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

Michael B. Monagan* and Gladys Monagan

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

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, and compiling and executing the programs from inside Maple. The tools for manipulating programs include automatic differentiation, code optimization, and the complexity analysis of a program. An application to an optimization problem from computer vision which requires a gradient computation is given.

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 automate completely the task of producing an efficient gradient program so that different models could be tried out quickly and 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 into 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' GENTRAN package [6], the work of van Hulzen et al. [2] in REDUCE, of Dewar [4], and of others. Gomez's macrofort and Capolsini's macroC [3] 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 language of the CAS before translation. This means that the computation can be executed in the CAS. This makes prototyping easier and provides a check against the final numeric code. We also provide tools for manipulating the program. These include tools for

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

(ii) optimization of the program

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

(iv) repackaging the program calling sequence and manipulating the formulae inside the program, and

(v) translation into a C or Fortran executable.

Another important advantage is that representing a computation as a program is potentially more efficient than trying to do so using formulae, because loops, local variables, and subroutine calls reduce the execution time and the space complexity of the code. We are not the first to attempt to design a system to manipulate programs in the context of CASs. In [5] Kaltofen et al. designed DAGWOOD, a system for creating and manipulating straight-line programs. The primary purpose of DAGWOOD was to compute GCDs of polynomials and to 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, manipulate and simplify these programs, ideally, as easy as it is to create, manipulate and simplify formulae.

The main application tool in our package is efficient derivative computation using automatic differentiation. See [1] and [7] for an overview of the field of automatic differentiation. Computing derivatives has many important applications in scientific computing. CASs are effective at computing derivatives; they can handle problems of modest

*This work was supported by NSERC of Canada and by Waterloo Maple Inc.

size and thus can be used reliably inside scientific applications. In contrast, CASs 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, CASs often cannot be used reliably in scientific applications.

The use of the reverse mode for automatic differentiation is more efficient than either symbolic differentiation used in CASs or the numeric approach of divided differences. 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 using the reverse mode is $O(m+n)$. The complexity for symbolic differentiation and the numerical approach of divided differences is $O(m \times n)$ in comparison. The use of automatic differentiation means that very large problems can be handled by a CAS.

2 The Code Generation Package

The CODEGEN package contains a number of components. To motivate their existence, 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 the user has created two functions f and g in two variables x and y, which are represented in the CAS by formulae.[1]

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

$$f := e^{(x+y)}\, y\, x - e^{(x+y)}\, x^2$$

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

$$g := 1 - e^{(x+y)}\, y\, x - e^{(x+y)}\, x$$

It is desired to calculate the Jacobian matrix of partial derivatives. Many users of CASs 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 numeric 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 build a program in the CAS for computing f and g. The `makeproc` command below creates a Maple procedure which on input of x and y returns a vector (a one-dimensional array in Maple) of $f(x,y)$ and $g(x,y)$. Alternatively, the user may specify that the vector is to be a parameter instead.

```
> with(codegen):
> A := vector([f,g]);
```

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

```
> 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
```

If we were writing this 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 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 [8]. The output is a program which on input of x, y constructs and returns the Jacobian matrix. In the code below, the Jacobian is the variable `grd`, a two dimensional array.

```
> J := GRAD(F,[x,y],result_type=array);

J := proc(x, y)
local dfr0, t4, t2, A1, A2, grd, t5, 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
```

[1] For readers not familiar with Maple, lines beginning with the > character are input commands to Maple. The output from Maple is centered and in italics unless it is a Maple procedure, in which case it is displayed left justified in same font as the input.

