Generation and Optimization of Numerical Programs by Symbolic Mathematical Methods

Richard Fateman
Computer Science Division, EECS
University of California, Berkeley

RIMS: Research Institute for Mathematical Sciences
Kyoto University
Program Transformation, Symbolic Computation
and Algebraic Manipulation

Outline

1. Introduction
2. Symbolic Mathematics Components
3. Symbolic Manipulation Systems as Glue

PART I: INTRODUCTION

- Symbolic Computation is different from Numeric Computation
  - **Programs** are **data**
    - Lisp
    - Scripting languages (programs are strings)
- Computer Algebra Systems (**CAS**) go further
  - **Programs** are **data** are **mathematical expressions**
  - Usually provide interactive environments
    - Mathematica, Maple, Macsyma, Axiom, Reduce, MuPad, many academic projects

Other aspects of CAS

- Commercial CAS are also knowledge representation systems
  - Notebooks
  - Libraries
  - Networking (TILU at Berkeley)
- CAS are being connected with other commercial environments
  - MathCAD+Maple, (numerics)
  - Matlab+Maple, (numerics, matrix computation)
  - Scientific Word + Maple or Mathematica (editor)
  - new Mathematica text editor
  - Academic projects (e.g. SENAC: Macsyma+ NAG library)

CAS: Data Types and Operations

- Symbolic Functions and expressions involving them
  - + * log, sin, Bessel, …
  - A*x+b*cos(y) – usually “algebraic trees”
  - Differentiation, Integration, Simplification, Approximation (series, expansions, economization…)
  - Valuation (convert an expression to a “number”)
- Extensible domains (real numbers, polynomials over the integers …)
- Arbitrary structures for numbers, symbols, strings, tables, trees
  - Input as strings
  - Output as strings, plots, pictures, typeset equations, web messages

PART II: SYMBOLIC COMPONENTS

- 3 Models: *Environment* vs *Toolkit* vs *Network*
  - "Complete" Environments:
    - Macintosh
    - Microsoft "interconnected" applications
    - Advanced workstation environments
  - Toolkit:
    - UNIX in all its variations
    - Subroutine libraries (netlib, etc.)
  - Networked model:
    - "Intelligent agents" on some network
    - Popular view with Java, RMI, CORBA

We need to Re-use Symbolic Components

- System Objectives:
  - Doing symbolic mathematics in support of numerical computation
  - Doing what other systems do (numerically, graphically) plus more

“UNIX” style toolkits are not the way to start

- Symbolic systems have environment support issues [storage model, run-time semantics for mathematical expressions, libraries for numerical or graphical]
- Once we have this base we can add auxiliary modules (CAS + library)
- We can try to use webcomponents via a “shallow” web interfaces (e.g. OpenMath, MathML). No persistent state is shared.

What do we mean by program that manipulate programs?

- Assemblers, interpreters,
- (pre-) Compilers, macro-expansion, etc.
- Advice takers
- The symbolic view is that

expressions  **programs**  **data**

Example of Advice (in Lisp)

To avoid complex results from **`sqrt`** one can ‘‘advise” **`sqrt`** that if its first argument is negative, it should instead print a message and replace the argument by its absolute value.

```
(advise sqrt :before negativearg nil
    (unless (>= (first arglist) 0)
        (format t
         "sqrt given negative number. we take (sqrt(abs ~s))"
           (first arglist))
        (setf arglist (list (abs (first arglist))))))
```

Another Example of Advice

Pattern matching on arguments is the way this is done in some languages like Prolog, CLOS, and (here) in Mathematica

```
Mysqrt[h_]  := Sqrt[h]              (*Default case*)
Mysqrt[-h_] := Sqrt[Abs[h]]         (* works for Mysqrt[-r] *)
Mysqrt[h_]  := Sqrt[Abs[h]] /;h<0   (* works for Mysqrt[-9] *)
```

But it’s not easy to manipulate Fortran or C

We are mostly restricted to generating text to be inserted as into Fortran or C code.

In this next example we start with a Fortran program, convert to Lisp, manipulate the program as data, and convert back to Fortran.

While FORTRAN (77, 90) may provide a standard for text, any compiler will convert such code to the moral equivalent of Lisp as soon as it is parsed. This intermediate language is essential for optimization, code generation etc.

Start with Fortran: Bessel Function evaluation (from *Numerical Recipes in Fortran*)

```
 …(selected lines…)

      DATA Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9/0.3989422D0,-
0.3988024D-1,
     *    -0.362018D-2,0.163801D-2,-0.1031555D-1,0.2282967D-
1,
     *    -0.2895312D-1,0.1787654D-1,-0.420059D-2/
       ...
        BESSI1=(EXP(AX)/SQRT(AX))*(Q1+Y*(Q2+Y*(Q3+Y*(Q4+
     *         Y*(Q5+Y*(Q6+Y*(Q7+Y*(Q8+Y*Q9)))))))
       ...
```

Equivalent Bessel Function evaluation in Lisp (1)

```
(setf
bessi1
(* (/ (exp ax) (sqrt ax))
   (poly-eval y
    ( 0.39894228d0 -0.3988024d-1 -0.362018d-2  0.163801d-2
     -0.1031555d-1  0.2282967d-1 -0.2895312d-1 0.1787654d-1
     -0.420059d-2))))
```

Just rearranging the coefficients. What is poly-eval function?

