A Toolbox for Program Manipulation and Efficient Code Generation with an Application to a Problem in Computer Vision

Michael Monagan & Gladys Monagan*

April 5, 1997

Abstract

We describe the design of a package for creating efficient numeric code. The package provides the user with tools for creating and manipulating programs, in this case Maple programs, converting the programs into C and Fortran code, and compiling and executing the programs from inside Maple. The tools for manipulating programs include facilities for automatic differentiation, code optimization, and the complexity analysis of a program. We also provide tools for structural manipulation of code, for example, converting formulae in the code into Horner form. An application to an optimization problem from computer vision which requires a gradient computation is given.

*Centre for Experimental and Constructive Mathematics, Simon Fraser University, Burnaby, British Columbia, V5A 1S6, Canada. `monagan@cecm.sfu.ca`

1 Introduction

This paper is about how to design a facility within a computer algebra system (CAS), in this case Maple, to make it easy for a user of the system to create an efficient numeric code, often C or Fortran, from a symbolic representation of a problem. The package is applied to an optimization problem in computer vision where we are trying to determine the wear of a railway line as accurately as possible. We wanted to completely automate the task of producing an efficient gradient program so that different models could be tried out quickly without error.

Most CASs contain a facility for converting a single formula, or a vector or matrix of formulae, into C or Fortran code. In some CASs, a code optimization facility is provided. The user can insert this (optimized) code in a C or Fortran subroutine using a text editor. Additionally, some software has been developed to further automate this "assembly process", to include, for example, the necessary program headers, variable declarations, compiler directives, etc. Previous work in this area includes Gates's GENTRAN package [?], the work of van Hulzen et al [?] in REDUCE, Dewar [?], and others. Gomez's macrofort and Capolsini's macroC [?] are prior attempts to do this in Maple.

A problem with this approach is that the user must be familiar with three languages; the language of the CAS in which he or she is working, the language of the "assembly directives" in which the numeric computation is created and represented, and thirdly, the target language, which is usually C or Fortran. This is because the assembly language directives often correspond closely with the target language.

Our approach differs in that we assemble a complete program in the CAS language before translation. This means that the computation can be executed in the CAS, which makes prototyping easier and provides a check against the final numeric code. We also provide tools for manipulating the program. These tools include

(i) differentiation of the program (automatic differentiation),

(ii) optimization of the program

(iii) estimation of the time complexity of the program,

(iv) tools for repackaging the program calling sequence,

(v) tools for manipulating the formulae inside the programs, and

(vi) translation into a C/Fortran executable.

There are other important advantages in assembling a complete program, and doing so in the CAS. Firstly, representing a computation as a program is theoretically more efficient than trying to do so using formulae because loops, local variables, and subroutine calls reduce the time and space complexity of the code. Secondly, the use of the reverse mode for automatic differentiation is more efficient than either symbolic differentiation or the numeric approach of divided differences. See [?] and [?] for an overview of the field of automatic differentiation. If the input program is a function of n variables, and it does m arithmetic operations (or elementary functions), then the complexity of the resulting code for computing gradient of the program is $O(m+n)$, compared with $O(m \times n)$ for either of the other approaches.

We are not the first to attempt to design a system to manipulate programs in the context of CA systems. In [?] Kaltofen et al designed DAGWOOD, a system for creating and manipulating straight-line programs. The primary purpose of DAGWOOD was to compute GCDs of and factor polynomials represented by straight-line programs using modular based methods. In our target applications, the primary purpose is to generate efficient numeric codes, hence computing with floating point numbers and arrays of floating point numbers is the main focus. However, for such a design to be of practical use, it must be easy to create and manipulate these programs, ideally, as easy as it is to create, manipulate, simplify, and expand formulae.

The main application of our package is efficient derivative computation based on automatic differentiation. CA systems are effective at computing derivatives; they can handle problems of modest size and thus can be used reliably inside scientific applications. In contrast, CA systems are not effective at integration or solving polynomial systems or solving differential equations because of the non-existence of solutions or the exponential complexity of the algorithms to compute them; thus for such problems, CA systems often cannot be used in scientific applications. Computing derivatives is an important application in scientific computing and the use of automatic differentiation means truly large problems can be handled by a CAS.

2 The Code Generation Package

The `codegen` package contains a number of components. To motivate the existence of these particular components, we walk through two simple applications which motivate the need for the capabilities demonstrated. We end with a summary giving some details about the package. The package will execute in Maple V Release 4. It is available from the authors.

