(4, 1) * b(1, 2) + a(4, 2) * b(2, 2) + a(4, 3) * b(3, 2) + a(4, 4) * b(4, 2)
    g(4, 3) = a(4, 1) * b(1, 3) + a(4, 2) * b(2, 3) + a(4, 3) * b(3, 3) + a(4, 4) * b(4, 3)
    g(4, 4) = a(4, 1) * b(1, 4) + a(4, 2) * b(2, 4) + a(4, 3) * b(3, 4) + a(4, 4) * b(4, 4)
end

Fig. 2. FORTRAN 77 subroutine resulting from the translation of the matrix multiplication expansion.

Table 1
Computation times (µs) of two rank 4 matrices multiplication with the classical loop and the expanded formulation, without and with compiler optimisation (-O3).

<table>
<thead>
<tr>
<th></th>
<th>Classic</th>
<th>Expanded</th>
</tr>
</thead>
<tbody>
<tr>
<td>No opt</td>
<td>11.5</td>
<td>1.5</td>
</tr>
<tr>
<td>O3</td>
<td>4.4</td>
<td>0.9</td>
</tr>
</tbody>
</table>

3.3 Optimization through algorithm expansion

In the previous section we have seen the benefit obtained by expanding an operation instead to program it with a loop. This principle can be extended to complex algorithms, when, as said before, the size of the data structures is fixed and not problem dependent. We illustrate our proposition with the inverse matrix computation problem.
Let us take a $4 \times 4$ matrix, that we build with the \texttt{SubscriptedMatrix} operator:

\begin{verbatim}
In[12]:= Eq[1] = SubscriptedMatrix[a, 4, 4];

In[13]:= MatrixForm[Eq[1]]
\end{verbatim}

\begin{verbatim}
Out[13]= a \rightarrow
\begin{pmatrix}
a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\
a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\
a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\
a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4}
\end{pmatrix}
\end{verbatim}

Mathematica allows obtaining the inverse of a matrix by application of the Cramer rule. This is done either in numeric or symbolic calculations. We obtain the symbolic expression of each element of the inverse matrix by the built in command \texttt{Inverse} and the functional operator \texttt{MatrixToRelation} of package MathCompile.

\begin{verbatim}
In[14]:= Eq[2] =
MatrixToRelation[Inverse[a/.Eq[1]], b]
\end{verbatim}

\begin{verbatim}
Out[14]= \{b_{1,1} \rightarrow \\
(-a_{2,4} a_{3,3} a_{4,2} + a_{2,3} a_{3,4} a_{4,2} + a_{2,4} a_{3,2} a_{4,3} - a_{2,2} a_{3,4} a_{4,3} - a_{2,3} a_{3,2} a_{4,4} + \\
a_{2,2} a_{3,3} a_{4,4})/(a_{1,4} a_{2,3} a_{3,2} a_{4,1} - a_{1,3} a_{2,4} a_{3,2} a_{4,1} - \\
a_{1,4} a_{2,2} a_{3,3} a_{4,1} + \ldots + a_{1,1} a_{2,2} a_{3,3} a_{4,4})
, \\
b_{1,2} \rightarrow (a_{1,4} a_{2,3} a_{3,4} a_{4,2} - a_{1,3} a_{2,4} a_{3,2} - a_{1,4} a_{3,2} a_{4,3} + a_{1,2} a_{3,4} a_{4,3} + \\
a_{1,3} a_{2,4} a_{3,2} a_{4,1} - a_{1,4} a_{2,2} a_{3,3} a_{4,1} + \ldots + a_{1,1} a_{2,2} a_{3,3} a_{4,4})/(a_{1,4} a_{2,3} a_{3,2} a_{4,1} - \\
a_{1,3} a_{2,4} a_{3,2} a_{4,1} - a_{1,4} a_{2,2} a_{3,3} a_{4,1} + \ldots + a_{1,1} a_{2,2} a_{3,3} a_{4,4})
, \\
\ldots , \\
b_{4,4} \rightarrow (-a_{1,3} a_{2,2} a_{3,1} + a_{1,2} a_{2,3} a_{3,1} + a_{1,3} a_{2,1} a_{3,2} - a_{1,1} a_{2,3} a_{3,2} - \\
a_{1,2} a_{2,1} a_{3,3} + a_{1,1} a_{2,2} a_{3,3})/(a_{1,4} a_{2,3} a_{3,2} a_{4,1} - \\
a_{1,3} a_{2,4} a_{3,2} a_{4,1} - a_{1,4} a_{2,2} a_{3,3} a_{4,1} + \ldots + a_{1,1} a_{2,2} a_{3,3} a_{4,4})
\}
\end{verbatim}

The full expression of the inverse matrix contains a lot of operations, many of which appear several times. We optimize the matrix inversion result of Eq[2] by a common subexpression elimination. We collect these subexpressions on the temporary array $q_i$, and eliminate them from the full expression by the command \texttt{CollectExpressions} of package MathCompile.