Bessel Function Evaluation in Lisp (2)

```
(let* ((z (+ (* (+ x -0.4474202468916 62d0) x) 0.555557444584 1143d0))
       (w (+ (* (+ x -2.1804403631654 97d0) z) 1.759291809106734d0)))
 (* (+ (* x (+ (* x (+ (* w (+ -1.7459865688143 45d0 w z))
                       1.213871280862968d0))
               9.493961562542 4d0))
       -94.97291570945 98d0)
    -0.00420059d0))
```

Poly-eval can do an in-line expansion that is then compiled.

Advantages include fewer multiplies (6, not 8) one more add (9, not 8) but somewhat more overlap for superscalar processor.

Bessel Function evaluation in Lisp (3)

Can we/ should we do this? Generally we need a “license” for rearrangement of code.

If the programmer/ designer really wanted *EXACTLY* this sequence of computations, we would have to respect that.

Why a higher-level model is nicer than a program if the algorithm can then be improved to better code: Are the Bessel function coefficient numbers approximations to better values?

Bessel Function evaluation in Lisp (4)

Conversion back to text for Fortran looks something like this…

```
“S=(X*(X*(W*(-1.74598656881434 5d0+W+Z)
+1.2138712808629 68d0)+9.493961562542 4d0)-
94.9729157094598d0)* -0.00420059d0”
```

Though we would question why this should be compiled better by Fortran than Lisp.

Another example: the Euler Equation

The Euler equation is a favorite benchmark of Celestial Mechanics symbolic calculation programs.

$$E = u + e \sin(E)$$

as commonly solved iteratively (for small e) gives this
4th order expansion for $E = u + A$

The Euler Equation, 4th order solution

E= u+A where A is

$$A_4 = \frac{e^4 \sin(4U)}{3} + \frac{3e^3 \sin(3U)}{8} + \frac{(12e^2 - 4e^4)\sin(2U)}{24} + \frac{(24e - 3e^3)\sin(U)}{24}$$

The Euler Equation, 4th order solution

In Fortran as rendered by Mathematica 2.0 (buggy):

```
      FortranForm=
     -  (24*e - 3*e**3)*Sin(U)/24 + (12*e**2 -4*e**4)*Sin(2*U)/24+
     -   3*e**3*Sin(3*U)/8 + e**4*Sin(4*U)/3
```

Note: in Fortran, 1/3 is computed as 0; this formatting is dangerous

The Euler Equation, 4th order solution

```
 in Mathematica … if we call… Expand[N[%]]

FortranForm=
     -  1.*e*Sin(U) - 0.125*e**3*Sin(U) + 0.5*e**2*Sin(2.*U) -
     -   0.166666666666666*e**4*Sin(2.*U) + 0.375*e**3*Sin(3.*U) -
     -   0.333333333333333*e**4*Sin(4.*U)
```

What are the precisions of the constants? Why do we multiply by 1.?

The Euler Equation, 4th order solution

Maple produces

```
 t0 = e**4*sin(4*U)/3+3.0/8.0*e**3*sin(3*U)+(12*e**2-4*e**4)*sin(2*
#U)/24+(24*e-3*e**3)*sin(U)/24
```

or after floating-point conversion using evalf

```
 t0 = 0.3333333E0*e**4*sin(4.0*U)+0.375E0*e**3*sin(3.0*U)+0.4166667
#E-1*(12.0*e**2-4.0*e**4)*sin(2.0*U)+0.4166667E-1*(24.0*e-3.0*e**3)
#*sin(U)
```

What are the precisions of the constants? Do we really want to compute **`e**4`** repeatedly?

The Euler Equation, 4th order solution

After `convert(expr,horner,[e])` Maple produces:

```
t0 = (sin(U)+(sin(2*U)/2+(3.0/8.0*sin(3*U)-sin(U)/8+(-sin(2*U)/6+s
     #in(4*U)/3)*e)*e)*e)*e
```

Somewhat inconsistent.. 3.0/8.0? But close…

We don't compute **`e**4`** repeatedly, but what about exploiting the dependency relationship between sin(u) and sin(2u)?

Sin and Cos computation

```
s  := sin(u)
c  := cos(u)
s2 := 2*s*c   ;;   this is sin(2u)
c2 := 2*c*c-1  ;;  this is cos(2u)
s3 := s*(2*c2+1);; this is sin(3u)
s4 := 2*s2*c2  ;;  this is sin(4u)  etc
```

There's an even better way for higher order, requiring only 2 mults and 2 adds for each new sin/cos pair.