2.1 A Jacobian Example

We show an example of a Jacobian computation. We begin by assuming that somehow the scientist or engineer has created two functions f and g in two variables x and y, which are represented in the CAS by formulae.

```
> f := exp(x+y)*y*x+exp(x+y)*x^2;
                                                      2
                   f := exp(x + y) y x + exp(x + y) x

> g := 1-exp(x+y)*y*x-exp(x+y)*x;
                  g := 1 - exp(x + y) y x - exp(x + y) x
```

It is desired to calculate the Jacobian matrix of partial derivatives. Many users of CA systems would proceed as follows. First, they would use the built-in symbolic differentiation facility to generate formulae for the Jacobian matrix. Next, to use this matrix with numerical software, they would convert it to C or Fortran code. Some editing of the code generated might then be required.

Instead, we proceed as follows. First we need to build a program for computing f and g in the CAS. Our program returns a vector of the two values $f(x, y)$ and $g(x, y)$.

```
> with(codegen):
Warning, new definition for fortran
> A := array([f,g]);
           [                            2                                ]
      A := [exp(x + y) y x + exp(x + y) x , 1 - exp(x + y) y x - exp(x + y) x]

> F := makeproc( A, [x,y] );
F := proc(x, y)
local A;
    A := array(1 .. 2);
    A[1] := exp(x + y)*y*x + exp(x + y)*x^2;
    A[2] := 1 - exp(x + y)*y*x - exp(x + y)*x;
    A
end
```

$$A := [e^{(x+y)}\, y\, x - e^{(x+y)}\, x^2,\ 1 - e^{(x+y)}\, y\, x - e^{(x+y)}\, x]$$

$$
\begin{array}{l}
F := \mathbf{proc}(x,\ y) \\
\quad \mathbf{local} A; \\
\qquad A := \mathrm{array}(1..2)\,; \\
\qquad A_1 := \exp(x+y) \times y \times x - \exp(x+y) \times x^2\,; \\
\qquad A_2 := 1 - \exp(x+y) \times y \times x - \exp(x+y) \times x\,; \\
\qquad A \\
\quad \mathbf{end}
\end{array}
$$

Of course, if we were writing the program by hand we would not calculate the exponential four times. We would store it in a local variable. Hence before we proceed to calculate the Jacobian, we optimize the program. This will also reduce the time and space complexity of computing the Jacobian and the space complexity of the resulting Jacobian program.

```
> F := optimize(F);
F := proc(x, y)
local A, t2, t4, t5;
    A := array(1 .. 2);
    t2 := exp(x + y);
    t4 := t2*y*x;
    t5 := x^2;
```

```
    A[1] := t4 + t2*t5;
    A[2] := 1 - t4 - t2*x;
    A
end
```

Now we create the Jacobian program using automatic differentiation. The GRAD command is detailed in [?]. The output is a program which on input of x, y constructs the Jacobian matrix.

```
> J := GRAD(F,[x,y],result_type=array);
J := proc(x, y)
local dfr0, t2, t4, t5, A1, A2, grd, df;
    t2 := exp(x + y);
    t4 := t2*y*x;
    t5 := x^2;
    A1 := t4 + t2*t5;
    A2 := 1 - t4 - t2*x;
    df := array(1 .. 5);
    dfr0 := array(1 .. 5);
    df[4] := 1;
    df[3] := df[4]*t2;
    df[2] := df[4];
    df[1] := df[4]*t5 + df[2]*y*x;
    dfr0[5] := 1;
    dfr0[2] := -dfr0[5];
    dfr0[1] := - dfr0[5]*x + dfr0[2]*y*x;
    grd := array(1 .. 2, 1 .. 2);
    grd[1, 1] := 2*df[3]*x + df[2]*t2*y + df[1]*exp(x + y);
    grd[1, 2] := df[2]*t2*x + df[1]*exp(x + y);
    grd[2, 1] := - dfr0[5]*t2 + dfr0[2]*t2*y + dfr0[1]*exp(x + y);
    grd[2, 2] := dfr0[2]*t2*x + dfr0[1]*exp(x + y);
    RETURN(grd)
end
```

Now, because of the chain rule, common subexpressions have re-appeared in the Jacobian, and secondly, we can see that constant folding will be effective on this example. A further optimization pass is usually effective.

```
> J := optimize(J);
J := proc(x, y)
local t4, t7, t8, df, t11, t13, dfr0, t5, t2, grd, t3;
    t2 := exp(x + y);
    t3 := t2*y;
    t4 := t3*x;
    t5 := x^2;
```

```
    t7 := t2*x;
    df := array(1 .. 5);
    dfr0 := array(1 .. 5);
    df[4] := 1;
    df[3] := t2;
    df[2] := 1;
    t8 := y*x;
    df[1] := t5 + t8;
    dfr0[5] := 1;
    dfr0[2] := -1;
    dfr0[1] := -x - t8;
    grd := array(1 .. 2, 1 .. 2);
    t11 := df[1]*t2;
    grd[1, 1] := 2*t7 + t3 + t11;
    grd[1, 2] := t7 + t11;
    t13 := dfr0[1]*t2;
    grd[2, 1] := -t2 - t3 + t13;
    grd[2, 2] := -t7 + t13;
    grd
end
```