\begin{verbatim}
In[15]:= Eq[3] = CollectExpressions[Eq[2], q_i]
\end{verbatim}
\[ \text{Out[15]} = \{ q_1 \rightarrow a_{1,4} a_{2,3}, q_2 \rightarrow a_{3,2} a_{4,1}, q_3 \rightarrow a_{1,3} a_{2,4}, q_4 \rightarrow a_{1,4} a_{2,2}, \]
\[ q_5 \rightarrow a_{3,3} a_{4,1}, q_6 \rightarrow a_{1,2} a_{2,4}, q_7 \rightarrow a_{1,3} a_{2,2}, q_8 \rightarrow a_{3,4} a_{4,1}, \]
\[ \quad \quad \ldots, \]
\[ q_{21} \rightarrow a_{1,1} a_{2,2}, q_{22} \rightarrow a_{3,1} a_{4,4}, q_{23} \rightarrow a_{3,2} a_{4,4}, q_{24} \rightarrow a_{3,3} a_{4,4}, \]
\[ q_{25} \rightarrow 1/(q_1 q_2 - q_2 q_3 - q_3 q_5 + q_5 q_6 + q_7 q_8 - q_8 q_9 - q_1 q_{10} + q_3 q_{10} + q_{11} q_{12} - q_{12} q_{13} - q_{14} q_{15} + q_{15} q_{16} + q_{16} q_{17} - q_{17} q_9 - q_1 q_{10} + q_3 q_{10} + q_{11} q_{12} - q_{20} q_{21} - q_7 q_{22} + q_9 q_{22} + q_{14} q_{23} - q_{16} q_{23} - q_{19} q_{24} + q_{21} q_{24}), \]
\[ b_{1,1} \rightarrow q_{25} (-q_{20} a_{2,2} + q_{24} a_{2,2} + q_{15} a_{2,3} - q_{23} a_{2,3} - q_{12} a_{2,4} + q_{16} a_{2,4}), \]
\[ b_{1,2} \rightarrow q_{25} (-q_{20} a_{1,2} - q_{24} a_{1,2} - q_{15} a_{1,3} + q_{23} a_{1,3} + q_{12} a_{1,4} - q_{18} a_{1,4}), \]
\[ \quad \quad \ldots, \]
\[ b_{4,4} \rightarrow q_{25} (-q_{7} a_{3,1} + q_{9} a_{3,1} + q_{14} a_{3,2} - q_{16} a_{3,2} - q_{19} a_{3,3} + q_{21} a_{3,3}) \} \]

Symbol \( \Box \) means that \( q \) is a subscripted variable. To show more explicitly the technique of algorithm expansion we perform the same inversion by another algorithm: the Gauss-Jordan elimination (without pivoting). The algorithm, expressed in algorithm specification language, is the following:

\[ \text{In[16]} := \text{Eq[4]} := \{ \]
\[ \quad n \rightarrow \text{Length}[a], \]
\[ \quad u \rightarrow a, \]
\[ \quad b \rightarrow \text{IdentityMatrix}[n], \]
\[ \quad \text{Do}\left[\{p \rightarrow \frac{1}{u_{i,i}}, \right. \] \[ \quad \quad \text{Do}\left[\{u_{i,j} \rightarrow pu_{i,j}, \right. \] \[ \quad \quad \quad b_{i,j} \rightarrow pb_{i,j}\}, \{j, 1, n\}\}, \right. \] \[ \quad \quad \text{Do}\left[\{\text{If}[\{k \neq i\}, \{q \rightarrow u_{k,i}, \right. \] \[ \quad \quad \quad \text{Do}\left[\{u_{k,j} \rightarrow u_{k,j} - qu_{i,j}, \right. \] \[ \quad \quad \quad \quad b_{k,j} \rightarrow b_{k,j} - qb_{i,j}\}, \{j, 1, n\}\}\}, \{k, 1, n\}\}\}, \right. \] \[ \quad \quad \text{Do}\left[\{i, 1, n\}\}\], \right. \] \[ \] \]
The above operation sequence is not directly executable. Writing it with this formalism allows to delay the choice of the target language and we can perform optimization actions before the translation. We translate the algorithm in FORTRAN 77 as it is, only with the purpose to evaluate the CPU time performance and to compare it with the ones of the other implementations. In order to expand the Gauss-Jordan algorithm, as we have done with the Cramer method, we need a Mathematica function that implements it. We generate it by translating Eq[4] in procedural executable language, under the form of a Mathematica Module by use of the operator MathModule of package MathCompile, with which the developer specify in the following order: the module name, the input, the output, the local variables, the equations to be translated and the correspondence between the symbols of algorithmic specification and the symbols of the final module.

In[17]:= MathModule[GJInverse,
   
   {{a, Matrix, {n, n}}}, {{b, Matrix, {n, n}}},
   
   {{n, Integer}, {i, Integer},
   
   {j, Integer}, {k, Integer},
   
   {p, Real}, {q, Real}, {u, Matrix, {n, n}}},
   
   Eq[4], {a_{i \rightarrow} a[[i, j]], a_{i \rightarrow} a[[i]]}]}

The generated module GJInverse can be used to inverse numerically or symbolically matrices. In our context we use it symbolically in order to obtain the expansion of the Gauss-Jordan algorithm. As we have done for the Cramer algorithm in Eq[2], we use the functional operator MatrixToRelation to get the relation giving each element of the matrix.

In[18]:= << GJInverse.m

In[19]:= Eq[5] = MatrixToRelation[GJInverse[a/.Eq[1]], b]

Out[19]= \{b_{1,1} \rightarrow \frac{1}{a_{1,1}} + \ldots - \ldots

\left(\frac{a_{2,1}}{a_{1,1}} - \ldots + \frac{a_{3,1}}{a_{1,1}} + \ldots + a_{3,1}\right) (...) \right) (\ldots)

\left(\frac{a_{2,1} \cdot a_{2,1}}{a_{1,1}} - \left(\frac{a_{2,1}}{a_{1,1}} + \ldots + a_{3,1}\right) (...) - \ldots + a_{3,1}\right) + a_{3,1}

b_{3,1} \rightarrow \frac{1}{a_{3,1}} + \ldots - \ldots

\left(\frac{a_{3,1}}{a_{1,1}} - \frac{a_{3,1} \cdot a_{3,1}}{a_{1,1}} - \left(\frac{a_{3,1}}{a_{1,1}} + \ldots + a_{3,1}\right) (...) - \ldots + a_{3,1}\right) + a_{3,1}
The result is of a huge complexity. As we have done in the case of Eq[2], we proceed to a common subexpression elimination via the command CollectExpressions.