Because of the chain rule, common subexpressions have appeared in the Jacobian. Secondly, we can see that constant folding will be effective on this example. A further optimization pass after automatic differentiation is usually effective.

```
> J := optimize(J);

J := proc(x, y)
local df, t3, t7, dfr0, t8, t13, t15, grd, t2, t4, t5;
  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[3] := -t2;
  t8 := y*x;
  df[1] := -t5 + t8;
  dfr0[1] := -x - t8;
  grd := array(1 .. 2, 1 .. 2);
  t13 := df[1]*t2;
  grd[1, 1] := 2*df[3]*x + t3 + t13;
  grd[1, 2] := t7 + t13;
  t15 := dfr0[1]*t2;
  grd[2, 1] := df[3] - t3 + t15;
  grd[2, 2] := -t7 + t15;
  grd
end
```

We execute the resulting program in the CAS at $x = 1.0$ and $y = 1.0$.

```
> print(J(1.0,1.0));
```

$$\begin{bmatrix} -7.389056101 & 7.389056099 \\ -29.55622440 & -22.16716830 \end{bmatrix}$$

Next, we determine an operation count of the program for comparison with other approaches.

```
> cost(J);
```

$$22\,storage + 18\,assignments + functions + 10\,additions + 9\,multiplications$$

Now we convert the program to a C subroutine.

```
> C(J);
#include <math.h>
void J(x,y,grd)
double x;
double y;
double grd[2][2];
{
  double df[5], t3, t7, dfr0[5], t8, t13, t15, t2, t4, t5;
  {
    t2 = exp(x+y);
    t3 = t2*y;
    t4 = t3*x;
    t5 = x*x;
    t7 = t2*x;
    df[2] = -t2;
    t8 = y*x;
    df[0] = -t5+t8;
    dfr0[0] = -x-t8;
    t13 = df[0]*t2;
    grd[0][0] = 2.0*df[2]*x+t3+t13;
    grd[0][1] = t7+t13;
    t15 = dfr0[0]*t2;
    grd[1][0] = df[2]-t3+t15;
    grd[1][1] = -t7+t15;
    return;
  }
}
```

In translating to C we make assumptions about the types of the variables. Notice also that the matrix of partial derivatives is passed as a parameter in the above C code. We will discuss these two points later. We compile the Maple code in C and execute it on the data $x = 1.0, y = 1.0$ as a practical check that the compiled program is correct.

```
> cJ := compile(J,precision=single):
compile:   Compiling and linking ./J.c and ./mainJ.c .
./J.c:
./mainJ.c:
> cJ(1.0,1.0);
```

$$\begin{bmatrix} -7.389056 & 7.389056 \\ -29.55622 & -22.16717 \end{bmatrix}$$

If we are going to use J with a subroutine from a numeric library package, that subroutine will very probably use a different calling sequence than our J program uses. It will use a vector for the independent variables x and y and it will probably pass a pointer to the storage for the Jacobian matrix in which to write the Jacobian entries rather than dynamically allocating storage each time the program is called. The user may effect these transformations explicitly as follows.

```
> J := packargs(J,[x,y],X):
> J := makeparam(grd::'array'(1..2,1..2),J):
> J := makevoid(J);

J := proc(X::array(1 .. 2), grd::array(1 .. 2, 1 .. 2))
local df, t3, t7, dfr0, t8, t13, t15, t2, t4, t5;
   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[3] := -t2;
   t8 := X[2]*X[1];
   df[1] := -t5 + t8;
   dfr0[1] := -X[1] - t8;
   t13 := df[1]*t2;
   grd[1, 1] := 2*df[3]*X[1] + t3 + t13;
   grd[1, 2] := t7 + t13;
   t15 := dfr0[1]*t2;
   grd[2, 1] := df[3] - t3 + t15;
   grd[2, 2] := -t7 + t15;
   RETURN()
end
```

At some point the user may wonder, whether, after such a sequence of 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);
```

$$(-x^2 + y\,x)\,e^{(x+y)} - e^{(x+y)}\,y\,x + e^{(x+y)}\,x^2$$

```
> simplify(");
```

$$0$$

We emphasize the following points about this example. Firstly, 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

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 on 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);
```

$$t := 1 + x + \frac{1}{2}\,x^2 + \frac{1}{6}\,x^3 + \frac{1}{24}\,x^4 + \frac{1}{120}\,x^5 + \frac{1}{720}\,x^6 + \frac{1}{5040}\,x^7$$

The 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 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.