Next, we estimate the complexity of the program for comparison with other approaches and execute it on some trial values.

```
> cost(J);
     22 storage + 22 assignments + functions + 9 additions + 8 multiplications

> print(J(1.0,1.0));
                           [36.94528050     22.16716830 ]
                           [                            ]
                           [-29.55622440    -22.16716830]
```

Now we convert the program to a C subroutine, compile it (in C), and execute it on the data $x = 1.0, y = 1.0$ as a practical check that the compiled program is correct. In translating the program to C we are making the obvious assumptions about the types of the variables.

```
> C(J);
#include <math.h>
void J(x,y,grd)
double x;
double y;
double grd[2][2];
{
  double t4, t7, t8, df[5], t11, t13, dfr0[5], t5, t2, t3;
```

```
  {
    t2 = exp(x+y);
    t3 = t2*y;
    t4 = t3*x;
    t5 = x*x;
    t7 = t2*x;
    df[3] = 1.0;
    df[2] = t2;
    df[1] = 1.0;
    t8 = y*x;
    df[0] = t5+t8;
    dfr0[4] = 1.0;
    dfr0[1] = -1.0;
    dfr0[0] = -x-t8;
    t11 = df[0]*t2;
    grd[0][0] = 2.0*t7+t3+t11;
    grd[0][1] = t7+t11;
    t13 = dfr0[0]*t2;
    grd[1][0] = -t2-t3+t13;
    grd[1][1] = -t7+t13;
    return;
  }
```

Note, the two by two matrix is passed as a parameter in the C subroutine. We will comment on this below.

```
> cJ := compile(J,precision=single):
bytes used=2029988, alloc=1244956, time=3.07
compile:   Translating Maple code to C
compile:   Compiling and linking ./J.c and ./mainJ.c .
./J.c:
./mainJ.c:
> cJ(1.0,1.0);
                              [36.94528     22.16717 ]
                              [                      ]
                              [-29.55622    -22.16717]
```

If we are going to use J with a subroutine from a numerical library package, that package will probably be designed to call our J subroutine with a vector of the independent variables x and y, hence, we need to repackage the parameters of our J routine as follows

```
> J := packargs(J,[x,y],X);
J := proc(X::array(1 .. 2))
local t4, t7, t8, df, t11, t13, dfr0, t5, t2, grd, t3;
    t2 := exp(X[1] + X[2]);
    t3 := t2*X[2];
    t4 := t3*X[1];
```

```
    t5 := X[1]^2;
    t7 := t2*X[1];
    df := array(1 .. 5);
    dfr0 := array(1 .. 5);
    df[4] := 1;
    df[3] := t2;
    df[2] := 1;
    t8 := X[2]*X[1];
    df[1] := t5 + t8;
    dfr0[5] := 1;
    dfr0[2] := -1;
    dfr0[1] := -X[1] - t8;
    grd := array(1 .. 2, 1 .. 2);
    t11 := df[1]*t2;
    grd[1, 1] := 2*t7 + t3 + t11;
    grd[1, 2] := t7 + t11;
    t13 := dfr0[1]*t2;
    grd[2, 1] := -t2 - t3 + t13;
    grd[2, 2] := -t7 + t13;
    grd
end
```

Furthermore, since in the application the Jacobian program J is expected to be called several times by the application, one would typically pass a pointer to the storage for the J matrix in which to write the Jacobian result rather than dynamically allocating storage each time the program is called. This can be done explicitly as follows

```
> J := makevoid(makeparam(grd::'array'(1..2,1..2),J));
J := proc(X::array(1 .. 2), grd::array(1 .. 2, 1 .. 2))
local t3, t8, t7, t11, t13, dfr0, df, t5, t4, t2;
    t2 := exp(X[1] + X[2]);
    t3 := t2*X[2];
    t4 := t3*X[1];
    t5 := X[1]^2;
    t7 := t2*X[1];
    df := array(1 .. 5);
    dfr0 := array(1 .. 5);
    df[4] := 1;
    df[3] := t2;
    df[2] := 1;
    t8 := X[2]*X[1];
    df[1] := t5 + t8;
    dfr0[5] := 1;
    dfr0[2] := -1;
    dfr0[1] := -X[1] - t8;
    t11 := df[1]*t2;
    grd[1, 1] := 2*t7 + t3 + t11;
```

```
    grd[1, 2] := t7 + t11;
    t13 := dfr0[1]*t2;
    grd[2, 1] := -t2 - t3 + t13;
    grd[2, 2] := -t7 + t13;
    RETURN()
end
```