\[ In[20]:= \text{Eq}[6] = \text{CollectExpressions}[\text{Eq}[5], q_6] \]

\[ Out[20]= \begin{cases} q_1 \rightarrow \frac{1}{a_{1,1}}, & q_2 \rightarrow q_1 a_{1,2}, & q_3 \rightarrow -q_2 a_{2,1} + a_{2,2}, & q_4 \rightarrow q_1 a_{1,3}, \\
q_5 \rightarrow \frac{1}{q_3}, & q_6 \rightarrow -q_3 a_{2,1} + a_{2,3}, & q_7 \rightarrow -q_2 a_{3,1} + a_{3,2}, & q_8 \rightarrow q_1 a_{1,4}, \\
q_9 \rightarrow q_5 q_6, & q_{10} \rightarrow -q_2 a_{4,1} + a_{4,2}, & q_{11} \rightarrow -q_7 q_9 - q_4 a_{3,1} + a_{3,3}, \\
q_{12} \rightarrow q_5 (-q_8 a_{2,1} + a_{2,4}), & q_{13} \rightarrow q_1 a_{2,1}, & q_{14} \rightarrow q_5 q_7, & q_{15} \rightarrow \frac{1}{q_{11}}, \\
q_{16} \rightarrow -q_7 q_{12} - q_8 a_{3,1} + a_{3,4}, & q_{17} \rightarrow -q_9 q_{10} - q_4 a_{4,1} + a_{4,3}, \\
q_{18} \rightarrow q_4 - q_2 q_9, & q_{19} \rightarrow q_{13} q_{14} - q_1 a_{3,1}, & q_{20} \rightarrow q_{15} q_{16}, & q_{21} \rightarrow q_{15} q_{18}, \\
q_{22} \rightarrow q_5 q_{10}, & q_{23} \rightarrow q_{15} q_{19}, & q_{24} \rightarrow q_{15} q_{17}, & q_{25} \rightarrow q_8 - q_2 q_{12} - q_{18} q_{20}, \\
q_{26} \rightarrow -q_{10} q_{12} - q_{17} q_{20} - q_8 a_{4,1} + a_{4,4}, & q_{27} \rightarrow q_{12} - q_9 q_{20}, \\
q_{28} \rightarrow q_{26} (q_{13} q_{22} - q_{17} q_{23} - q_1 a_{4,1}), & q_{29} \rightarrow (-q_{22} + q_{14} q_{24}) q_{26}, & q_{30} \rightarrow q_{24} q_{26}, \\
b_{1,1} \rightarrow q_1 - q_{19} q_{21} - q_{25} q_{28} + \frac{q_3 a_{1,2} a_{2,1}}{a_{1,1}}, & b_{1,2} \rightarrow -q_2 q_5 + q_{14} q_{21} - q_{25} q_{29}, \\
b_{1,3} \rightarrow -q_{21} + q_{25} q_{30}, & b_{1,4} \rightarrow -q_{25} q_{26}, & b_{2,1} \rightarrow -q_5 q_{13} - q_9 q_{23} - q_{27} q_{28}, \\
b_{2,2} \rightarrow q_5 + \frac{q_6 q_7 q_{15}}{q_3^2} - q_{27} q_{29}, & b_{2,3} \rightarrow -q_9 q_{15} + q_{27} q_{30}, & b_{2,4} \rightarrow -q_{26} q_{27}, \\
b_{3,1} \rightarrow q_{23} - q_{20} q_{28}, & b_{3,2} \rightarrow -q_{14} q_{15} - q_{20} q_{29}, & b_{3,3} \rightarrow q_{15} + \frac{q_{16} q_{17} q_{26}}{q_{11}^2}, \\
b_{3,4} \rightarrow -q_{20} q_{26}, & b_{4,1} \rightarrow q_{28}, & b_{4,2} \rightarrow q_{29}, & b_{4,3} \rightarrow -q_{30}, & b_{4,4} \rightarrow q_{26} \end{cases} \]

Even if we have separated the expansion process into two phases (the generation of the module implementing the algorithm and its invocation to generate the expanded form) they can be joined in one producing the expanded algorithm from its specification.

We translate now Eq[3], Eq[4] and Eq[6] in FORTRAN 77 in order to obtain the subroutines giving the inverse of a $4 \times 4$ matrix with:

- the expanded Cramer solution,
- the conventional loop algorithm implementing the Gauss-Jordan elimination,
- the expanded algorithm implementing the Gauss-Jordan elimination.

For Eq[3] we have:
In[21]:- FortranProgramUnit[S, inv4Cr,

    {{a, real, {4, 4}}}, {{b, real, {4, 4}}},

    {{q, real, {25}}}, Eq[3], {}]

The automatically coded subroutine of the matrix inversion with the Cramer rule is shown in Fig. 3.

In translating Eq[4] to obtain the base algorithm of the Gauss Jordan elimination scheme in FORTRAN 77 we use the classical memory manage consisting in using the input matrix as storage array, i.e.: \( u \rightarrow a \). To do an equitable comparison we set the rank of the matrix to 4 and we use a specialized version of the subroutine setting \( b \) to an identity matrix (idmat 4).

In[22]:- FortranProgramUnit[S, inv4GJL,

    {{a, real, {4, 4}}},

    {{b, real, {4, 4}}}, {{p, real}, {q, real},

    {i, integer}, {j, integer}, {k, integer}},

    Drop[Eq[4], 2], {n \rightarrow 4, u \rightarrow a}]

The coded subroutine is shown in Fig. 4.

For Eq[6] we have:

In[23]:- FortranProgramUnit[S, inv4GJE,

    {{a, real, {4, 4}}}, {{b, real, {4, 4}}},

    {{q, real, {30}}}, Eq[6], {}]

The coded subroutine of the expanded Gauss Jordan algorithm is shown in Fig. 5.