We give an alternative approach which a user of a CAS might take. It is surely reasonable to proceed instead 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 );
```

$$p := \begin{cases} \dfrac{1}{\mathrm{e}(-x)} & x < 0 \\ \displaystyle\sum_{i=0}^{7} \frac{x^i}{i!} & x < .5 \\ \mathrm{e}(\frac{1}{2}\,x)^2 & \mathit{otherwise} \end{cases}$$

The piecewise function definition is a symbolic representation of a conditional statement. The `Sum` function can be thought of as a symbolic representation of a for loop. Note that the capital S distinguishes the `Sum` from the `sum` command in Maple which would 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 translate properly this symbolic sum 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 an engineer with some assurance that the compiled code is correct.

```
> ec := compile(e, precision=single):
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

We emphasize the following points about this second example. Users will expect that if symbolic representations for sums and products are provided by the CAS, that the resulting code that is created is at least as good as one would write by hand.

3 Details about the Package

Our approach to the design of the CODEGEN 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. Some of the commands that we illustrated, `optimize`, `cost`, `C`, already exist in Maple. We have been updating these commands in Maple to work for Maple procedures as well as formulae.

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 CAS's native language and if necessary, the CAS needs to be 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. Specifically we allow Fortran style parameter declarations for arrays as in

```
proc(n::integer,A::array(1..n,1..n))
```

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. What we do instead is first convert the Maple procedure into an intermediate representation, a DAG, which is convenient to work with. We apply the program-to-program transformations on this intermediate representation, then convert the result back to a Maple procedure. The intermediate representation allows us to attach type information and other information to the program. 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 [8]. It provides facilities for gradients, Jacobians and Hessians of Maple programs. The Maple programs may use any function which Maple knows how to differentiate. For unknown functions it 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

The code optimizer is designed to perform optimizations on programs which can be done in expected linear time in the size of the input program. 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, optimizations such as common subexpression elimination and constant folding can be done in linear time. For common sub-expression optimization we make use of Maple's built-in hashing of formulae to identify all distinct subexpressions in linear time. Note, by *subexpressions* we mean *suboperands* in Maple. For example, in Maple, `x` is a suboperand of `w*x*y*z` but `x*y` is not. For constant folding, we use assignment and evaluation to replace variables in formulae in linear time.

3.5 The cost facility

The example previously given for the `cost` command 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));
```

$$
\begin{aligned}
c := {} & l\,n\,(1 + (\begin{cases} 0 & -1+m<0 \\ -1+m & otherwise \end{cases})\,)\ additions \\
& + l\,n\,(\begin{cases} 0 & -1+m<0 \\ m & otherwise \end{cases})\ multiplications \\
& + l\,n\,((\begin{cases} 1 & -1+m<0 \\ m & otherwise \end{cases}) + 1)\ assignments \\
& + 6\ storage
\end{aligned}
$$

This is an exact operation count. To obtain a simplified asymptotic count one may do

```
> assume(m>1);
> assignments := 0: # don't count assignments
> storage := 0: # don't count storage
> collect(c,[l,n,m]);
```

$$(additions + multiplications)\,m\,n\,l$$

3.6 Building and manipulating programs

The `makeproc` command is used to build a program with given parameters. We can build a program to evaluate a single formula or an array of formulae. The user may specify whether an array is to be created dynamically inside the program and returned as the value of the program, or passed as a parameter. We use equations to represent assignments and a list of expressions to represent a sequence of

computations. Symbolic sums and products, and piecewise function definitions can be used to encode some conditional statements and loops in formulae. More complicated programs must be built using the intermediate representation described previously.

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 when translating into C or Fortran. Type information for parameters can be specified for a Maple procedure in the Maple code. This is usually sufficient to determine the types of all other variables in the program. We can also deduce the the types of certain expressions in Maple programs from their context. For example, the type of `i` in `A[2*i-1]` must be an integer if `A` is known to be an array. Where there is no information we assume floating point by default. The user may also override any type analysis by specifying the type of a variable to be used in translation.

The compile command constructs a C executable and 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 certain details of the application. We provide the following 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. 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 faster — the vehicle drives at 60 kmph along the railway 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. In the comparison, 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.

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. We also require that the code be fast because speed is critical in this application.

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 Maple code below, the variables $\bar{x}$ and $\bar{y}$ are the coordinates of the resulting point. The function f calculates the square of the distance between $(\bar{x}, \bar{y})$ and an arc 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 d1, d2, s, cr0, t1, t3, grd, t8, t7,
t10, dfr0, t19, t21;
   s := sin(theta);
   cr0 := cos(theta);
   t1 := x + C;
   t3 := y + D;
   t7 := h - t1*cr0 - s*t3;
   d1 := t7^2;
   t8 := k - t3*cr0 + 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*cr0;
   grd[2] := t19*cr0 + t21*s;
   grd[3] := - dfr0[2]*s + dfr0[1]*cr0;
   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 that the automatic differentiation does not recognize the special nature of the conditional statement and consequently produces what looks like poor code. This is typical of what happens with many of the program-to-program transformations. We find that we are always working on code optimization. Here is the first version