At some point one may wonder, whether, after a sequence of such manipulations, the resulting program is correct. A simple independent symbolic proof is often possible for straight-line programs.

```
> A := array(1..2,1..2):
> X := array([x,y]):
> J(X,A);
> A[1,1] - diff(f,x);
                2                                                2
              (x  + y x) exp(x + y) - exp(x + y) y x - exp(x + y) x

> simplify(");
                                            0
```

We emphasize also the following points about this example. Firstly, as mentioned, in general, a more efficient program is obtained by using automatic differentiation compared with symbolic differentiation. This can be quantified precisely by using the `cost` command to compare the cost of two programs. Secondly, there are various changes one may want to make to the calling sequence of a program to make it compatible with other software.

2.2 A Symbolic Loop Example

In this example we suppose that a user wants to build an efficient subroutine for evaluating a function $f(x)$. The example that we use is e^x to keep the presentation simple and the focus on the tools and not how to calculate e^x. For e^x we construct a simple Taylor series accurate to 10^{-7} on [0,0.5].

```
> t := taylor(exp(x),x,8):
> t := convert(t,polynom);
                       2        3         4          5          6            7
     t := 1 + x + 1/2 x  + 1/6 x  + 1/24 x  + 1/120 x  + 1/720 x  + 1/5040 x
```

This simplest way to build a program is to write what is essentially a template for the program and substitute specific formulae and values for dummy variables, T in this case, into the template.

```
> e := subs(T=t, proc(x) local h;
>     if x<0 then 1/e(-x)
>     elif x<0.5 then T
>     else  h := e(x/2); h^2
>     fi
>   end);
e := proc(x)
local h;
    if x < 0 then 1/e(-x)
    elif x < .5 then 1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5
         + 1/720*x^6 + 1/5040*x^7
    else h := e(1/2*x); h^2
    fi
end

> e := horner(e,x);
e := proc(x)
local h;
    if x < 0 then 1/e(-x)
    elif x < .5 then 1 +
        (1 + (1/2 + (1/6 + (1/24 + (1/120 + (1/720 + 1/5040*x)*x)*x)*x)*x)*x)*x
    else h := e(1/2*x); h^2
    fi
end
```

Of course the polynomial could have been converted to Horner form before the substitution was made, however, we argue that the tools for manipulating formulae should work for programs as well. We would like to be able to apply any command for manipulating formulae that preserves equality, such as collecting in a variable, factoring or expanding a formula, to a program. The default action of such commands is to apply the command to all formulae inside the program, i.e. to the right-hand side of assignment statements for example.

It is surely reasonable, however, to proceed in the following "symbolic" way —

```
> e := 'e':
> p := piecewise(
>   x<0, 1/e(-x),
>   x<0.5, Sum(x^i/i!,i=0..7),
>   e(x/2)^2 );
```

$$\begin{cases} (e(-x))^{-1} & x < 0 \\ \sum_{i=0}^{7} \frac{x^i}{i!} & x < 0.5 \\ (e(1/2\,x))^2 & \text{otherwise} \end{cases}$$

in which case, we have to be prepared to handle such symbolic constructions in a CA system. The piecewise function definition is a symbolic representation of a conditional statement. The Sum function can be thought of as a symbolic representation for a for loop. Note, the capital

S distinguishes Sum from the sum command in Maple which would compute (expand) the sum into $1 + x + x^2/2 + ... + x^7/5040$. We proceed as follows. First we create a program from the formula

```
> e := makeproc(p,x);
e := proc(x)
    piecewise(x < 0, 1/e(-x), x < .5, Sum(x^i/i!, i = 0 .. 7), e(1/2*x)^2)
end
```

Translation of the piecewise command into a conditional statement is straight-forward, however, a naive translation of the sum into a loop which leaves the factorial computed explicitly would destroy any efficiency that we are trying to obtain. Hence we must properly translate this symbolic sum construction in the CAS into the target language.

```
> e := prep2trans(e);
e := proc(x)
local t1, i2, t2, s2, t3;
    if x < 0 then t1 := e(-x); 1/t1
    elif x < .5 then
        t2 := 1; s2 := 1; for i2 to 7 do t2 := x*t2/i2; s2 := s2 + t2 od; s2
    else t3 := e(1/2*x); t3^2
    fi
end
```

Compiling, in single precision, and checking the result of the compiled C code with the interpreted code in Maple provides additional assurance that the compiled code is correct.

```
> ec := compile(e, precision=single):
compile:   Translating Maple code to C
compile:   Compiling and linking ./e.c and ./maine.c .
./e.c:
./maine.c:
> exp(1.0), e(1.0), ec(1.0);
                          2.718281828, 2.718281823, 2.718281
```

Users will expect that if a CAS contains symbolic representations for piecewise function definitions, sums and products, sequences, finite differences, etc. then they must be translatable, and the resulting code must be reasonable, at least as good as one would code by hand.