We have tested these subroutines and compared their CPU times. If we assign the value 100 to the time taken by the classical Gauss-Jordan implementation, the expanded implementation takes 18 and the Cramer implementation takes 23. The CPU time saving of the expanded version with respect to the classical Gauss-Jordan algorithm, is due not only to the economy in loop overhead, but also in common subexpression elimination. In fact the expansion reveals all the repeated multiplications that are done in the loop. The comparison between the CPU times for an Absoft FORTRAN compiler on a MacIntosh G3 233 MHz processor (with two optimisation levels) is shown in Table 2. We can see that the expanded Gauss-Jordan algorithm is faster than the expanded Cramer, but this last has the advantage to do not have numerical instabilities, when Gauss-Jordan without pivoting has, if the matrix is ill conditioned. To implement the expanded Gauss-Jordan algorithm we must put the matrix in an order that depends on the physical problem we are solving.
Subroutine inv4Cr(a,b)
implicit none
real a(4,4)
real b(4,4)
real q(25)
q(1) = -a(1,4)*a(2,3)
q(2) = a(3,2)*a(4,1)
q(3) = a(1,3)*a(2,4)
q(4) = a(1,4)*a(2,2)
q(5) = a(3,3)*a(4,1)
q(6) = a(1,2)*a(2,4)
q(7) = a(1,3)*a(2,2)
q(8) = a(3,4)*a(4,4)
q(9) = a(1,2)*a(2,3)
q(10) = a(3,1)*a(4,2)
q(11) = a(1,4)*a(4,2)
q(12) = a(3,3)*a(4,2)
q(13) = a(1,1)*a(2,4)
q(14) = a(1,3)*a(2,1)
q(15) = a(3,4)*a(4,2)
q(16) = a(1,1)*a(2,3)
q(17) = a(3,1)*a(4,3)
q(18) = a(3,2)*a(4,3)
q(19) = a(1,2)*a(4,2)
q(20) = a(3,3)*a(4,3)
q(21) = a(1,1)*a(4,2)
q(22) = a(3,3)*a(4,4)
q(23) = a(3,2)*a(4,4)
q(24) = a(3,3)*a(4,4)

q(25) = 1/(q(1)*q(2) - q(3)*q(4) - q(5)*q(6) + q(7)*q(8) + q(9)*q(10) + q(11)*q(12) - q(13)*q(14) + q(15)*q(16) - q(17)*q(18) - q(19)*q(20) - q(21)*q(22) + q(23)*q(24))

b(1,1) = (-a(2,4)*q(12)) + a(2,3)*q(15) + a(2,4)*q(18) - a(2,2)*q(21)
b(1,2) = (-a(1,4)*q(12)) - a(1,3)*q(15) + a(1,4)*q(18) + a(1,2)*q(21)
b(1,3) = (-a(4,2)*q(12)) - a(4,3)*q(15) + a(4,3)*q(18) - a(4,4)*q(21)
b(1,4) = (-a(3,2)*q(12)) - a(3,3)*q(15) + a(3,3)*q(18) - a(3,4)*q(21)
b(2,1) = a(2,4)*q(5) + a(2,3)*q(8) + a(2,4)*q(17) + a(2,1)*q(20)
b(2,2) = (-a(1,4)*q(5)) + a(1,3)*q(8) + a(1,4)*q(17) - a(1,1)*q(20)
b(2,3) = a(4,2)*q(5) + a(4,3)*q(8) + a(4,3)*q(17) - a(4,4)*q(20)
b(2,4) = (-a(3,2)*q(5)) + a(3,3)*q(8) + a(3,3)*q(17) - a(3,4)*q(20)
b(3,1) = -a(2,4)*q(2) + a(2,2)*q(8) + a(2,4)*q(10) - a(2,1)*q(15)
b(3,2) = a(1,4)*q(2) - a(1,2)*q(8) - a(1,4)*q(10) + a(1,1)*q(15)
b(3,3) = a(4,2)*q(2) - a(4,2)*q(8) - a(4,4)*q(10) + a(4,1)*q(15)
b(3,4) = a(3,2)*q(2) - a(3,3)*q(8) - a(3,3)*q(10) + a(3,1)*q(15)
b(4,1) = -a(2,4)*q(7) + a(2,1)*q(24) - a(2,4)*q(25)
b(4,2) = (-a(1,4)*q(7)) - a(1,3)*q(24) - a(1,4)*q(25) + a(1,1)*q(24)
b(4,3) = a(4,2)*q(7) - a(4,3)*q(24) - a(4,3)*q(25) + a(4,1)*q(24)
b(4,4) = (-a(3,2)*q(7)) - a(3,3)*q(24) - a(3,3)*q(25) + a(3,1)*q(24)

end

Fig. 3. FORTRAN 77 subroutine resulting from the translation of the inverse matrix calculation with Cramer's method.
Subroutine invGJL(a, b)
implicit none
real a(4,4)
real b(4,4)
real p
integer i
integer j
integer k
call idmat(4,b)
do i = 1, 4, 1
  p = 1/a(i,i)
do j = 1, 4, 1
  a(i,j) = p*a(i,j)
b(i,j) = p*b(i,j)
end do
end do
if (k.ne.i) then
  q = a(k,i)
do j = 1, 4, 1
  a(k,j) = -q*a(k,j) + a(k,j)
b(k,j) = -q*b(k,j) + b(k,j)
end do
end if
end do
end do
end

Fig. 4. FORTRAN 77 subroutine resulting from the translation of the conventional loop algorithm implementing the Gauss-Jordan elimination.