```
> 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 s, ybar, xbar, d, r, dfr0, cr0, grd;
  s := sin(theta);
  cr0 := cos(theta);
  xbar := (x + C)*cr0 + (y + D)*s;
  ybar := (y + D)*cr0 - (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]*cr0;
    grd[2] := dfr0[4]*cr0 + 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]*cr0;
    grd[2] := dfr0[4]*cr0 + 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 translated into C using single precision.

```
> 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:

> 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 t3, t10, t8, t13, t16, cr0, s,
      dfr0[5], r, t14, t1;
  {
    s = sinf(theta);
    cr0 = 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*cr0;
    grd[1] = t14*cr0+t16*s;
    grd[2] = -dfr0[1]*s+dfr0[0]*cr0;
    return;
  }
}
```

The `sinf`, `cosf`, and `sqrtf` function calls in the C code are for single precision. The gradient vector `grd` is returned through a parameter.

This code is clearly not limited to the computer vision application. This final generated code is close to the best possible 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 task of obtaining a gradient this was easy to change.

In the application, the CAD model for the rail consists of a sequence of lines and arcs. It is a piecewise function. Determining which piece the data belongs to is done after the values of C, D and θ have been updated in the iteration. This is not described here.

5 Conclusion

The way we have designed our code generation package is to use complete programs as the primary data representation for a computation and to make all code manipulations as program-to-program transformations. This leads to a very clean design in which the codes can be prototyped easily in the CAS. Also, from a complexity viewpoint, the use of programs to represent a computation is more general and more efficient than formulae. Although the components of our system worked smoothly together in the real application from computer vision that we presented, each of the main components has its 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. Adding symbolic constructions like this to a CAS would make it easier to build certain programs.

- Program Limitations

  Not all Maple programs can be converted to C or Fortran code, or differentiated by the GRAD automatic differentiation software. This in itself is not a major problem. However, 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 program of most program-to-program transformations needs further optimization. At present our optimizer does not handle loops and does not do in-lining of code. The reader should understand that just as the `simplify` command is arguably the most important command in a CAS for manipulating formulae, the program optimization facility is very important when manipulating programs.

- Maple Compiler

  The compiler that we have developed is simple. It converts the Maple code to C, constructs a C executable and communicates data via files. The resulting code is slow for two reasons. Maple arrays are unsuitable for large floating point computations and the data communication overhead may be high. A new hardware floating point data structure is being incorporated into Maple to resolve the first problem and a dynamic linking mechanism is being added to resolve the second problem.

Finally, CASs 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 can be done and hopefully how clean the resulting system can be for creating efficient programs.

References

[1] Berz, M., Bischof, C., Corliss, G., and Griewank, A., Eds. *Computational Differentiation: Techniques, Applications, and Tools* (1996), Proceedings of the Second SIAM International Workshop on Computational Differentiation, SIAM.

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

[3] Capolsini, P., and Gomez, C. MacroC and Macrofort: C and Fortran Code Generation within Maple. *MapleTech 3*, 1 (1996), 14–19.

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

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

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

[7] Griewank, A., and Corliss, G. F., Eds. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application* (1991), Proceedings of the First SIAM Workshop on the Automatic Differentiation of Algorithms, SIAM.

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