3 Details about the Package

Our approach to the design of the package is to provide a toolbox of Maple program-to-program transformations as illustrated in the previous two examples, and a language in which to program other transformations.

3.1 Maple Procedures as the Program Representation

The most important design question is what should this language for representing programs be? We feel that it should be the CA system's language and if necessary, the CA system needs to be designed or modified to support this. One needs to be able to construct and destruct programs, compile and execute or interpret programs, in the language of the CAS. We have had to modify the parameter type checking mechanism in Maple to make this work smoothly.

3.2 INTREP – an intermediate representation for Maple procedures

Although it is possible to program the Maple program to program transformations using Maple procedures directly as the data representation for programs (using the addressof, disassemble, assemble and pointto commands), it is awkward and error prone. What we do instead is first convert the Maple procedure into an intermediate form which is convenient to work with, then apply the program-to-program transformations on this intermediate form, then convert the resulting intermediate form back to a Maple procedure. The intermediate form allows us to attach type information about the Program's variables. We do not expect average users to see this intermediate form or need to program with it.

3.3 The GRAD facility for automatic differentiation

The automatic differentiation facility is described in [?]. It provides facilities for gradients, Jacobians and Hessians, of programs where any function for which Maple knows it's derivative can be used — actually, even if Maple does not know the derivative, it will still applies the chain rule, e.g.

```
> f := proc(x,y) g(x)*h(x,y) end;
> GRAD(f);
proc(x, y)
   RETURN([D[1](g)(x)*h(x, y) + g(x)*D[1](h)(x, y), g(x)*D[2](h)(x, y)])
end
```

3.4 The optimize facility

We've restricted our code optimizer to optimizations which can be done in expected linear time in the size of the input expressions. The reason for this is so that we can handle truly large codes, in particular large Hessians and Jacobians. This means that little data flow analysis can be done, however, basic optimizations such as common subexpression elimination and constant folding can be done in linear time. For common sub-expression optimization we make use of Maples builtin unique representation of expressions to identify all distinct subexpressions in linear time. Note, but subexpressions we understand that to mean suboperands of Maple expressions. For example, in Maple, `sin(x)` is a suboperand of `sin(x)*x*y` but `x*y` is not. For constant folding, we use assignment and evaluation to replace variables in formulae by their values in linear time.

3.5 The cost facility

The example previously given showed a simple calculation of the number of arithmetic operations that a straight-line program makes. We can also count the number of operations in loops and conditional statements. At present we do not analyze recursive programs. For example

```
> f := proc(l, m, n)
> local i, j, k;
> global a, b, c;
>     for i to l do
>       for j to n do
>         c[i, j] := sum(a[i, k]*b[k, j], k = 1 .. m)
>       od
>     od
> end:
> c := cost(prep2trans(f));
         /    /{   0           -1 + m < 0\\
c := l n |1 + |{                         || additions
         \    \{ -1 + m        otherwise //

           /{ 0        -1 + m < 0\
     + l n |{                    | multiplications
           \{ m        otherwise /

           //{ 1        -1 + m < 0\    \
     + l n ||{                    | + 1| assignments + 6 storage
           \\{ m        otherwise /    /
```

This is an exact operation count. We would like also to obtain is an asymptotic operation count, i.e. to see $O(lmn)$.

```
> assume(m>1);
> assignments := 0; # don't count assignments

                                  assignments := 0

> collect(c,[l,n,m],distributed);

                  6 storage + (additions + multiplications) m~ n l
```

3.6 Building and manipulating programs

The makeproc command is used to build a program with given parameters. We can build a program from a single formulae or an array of formulae. Symbolic sums and products, and piecewise function definitions can be used to encode some conditional statements and loops in formulae. We use equations to represent assignments and a list of expressions to represent a sequence of computations. More complicated programs must be built using the intermediate form of the programs.