Table 2
Computation times (µs) of a rank 4 matrix inversion with the classical Gauss-Jordan algorithm (GJ), the expanded GJ, the unrolled GJ, the expanded Cramer algorithm and the expanded decomposed algorithm, without and with compiler optimisation (-O3).

<table>
<thead>
<tr>
<th></th>
<th>Classic GJ</th>
<th>Expanded GJ</th>
<th>Unrolled GJ</th>
<th>Cramer</th>
<th>Decomposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>No opt.</td>
<td>15.7</td>
<td>2.9</td>
<td>6.0</td>
<td>3.6</td>
<td>2.7</td>
</tr>
<tr>
<td>O3</td>
<td>7.35</td>
<td>2.46</td>
<td>4.49</td>
<td>3.11</td>
<td>2.23</td>
</tr>
</tbody>
</table>

The examples presented in this section show how a CAS can be useful in evaluating the efficiency of different implementations of algorithms. Their use will satisfy the creativity of the developer allowing him to try different ideas. We have applied the expansion technique to other inversion methods of a 4 × 4 matrix as the Strassen algorithm [15,17] and the Mazúch-Kozánek algorithm [11]. All of them showed a big gain with respect to the correspondent classical programming of the same algorithm. Another interesting method consists in decomposing the matrix in 2 × 2 submatrices

\[
\text{In}[24]:= \mathbf{a} \rightarrow \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} ;
\]

where

\[
\text{In}[25]:= \mathbf{A}_{k,1} \rightarrow \begin{pmatrix} \mathbf{a}_{2}(k-1)+1, 2(1-1)+1 & \mathbf{a}_{2}(k-1)+1, 2(1-1)+2 \\ \mathbf{a}_{2}(k-1)+2, 2(1-1)+1 & \mathbf{a}_{2}(k-1)+2, 2(1-1)+2 \end{pmatrix} ;
\]
Subroutine invGJE(a, b)
implicit none
real a(4,4)
real b(4,4)
real q(30)
q(1) = 1/a(1,1)
q(2) = -a(1,2) * q(1)
q(3) = -a(2,2) + a(2,1) * q(2)
q(4) = -a(3,2) + a(3,1) * q(2)
q(5) = 1/q(3)
q(6) = -a(2,3) + a(2,1) * q(4)
q(7) = -a(3,3) + a(3,1) * q(4)
q(8) = -a(4,2) * q(1)
q(9) = q(5) * q(6)
q(10) = a(4,2) - a(4,1) * q(2)
q(11) = -a(3,3) + a(3,1) * q(4) - q(7) * q(9)
q(12) = q(5) * (a(2,4) - a(2,1) * q(8))
q(13) = a(2,1) * q(1)
q(14) = q(5) * q(7)
q(15) = 1/q(11)
q(16) = a(3,4) - a(3,1) * q(8) - q(7) * q(12)
q(17) = -a(4,3) + a(4,1) * q(4) - q(9) * q(10)
q(18) = q(4) - q(2) * q(9)
q(19) = -a(3,3) * q(1) + q(13) * q(14)
q(20) = q(15) * q(16)
q(21) = q(15) * q(18)
q(22) = q(5) * q(10)
q(23) = q(15) * q(19)
q(24) = q(15) * q(19)
q(25) = -q(8) + q(2) * q(11) - q(18) * q(20)
q(26) = 1/(a(4,4)) - a(4,1) * q(8) - q(10) * q(12) - q(17) * q(20)
q(27) = -q(12) - q(9) * q(20)
q(28) = -q(4) - q(2) * q(9)
q(29) = -q(22) + q(14) * q(24) + q(26)
q(30) = q(24) * q(26)
b(1,1) = q(1) + a(1,2) * q(2) + a(1,1) * q(5) + a(1,1) * q(11) - q(19) * q(21) - q(25) * q(28)
b(1,2) = -(q(2) * q(5)) + q(14) * q(21) - q(25) * q(29)
b(1,3) = -(q(2) * q(5)) + q(25) * q(30)
b(1,4) = -(q(25) * q(26))
b(2,1) = -(q(5) * q(13)) - q(3) * q(23) - q(27) * q(28)
b(2,2) = q(5) + (q(6) * q(7) * q(15)) / q(3) * q(22) - q(27) * q(29)
b(2,3) = -(q(9) * q(15)) + q(27) * q(30)
b(2,4) = -(q(26) * q(27))
b(3,1) = q(23) - q(20) * q(28)
b(3,2) = -(q(14) * q(15)) + q(20) * q(29)
b(3,3) = q(15) + (q(16) * q(17) * q(26)) / q(11) * q(22)
b(3,4) = -(q(20) * q(25))
b(4,1) = -q(28)
b(4,2) = -q(29)
b(4,3) = -q(30)
b(4,4) = -q(26)
end

Fig. 5. FORTRAN 77 subroutine resulting from translation of the expanded Gauss-Jordan algorithm.

and computing the inverse of \( a \) on the basis of the inverse matrices \( A^{-1}_{k,1} \) by solving the equation

\[
In[26]:= \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} 1 \ 0 \\ 0 \ 1 \end{pmatrix}
\]

with respect to the unknown submatrices \( B_{k,1} \). The symbols \( \{1\}_{2} \) and \( \{0\}_{2} \) represent the identity and zero matrices of rank 2. Using the same CPU time convention
as above, the relative CPU time taken by this kind of expansion is 17.

The technique presented in this section brings benefits when expanding the algorithm lets appear multiple occurrences of the same operation produced in different combinations. This gives potential time saving via common subexpression elimination. If it is not the case, the technique presented in the next section is more suitable.