Sin and Cos computation even faster…

```
k1=sin(u), k2=4*k1^2,
s[0]=sin(u),s[1]=1, c[0]=1, c[1]=cos(u)
```

The inner loop for I>1 is

```
s[I]  := s[I-2]+c[I-1]     //sin(I*u)/sin(u)
c[I]  := c[I-2]-k2*s[I]    //cos(I*u)
```

When you need sin,

```
  Sin(n*u) is  k1*s[n]
```

***No computer system comes close to recognizing this automatically*.**

Computing Derivatives

Many students who having studied the use of a ‘‘symbolic” language like Lisp will have seen differentiation as a small exercise in tree-traversal and transformation. They will likely view closed-form symbolic differentiation as trivial, if for no other reason than it can be expressed in a half-page of code:

Derivatives in Lisp; irrelevant though

```
(defun d(e v)(if(atom e)(if(eq e v)1 0)
(funcall(or(get(car e)'d)#'undef)e v)))
(defun undef(e v)`(d,e,v))
(defun r(op s)(setf(get op'd)(compile()`(lambda(e v)(let((x(cadr e)))
(list'*(subst x'x',s)(d x v)))))))
(r'cos'(* -1(sin x)))
(r'sin'(cos x))
(r'exp'(exp x))
(r'log'(expt x -1))
(setf(get'+'d)#'(lambda(e v)`(+,@(mapcar #'(lambda(r)(d r v))(cdr e)))))
(setf(get'*'d)
#'(lambda(e v)`(*,e(+,@(mapcar #'(lambda(r)`(*,(d r v)(expt,r -1)))(cdr
e))))))
(setf(get'expt'd)#'(lambda(e v)`(*,e,(d`(*,(caddr e)(log,(cadr e)))v))))
```

Derivative of a program?

Viewing a subroutine as a manifest representation of a mathematical function, we can try to push this idea as far possible.

The alternative is using a ‘‘numerical’’ derivative of f(x) at a point c computed by chosing some small Delta and computing (f(c+Delta)-f(c))/Delta.

Numerical differentiation yields a result of unknown, but probably low, accuracy.

(Useful literature has developed in the last 2 decades: ADIFOR at Argonne National Lab for example)

Other closed forms from CAS

```
Integral of 1/(z^5+1)  in Fortranform from Mathematica

 (Sqrt((5 - Sqrt(5))/2.)*
     -     ArcTan(2*Sqrt(2/(5 - Sqrt(5)))*
     -       ((-1 - Sqrt(5))/4. + z)))/5. +
     -  (Sqrt((5 + Sqrt(5))/2.)*
     -     ArcTan(2*Sqrt(2/(5 + Sqrt(5)))*
     -       ((-1 + Sqrt(5))/4. + z)))/5. + Log(1 + z)/5. -
     -  ((1 - Sqrt(5))*Log(1 - ((1 - Sqrt(5))*z)/2. +
     -       z**2))/20. -
     -  ((1 + Sqrt(5))*Log(1 - ((1 + Sqrt(5))*z)/2. +
     -       z**2))/20.
```

This is probably OK. What about 1/(z^64+1) vs numerical integration?

Exact or high-precision values

Arithmetic on objects of variable size is offered:
Most CAS support exact integer and rational computing.
Rational domain cannot handle exponential, log, trignometric function computing → arbitrary-precision floats.

exp(pi*sqrt(163))=262537412640768743.99999999999992500726.

This last expression is not an integer, but it is very close.

Documents/ Electronic Notebooks

Output as TeX, html, xml, mathml,
Notebooks (Mathematica, Maple, Macsyma …)
Spreadsheets (Theorist, MathCAD)
Graphics into AVS, other graphics packages

If the purpose of computing is insight, the documentation and analysis must play a role in the problem-solving environment.

PART III: GLUE

How do we communicate, store scientific programs/ math / program proofs?

Computer text files?

Output as TeX, html, xml, mathml,

Notebooks (Mathematica, Maple, Macsyma …)

Spreadsheets (Theorist, MathCAD)

Graphics into AVS, other graphics packages

Why should computer algebra systems work better than (say) Perl or Tcl/Tk or Python or other scripting languages?

You can’t talk effectively about math or programs without foundations

Experiments have shown that it is nearly impossible for two people to express mathematics over the telephone.
If we treat programs like formal systems, a foundation must be available for making sure that syntactically valid and semantically consistent communication is going on.

CAS provide a basis for this communication.

All that Perl tells you is that you have a “string”.

Development of optimizing code makes sense in CAS

We hope we have shown that such transformation are quite plausible, and in fact reasonably easy --- if we know what transformations are needed --- in a suitable symbolic computation system.

Compare this to a “Compiler back end” written in C/C++ etc where the notion of a polynomial or a cosine is so thin.

Trends for producing quality scientific software

Problem Solving Environments

- Scientific code production and testing
- Integration of user interfaces to applications
- Higher importance on
  - reliability,
  - correctness,
  - verification
- Computer symbolic mathematical computation
  - an essential foundation
  - a collection of tools.