3.7 The C, Fortran, and compile commands

The type of a variable in a Maple program is usually not specified. It needs to be determined if we are translating to C or Fortran code. If type information is not specified, we make heuristic assumptions about the types of certain expressions in Maple programs, such as the type of `i` in `A[2*i-1]` is an integer if `A` is an array. Other types are deduced and where there is no information we assume floating point. The compile command constructs a C executable and at present communicates data with Maple via files.

4 An Application in Computer Vision

The application presented in this section is part of a larger application of computer vision to a problem from the railway industry. For proprietary reasons, we are not able to give many details of the application. We provide the following for background information.

Rail wears over time. When the inside of the rail becomes too worn it is necessary to grind it to restore its shape. Because the steel used on rails is very hard, grinding is expensive, hence, minimizing the amount of grinding required, for example saving one one-thousandth of an inch, can save a lot of money. The old way of determining how much grinding is required was to have a technician stop at intervals along the railway line and measure by hand the wear of the rail. The new way of doing it is to attach what is essentially a light and a camera to a vehicle, take photographs of the rail and then measure the wear of the rails from the digital images. This can be done more quickly — the vehicle drives at 60 kmph along the line — and more thoroughly — several photos per second can be taken and processed. However, accuracy is very difficult to obtain for several technical reasons.

The computer processing of the digital photographs proceeds in outline as follows. An analysis of the digital image is done to determine the cross section of the rail line. In order to determine the wear, the image is compared with a model rail (the CAD specification of the rail) where the bottom of the rail is used as a reference point. The part of the application of interest in this paper is the matching of the base and stem of the rail. It is a basic optimization problem of matching data to a model. The CAD specification for the cross section of the rail is a piecewise function consisting of a sequence of arcs and lines. Thus what we want to do is to move (i.e. translate and rotate) the data to fit the pieces of the model image as closely as possible. That is, given a data point (x, y), we want to translate the point by (C, D) then rotate it by θ radians about the origin so that some function of the distance between the data and the model is minimized. We use a standard optimization method which requires that we be able to compute the gradient of the function with respect to C, D, θ. We require the gradient for the different curve types and we want to be able to try different measures for closeness.

Note that the approach described here is not restricted to data collected using structured light. For other types of images, one would typically binarize the gray scale image and then one would extract the contours of the objects to have data points to match against ideal models.

4.1 Arc Case

Given a point (x, y), we translate (x, y) by (C, D) then rotate by θ radians about the origin. In the code below, which we have input to Maple, the variables $\bar{x}$ and $\bar{y}$ are the coordinates of the resulting point. The function calculates the square of the distance between $(\bar{x}, \bar{y})$ and the edge of a circle with center (h, k) and radius r.

```
> f := proc(C,D,theta,x,y,h,k,r) local s,c,xbar,ybar,d1,d2,d;
>      s := sin(theta);
>      c := cos(theta);
>      xbar := (x+C)*c + (y+D)*s;
>      ybar := (y+D)*c - (x+C)*s;
>      d1 := (h-xbar)^2;
>      d2 := (k-ybar)^2;
>      d := sqrt( d1+d2 ) - r;
>      d^2
> end:
```

The optimized gradient, with respect to C, D, θ, is computed below. This was translated into C for use with the application.

```
> F := optimize( GRAD(f,[C,D,theta],result_type=array) );
F := proc(C, D, theta, x, y, h, k, r)
local dfr0, s, d1, grd, d2, t1, t3, c, t7, t8, t10, t19, t21;
    s := sin(theta);
    c := cos(theta);
    t1 := x + C;
    t3 := y + D;
    t7 := h - t1*c - t3*s;
    d1 := t7^2;
    t8 := k - t3*c + t1*s;
    d2 := t8^2;
    t10 := sqrt(d1 + d2);
    dfr0 := array(1 .. 7);
    dfr0[7] := 2*t10 - 2*r;
    dfr0[6] := 1/2*dfr0[7]/t10;
    dfr0[5] := dfr0[6];
    dfr0[4] := -2*dfr0[5]*t8;
    dfr0[3] := -2*dfr0[5]*t7;
    t19 := dfr0[4];
    t21 := dfr0[3];
    dfr0[2] := t19*t3 + t21*t1;
    dfr0[1] := - t19*t1 + t21*t3;
    grd := array(1 .. 3);
```

```
    grd[1] := - t19*s + t21*c;
    grd[2] := t19*c + t21*s;
    grd[3] := - dfr0[2]*s + dfr0[1]*c;
    grd
end
```