3.4 Optimization through algorithm unrolling

Another way to write explicitly an algorithm expressed with Do and If blocks is to extend the technique of unrolling applied by the majority of compilers. This permits to avoid the overheads caused by the end loop test conditions. With CAS, loop unrolling may be done very naturally, just exploiting their evaluation process. Unroll algorithms consists in writing sequentially all the statements that would be executed by the algorithm in the block form. We show an application of this principle by unrolling the Gauss-Jordan elimination algorithm discussed in Section 3.3 and expressed by Eq[4]. The operator UnrollAlgorithm of package Math-Compile implements this functionality. It is invoked as follows:

\[
\text{In}[27]:= \text{Eq}[7] = \text{UnrollAlgorithm}\{\{a./\text{Eq}[1]\}, \text{Eq}[4]\}
\]
\[
\text{Out}[27]= \{\{u \rightarrow \{(a_{1,1}, a_{1,2}, a_{1,3}, a_{1,4}), (a_{2,1}, a_{2,2}, a_{2,3}, a_{2,4})\},
\{a_{3,1}, a_{3,2}, a_{3,3}, a_{3,4}\}, (a_{4,1}, a_{4,2}, a_{4,3}, a_{4,4})\}\},
\{b \rightarrow \text{IdMat}[4]\}, \{p \rightarrow \frac{1}{u_{1,1}}\}, \{u_{1,1} \rightarrow pu_{1,1}\}, \{b_{1,1} \rightarrow pb_{1,1}\},
\{u_{1,2} \rightarrow pu_{1,2}\}, \{b_{1,2} \rightarrow pb_{1,2}\}, \{u_{1,3} \rightarrow pu_{1,3}\},
\{b_{1,3} \rightarrow pb_{1,3}\}, \ldots , \{b_{4,4} \rightarrow -qb_{1,4} + b_{4,4}\}, \{p \rightarrow \frac{1}{u_{2,2}}\},
\{u_{2,1} \rightarrow pu_{2,1}\}, \{b_{2,1} \rightarrow pb_{2,1}\}, \{u_{2,2} \rightarrow pu_{2,2}\}, \{b_{2,2} \rightarrow pb_{2,2}\},
\{u_{2,3} \rightarrow pu_{2,3}\}, \ldots , \{b_{4,4} \rightarrow -qb_{2,4} + b_{4,4}\},
\{p \rightarrow \frac{1}{u_{3,3}}\}, \{u_{3,1} \rightarrow pu_{3,1}\}, \{b_{3,1} \rightarrow pb_{3,1}\}, \{u_{3,2} \rightarrow pu_{3,2}\},
\{b_{3,2} \rightarrow pb_{3,2}\}, \ldots , \{b_{4,4} \rightarrow -qb_{3,4} + b_{4,4}\},
\{p \rightarrow \frac{1}{u_{4,4}}\}, \{u_{4,1} \rightarrow pu_{4,1}\}, \{b_{4,1} \rightarrow pb_{4,1}\}, \{u_{4,2} \rightarrow pu_{4,2}\},
\{b_{4,2} \rightarrow pb_{4,2}\}, \ldots , \{u_{3,4} \rightarrow u_{3,4} + q u_{4,4}\}, \{b_{3,4} \rightarrow b_{3,4} - qb_{4,4}\}\}
\]

As done before, in translating the unrolled algorithm in FORTRAN 77 we use the classical memory manage in the Gauss Jordan elimination scheme, consisting in using the input matrix as storage array, i.e.: \(u \rightarrow a\).
In[28]:- FortranProgramUnit[S, inv4GJU,

    {{a, real, {4, 4}}, {b, real, {4, 4}}},
    {{p, real}, {q, real}},
    Drop[Eq[7], 1], {u \rightarrow a}]

The coded subroutine is shown in Fig. 6. Notice that the invocation to the function IdentityMatrix appearing in Eq[4] has been translated by a call to a specialised version (idmat 4) of the FORTRAN subroutine producing an identity matrix. This subroutine has been obtained by expansion of the corresponding algorithm.

We have tested this subroutine and compared the CPU time with the ones discussed in Section 3.3. Assigning the value 100 to the time spent by the classical implementation, this one takes 38. This is more than the time spent by the expanded implementation. A full comparison between the CPU times for an Absoft FORTRAN compiler on a MacIntosh G3 233 MHz processor (with two optimisation levels) is shown in Table 2. It must be emphasized that this implementation executes exactly the same operations of the classical one, in the same order. The computing time saving is due to avoiding the loop overhead and the tested condition k \neq 1. In our case this test is executed at “compile time” and not at run time. It is obvious that this has been possible because the test condition is performed on the value of loop indices (known at compile time) and not on values of the matrix elements. For this reason the Gauss-Jordan algorithm with pivoting can not be unrolled.

The unrolling technique can be used in the case of recursive algorithms. In this case the unrolling is performed by recursively inlining the calls to the function implementing the algorithm. We have tested this application on the Strassen matrix multiplication algorithm [15,17] applied to a 4 \times 4 matrix. This method has the particularity to change the number of elementary multiplications from \( N^3 \) to \( N^{\log_2 7} \) for a \( N \times N \) matrix. As it increases the number of additions it is interesting for high values of \( N \). In our implementation, if we assign the value of 100 to the time spent by the classical one (as presented in Section 3.2), the time taken is 24, more than the one taken by the natural expansion. This result is due to the fact that for \( N = 4 \), the reduction of the number of multiplications from exponent 3 to exponent \( \log_2 7 \), does not compensate the increase of the number of additions.

In the case of iterative algorithms where the test condition is performed on the residual error of the result, the unrolling strategy can also be applied. Let us imagine that from a statistical investigation it appears that our algorithm takes an average of \( N \) iterations to converge and it is called millions of times in our code. A good strategy is to unroll \( N \) instances of one iteration (without performing the convergence test) and insert the unrolled sequence in a higher loop that performs the convergence test. In the majority of cases only one test is executed and a gain in the loop overhead (and convergence test) is obtained. A test on a subprogram implementing the Newton root search of transcendental equations has been done. The version coded using
Subroutine inv4GJU(a, b)
implicit none
real a(4, 4)
real b(4, 4)
real p
real q
call idmat4(b)
p = 1/a(1,1)
a(1,1) = p*a(1,1)
b(1,1) = p*b(1,1)
a(1,2) = p*a(1,2)
b(1,2) = p*b(1,2)
a(1,3) = p*a(1,3)
b(1,3) = p*b(1,3)
a(1,4) = p*a(1,4)
b(1,4) = p*b(1,4)
...
q = a(4,4)
a(4,1) = -(q*a(1,1)) + a(4,1)
b(4,1) = -(q*b(1,1)) + b(4,1)
a(4,2) = -(q*a(1,2)) + a(4,2)
b(4,2) = -(q*b(1,2)) + b(4,2)
a(4,3) = -(q*a(1,3)) + a(4,3)
b(4,3) = -(q*b(1,3)) + b(4,3)
a(4,4) = -(q*a(1,4)) + a(4,4)
b(4,4) = -(q*b(1,4)) + b(4,4)
...
p = 1/a(4,4)
a(4,1) = p*a(4,1)
b(4,1) = p*b(4,1)
a(4,2) = p*a(4,2)
b(4,2) = p*b(4,2)
a(4,3) = p*a(4,3)
b(4,3) = p*b(4,3)
a(4,4) = p*a(4,4)
b(4,4) = p*b(4,4)
...
q = a(3,4)
a(3,1) = a(3,1) - q*a(4,1)
b(3,1) = b(3,1) - q*b(4,1)
a(3,2) = a(3,2) - q*a(4,2)
b(3,2) = b(3,2) - q*b(4,2)
a(3,3) = a(3,3) - q*a(4,3)
b(3,3) = b(3,3) - q*b(4,3)
a(3,4) = a(3,4) - q*a(4,4)
b(3,4) = b(3,4) - q*b(4,4)
end

Fig. 6. FORTRAN 77 subroutine resulting from the translation of the unrolled Gauss-Jordan algorithm.

the UnrollAlgorithm function showed a CPU time saving spreading from 5 to 20%, depending on the complexity of the equation and on the optimization level demanded to the FORTRAN compiler (the more is the optimization done by the compiler, the less is the saving coming from the unrolling technique).

3.5 Optimization of test conditions and loops by statement replication

It is well known that If blocks nested in Do loops slow down the CPU performance. One way to avoid this problem is to reverse the If and Do, when the condition is loop independent, at the price of code replication. But the loss in memory
is negligible with respect to the gain in CPU time. Techniques similar to this one have been already implemented by compilers \[12,13\]. To show how to apply this technique, let us consider the following FORTRAN programming sequence where, to factor one instruction, the programmer has nested an \texttt{If} block into a \texttt{Do} loop.

\begin{verbatim}
   do i = 1, n
      if (b .gt. 1) then
         x = a(i)*b
      else
         x = a(i)*a(i)
      end if
      y(i,1) = x*cos(b*x) + g(i,1)*f(i,2)
      y(i,2) = x*sin(b*x) + g(i,2)*f(i,1)
   end do
\end{verbatim}

One of the reasons invoked to justify the factorization is modularity. It is true that modularity must be maintained until the lower levels of programming. Doing that, one is sure that under each one of the two conditions the statement executed is the same. Writing twice the same statement is dangerous during the evolution of the code. One people could change one of them and keep unchanged the other one. Writing a subroutine instead is not a good solution because it adds a jump. If the programming sequence is written in algorithm specification language, the programmer can write the sequence in this way, and ask the CAS to optimize.

\texttt{In[29]}:= \texttt{DoIfBlock} :=
\begin{verbatim}
   Do [{
      If[b > 1,
         {x \rightarrow a_1 b},
         {x \rightarrow a_1^2}],
      y_{i,1} \rightarrow x \, \cos(b \, x) + g_{i,1} \, f_{i,2},
      y_{i,2} \rightarrow x \, \sin(b \, x) + g_{i,2} \, f_{i,1}],
      {i, 1, n}
   }
\end{verbatim}

In fact, we can see that the test condition does not depend on the loop iteration. Then the \texttt{If} block can be extracted from the \texttt{Do} loop. The transformation rule \texttt{ReverseDoIf} of package MathCompile performs this optimization. Of course it verifies the independence of the test condition before doing the transformation.

\texttt{In[30]}:= \texttt{IfDoBlock = DoIfBlock/.ReverseDoIf}
if (b.gt.1) then
  do i = 1, n, 1
  x = b*a(i)
  y(i,1) = x*Cos(b*x) + f(i,2)*g(i,1)
  y(i,2) = f(i,1)*g(i,2) + x*Sin(b*x)
  end do
else
  do i = 1, n, 1
  x = a(i)*x
  y(i,1) = x*Cos(b*x) + f(i,2)*g(i,1)
  y(i,2) = f(i,1)*g(i,2) + x*Sin(b*x)
  end do
end if

Fig. 7. FORTRAN translation of the IF-Do block fragment with code replication.

Out[30] = If[b > 1,

Do [{(x = a_i b), y_{i,1} = x Cos[b x] + g_{i,1} f_{i,2}, y_{i,2} = x Sin[b x] + g_{i,2} f_{i,1}},
       (i, 1, n)], Do[{(x = a^2_i), y_{i,1} = x Cos[b x] + g_{i,1} f_{i,2},
                      y_{i,2} = x Sin[b x] + g_{i,2} f_{i,1}}, (i, 1, n)]

The result of the final FORTRAN translation of this programming sequence (obtained by invocation of the functional operator FortranFragment of MathCompile) contains twice the expressions but the generated binary code is more efficient.