4.2 Line Case

This case is for the distance between a point, after translation and rotation, and a line. We tried using the signed distance rather than the square of the distance. Given a point (x, y), we translate it by (C, D) then rotate it by θ degrees about the origin to obtain $(\bar{x}, \bar{y})$ as before. Then we calculate the signed distance between $(\bar{x}, \bar{y})$ and the line $y = mx + b$ as follows

$$\begin{cases} \frac{m\bar{x}-\bar{y}+b}{\sqrt{m^2+1}} & b < 0 \\ -\frac{m\bar{x}-\bar{y}+b}{\sqrt{m^2+1}} & \text{otherwise} \end{cases}$$

where a positive distance corresponds to a point lying above the line. In the first version of the program below, as we tried it, the piecewise definition for the distance is implemented as a conditional statement. In the second version we have encoded it as a formula using the sign function, `signum` in Maple. We show both versions because what happens in the former case is is that the automatic differentiation does not recognize the special nature of the if statement and consequently produces what looks like poor code. This is typical of what happens with many of the program-to-program transformations that we have tried. The resulting code is sub-optimal and one really wants to improve the code optimizer.

```
> f := proc(C,D,theta,x,y,m,b) local s,c,xbar,ybar,d,r;
>      s := sin(theta);
>      c := cos(theta);
>      xbar := (x+C)*c+(y+D)*s;
>      ybar := (y+D)*c-(x+C)*s;
>      d := m*xbar-ybar+b;
>      r := sqrt(m^2+1);
>      if b>=0 then -d/r else d/r fi;
> end:
> GRAD(f,[C,D,theta],result_type=array);
proc(C, D, theta, x, y, m, b)
local ybar, xbar, r, d, c, grd, s, dfr0;
    s := sin(theta);
    c := cos(theta);
    xbar := (x + C)*c + (y + D)*s;
    ybar := (y + D)*c - (x + C)*s;
    d := m*xbar - ybar + b;
    r := sqrt(m^2 + 1);
    if 0 <= b then
        dfr0 := array(1 .. 5);
        dfr0[5] := - 1/r;
```

```
        dfr0[4] := -dfr0[5];
        dfr0[3] := dfr0[5]*m;
        dfr0[2] := dfr0[4]*(y + D) + dfr0[3]*(x + C);
        dfr0[1] := dfr0[4]*(-x - C) + dfr0[3]*(y + D);
        grd := array(1 .. 3);
        grd[1] := - dfr0[4]*s + dfr0[3]*c;
        grd[2] := dfr0[4]*c + dfr0[3]*s;
        grd[3] := - dfr0[2]*sin(theta) + dfr0[1]*cos(theta);
        RETURN(grd)
    else
        dfr0 := array(1 .. 5);
        dfr0[5] := 1/r;
        dfr0[4] := -dfr0[5];
        dfr0[3] := dfr0[5]*m;
        dfr0[2] := dfr0[4]*(y + D) + dfr0[3]*(x + C);
        dfr0[1] := dfr0[4]*(-x - C) + dfr0[3]*(y + D);
        grd := array(1 .. 3);
        grd[1] := - dfr0[4]*s + dfr0[3]*c;
        grd[2] := dfr0[4]*c + dfr0[3]*s;
        grd[3] := - dfr0[2]*sin(theta) + dfr0[1]*cos(theta);
        RETURN(grd)
    fi
end
```

Here is the second version of the code.

```
> f := proc(C,D,theta,x,y,m,b) local s,c,xbar,ybar,d,r;
>      s := sin(theta);
>      c := cos(theta);
>      xbar := (x+C)*c+(y+D)*s;
>      ybar := (y+D)*c-(x+C)*s;
>      d := m*xbar-ybar+b;
>      r := sqrt(m^2+1);
>      -signum(b)*d/r
> end:
> F := GRAD(f,[C,D,theta],result_type=array);
F := proc(C, D, theta, x, y, m, b)
local dfr0, r, d, xbar, grd, c, s, ybar;
    s := sin(theta);
    c := cos(theta);
    xbar := (x + C)*c + (y + D)*s;
    ybar := (y + D)*c - (x + C)*s;
    d := m*xbar - ybar + b;
    r := sqrt(m^2 + 1);
    dfr0 := array(1 .. 5);
    dfr0[5] := - signum(b)/r;
    dfr0[4] := -dfr0[5];
    dfr0[3] := dfr0[5]*m;
```

```
    dfr0[2] := dfr0[4]*(y + D) + dfr0[3]*(x + C);
    dfr0[1] := dfr0[4]*(-x - C) + dfr0[3]*(y + D);
    grd := array(1 .. 3);
    grd[1] := - dfr0[4]*s + dfr0[3]*c;
    grd[2] := dfr0[4]*c + dfr0[3]*s;
    grd[3] := - dfr0[2]*sin(theta) + dfr0[1]*cos(theta);
    RETURN(grd)
end
```