In[31]:= FortranFragment[ifdoblk, IfDoBlock]

The coded fragment is shown in Fig. 7.

Applying source to source optimizing transformations is a currently used technique. Some optimizing compilers perform the transformation using the same language for the initial and target code. This is the case, for example, of the parallelizer compiler Polaris [14,8] that operates on FORTRAN in input and output. Acting on an intermediate specification language as the one presented in this paper, has the advantage to be manipulated by high level CAS operators. This makes easy building optimizing functional operators even for a person not skilled with compiler’s techniques. For example, the transformation rule ReverseDoIf takes about 10 lines, including the dependence analyzer.

4. Discussion

The optimizations that we presented, although inspired by traditional types of optimization, are not performed in general by compilers. For example compilers perform loop unrolling but they do not make algorithm expansion or unrolling. Nevertheless compilers perform other types of optimization that are not realizable by CAS, like fine optimizations in register allocation. The optimization performed by the CAS is not an alternative to the one performed by the compiler but a complement.
CAS optimizing techniques can be combined to find the optimal solution of a problem of fixed size. Fixed size problems occur in numerical simulators where meshing does not change from one simulation to another one. As the problem can be of very big size, full expansion cannot be executed on the global solving algorithm. Indeed the result of the algorithm expansion can reach a very large size and beyond a certain limit it is not advised to proceed to this technique, because several problems can be encountered: the memory size of the code increases; the compiler will not be able to perform fine optimizations. To quantify this assertion we present in Table 3 the relation between matrix rank and memory requirement (in terms of number of temporary variables and size of the binary object file), for the expansion of the matrix inversion with the Gauss-Jordan algorithm. It is not advised to go beyond rank 10 in order to avoid a big amount of temporary variables that could compromise the compiler performance if their size was greater than the maximum cache size. Moreover, for big size problems, the limits of the CAS can be reached (memory overflow, huge run time). Nevertheless the problem matrix can be decomposed in submatrices over which the expansion can be performed. For example, for a problem described by a tridiagonal matrix, we can decompose the matrix in a series of $10 \times 10$ submatrices (if the rank of the matrix is multiple of 10) whose expanded symbolic solution (using the technique presented in Section 3.3), after common subexpression elimination, is very compact. The global solution can be obtained by solving the tridiagonal $10 \times 10$ block matrix system with the backsubstitution algorithm. This can be done in a loop style programming or by unrolling the algorithm with the technique presented in Section 3.4. If we use the classical loop style programming we can apply the expansion optimization technique to a problem of variable size. In this case the remainder of submatrices decomposition (the blocks) must be inversely by a classical method.

Table 3
Number of temporary variables and size of the binary object file (bytes), without and with compiler optimisation (-O3), for the expansion of the matrix inversion with the Gauss-Jordan algorithm, versus the rank of the matrix.

<table>
<thead>
<tr>
<th>Rank</th>
<th>4</th>
<th>6</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temp. var. dim.</td>
<td>30</td>
<td>83</td>
<td>246</td>
</tr>
<tr>
<td>Bin. size, No opt.</td>
<td>2063</td>
<td>4867</td>
<td>17280</td>
</tr>
<tr>
<td>Bin. size, O3</td>
<td>1687</td>
<td>3859</td>
<td>11904</td>
</tr>
</tbody>
</table>

The techniques presented here are an example of the possibilities offered by CASs. Each domain has its potential sources of computing time saving which can be discovered by the developer trying his ideas quickly thanks to CASs.

If CASs were introduced in compilers, some of these techniques could be used to improve the efficiency in partial evaluation [10,3]. The compiler itself could replace general purpose routines by the specialised programs obtained by expansion or unrolling of the programmed algorithms.

As far as accuracy is concerned there is a difference between CASs and usual lan-
guages as FORTRAN. In CASs we can adjust the precision and then reach a very high numerical accuracy. Usual languages allow also adjusting the precision but they are less flexible in the choice.

Package MathCompile is written in Mathematica [18] language but other CASs as Maple [16] or MuPAD [9] could be used. It is composed of functional operators and transformation rules working on mathematical expressions. MathCompile is a package in evolution. The existing functions can be enhanced and new functions can be added in order to perform transformations that appear necessary when new types of problems are approached. Although this package has been conceived to generate code components in the field of diffusion phenomena (for example nuclear reactor physics) and fluid dynamics, it can be used to develop scientific computer programs from specification to code production when the complete mathematical formulation is given.

5. Conclusion

We have presented some optimization techniques that can be applied by using computer algebra systems. They are an extension of similar optimizations done by actual compilers. Compiler optimizations, as they are decided automatically, can not always be superposed and a trade off is needed between conflicting strategies. Moreover the pattern detection for optimization can fail, for lack of information in the program structure. A solution to these problems is the control of optimization process taken by the developer, in the algorithm specification phase, as done according to our approach. For this purpose, functionalities of CAS and its associated packages, as the one presented here (MathCompile) can be used in a modular way. The developer decides what kind of transformation must be done and where. If the developer needs some new functions in his development process, he can write them (in the form of modules and transformation rules) and insert them in his own package. A package, as MathCompile, is a collection of modules and transformation rules written in Mathematica language. Using it enables extending Mathematica with functional operators necessary in the scientific code development process. The following capabilities are included in MathCompile: identification of special structure matrix coefficients in sets of equations, optimizations as common subexpression elimination, translation in FORTRAN language.

The tests performed show a good CPU time gain in the optimized code. Although the results correspond to a Macintosh G3 233 MHz processor, tests on other architectures have shown similar trends but with different time ratios. This approach could be extended to other aspects of optimization, as parallelization. If compared to the code synthesis systems philosophy, the approach proposed here demands more work to the developer, but less if compared to the classical development approach. Its finer extent assigns it a larger flexibility and a wider application domain.
References