We show the C code for this case. You can see that the gradient vector grd is returned through a parameter.

```
> C(F,optimized,precision=single);
#include <math.h>
void F(C,D,theta,x,y,m,b,grd)
float C;
float D;
float theta;
float x;
float y;
float m;
float b;
float grd[3];
{
  float t1, c, s, r, dfr0[5], t16, t10, t14, t13, t8, t3;
  {
    s = sinf(theta);
    c = cosf(theta);
    t1 = x+C;
    t3 = y+D;
    t8 = m*m;
    r = sqrtf(t8+1.0);
    t10 = signum(b);
    dfr0[4] = -t10/r;
    t13 = dfr0[4];
    dfr0[3] = -t13;
    dfr0[2] = t13*m;
    t14 = dfr0[3];
    t16 = dfr0[2];
    dfr0[1] = t14*t3+t16*t1;
    dfr0[0] = -t14*t1+t16*t3;
    grd[0] = -t14*s+t16*c;
    grd[1] = t14*c+t16*s;
    grd[2] = -dfr0[1]*s+dfr0[0]*c;
    return;
  }
}
```

This code is clearly not limited to the computer vision application. This final generated code is close to the best possible piece of code and might be of value elsewhere. We remark that when we first tried this out, we rotated then translated the points. We then realized that the rest of the software in the application was assuming translation followed by rotation. Having automated the process this was trivial to change.

5 Conclusion

The way we have designed our code generation package is to use complete programs as the primary representation and to build all code manipulations as program-to-program transformations. This leads to a very clean system design in which the codes can be prototyped easily in the CA system. Also, from a complexity viewpoint, the use of programs to represent functions is more general and more efficient than formulae (the mainstay of CA systems). Although the components of our system worked smoothly together in the real application from computer vision that we presented, each of the main components of has it's shortcomings, however. We list a number of them here

— Program Creation
Because we have symbolic representations for formulae, piecewise functions, sums and products, we can easily assemble many programs. But we have no symbolic representation for an iteration, for example adding up the terms of a series until it converges. Thus if one wants to create a program with such an iteration, one must build the program in the intermediate representation. A nice addition to a CA system would be a symbolic form of an iteration.

— Program Limitations
Not all Maple programs can be converted to C or Fortran code, or differentiated by automatic differentiation software. What we would like to be able to do is to handle programs which perform matrix/vector arithmetic. For this a Fortran 90 code generator would be valuable.

— Program Optimization
The output of most program-to-program transformations leaves behind code that needs further optimization. Because the programs are being built by a CAS, interesting optimizations like the code for evaluating symbolic sums can be contemplated because the CAS provides the algebraic manipulation capabilities for doing them.

Finally, CA systems need to be designed with the understanding that the system not just provide tools for manipulating formulae (functions represented by mathematical expressions), but also functions represented by programs. One can simplify a formula using the CAS's simplify command. One should be able to simplify a program using (perhaps) the same command. This paper has alluded to what needs to be and can be done and hopefully how clean the resulting system can be for creating efficient programs.

References

[1] B. L. Gates, A numerical code generation facility for REDUCE. *Proceedings of SYMSAC '86*, pp 94–99, ACM Press, 1986.

[2] W.N. Borst, V.V. Goldman, J.A. van Hulzen, GENTRAN 90: A REDUCE package for the generation of Fortran 90 code. *Proceedings of ISSAC '94*, pp 45–51, ACM Press, 1994.

[3] M. C. Dewar, IRENA – An Integrated Symbolic and Numerical Computation Environment. *Proceedings of ISSAC '89*, pp 171–176, ACM Press, 1989.

[4] P. Capolsini and C. Gomez, MacroC and Macrofort: C and Fortran Code Generation within Maple, *MapleTech* **3** no. 1, (Birkhauser, 1996) 14–19.

[5] *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.*, Proceedings of the first SIAM Workshop on the Automatic Differentiation of Algorithms, edited by A. Griewank and G. F. Corliss, SIAM, 1991.

[6] *Computational Differentiation: Techniques, Applications, and Tools.*, Proceedings of the second SIAM International Workshop on Computational Differentiation, edited by M. Berz, C. Bischof, G. Corliss, and A. Griewank, SIAM, 1996.

[7] T. Freeman, G. Imirzian, E. Kaltofen, A System for Manipulating Polynomials Given by Straight-Line Programs. *Proceedings of SYMSAC '86*, pp 169–175, ACM Press, 1986.

[8] M.B. Monagan, R.R. Rodini, An Implementation of the Forward and Reverse Modes of Automatic Differentiation in Maple. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings of the Second International Workshop on Computational Differentiation, pp 353–362, SIAM, 1996.