Computation of functions and their derivatives in content

V.V. Levitin

Department of Analysis, Algebra and Geometry

Computation of functions and their derivatives in CONTENT

V.V. Levitin

Institute of Mathematical Problems of Biology,
Russian Academy of Sciences, Pushchino, Moscow Region,
142292 Russia
&
CWI,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

Scientific software for investigation of dynamical systems usually allows the user to specify systems in the form of functions in some language. There are several problems related to this: how to compute derivatives of these functions (since numerical algorithms need to compute first and higher order derivatives) and how to convert these functions and their derivatives to a machine suitable form. The key issues are minimal effort required from the user to manipulate functions and high efficiency in calculations that extensively compute functions and their derivatives. This work presents some solutions to these problems which are incorporated in CONTENT, a new program for bifurcation analisys of dynamical systems. Derivatives are built during a three-stage process by means of transforming C source given by the user for right-hand sides into a C++ program which constructs another C program for computing derivatives. A special C++ class with overloaded operators and functions is used to produce this C program during the execution of the C++ program. A description of a dynamical system (i.e., functions and their derivatives) is compiled and linked as a dynamic library during a CONTENT session. This does not require linking with the rest of software. Dynamic libraries for different systems are created, loaded, and functions in them are accessed without necessity to quit CONTENT.

1. INTRODUCTION

This work describes some problems and their solution occuring in the develpoment of CONTENT (Kuznetsov & Levitin 1995).

CONTENT is intended to be an integrated environment for various algorithms for analysis of ordinary differential equations and maps with primary emphasis on their bifurcation analysis. These algorithms require a large number of computation of functions that define dynamical system and their derivatives.

One of the requirements to CONTENT is to provide a user with the ability to specify and manipulate equations in a comfortable way. On the other hand this should be combined with the possibility to compute functions given by the user and their derivatives efficiently.

Here we consider decisions made in CONTENT development to meet these requirements.

Section 2 describes the transformation of a specification of a dynamical system given by the user into a C function. A way in which derivatives of functions are constructed is presented in Section 3. Then several complications induced by a more general way of dynamical system definition are discussed in Section 5. Section 6 sketches how CONTENT deals with the problem of conversion of source text of functions and their derivatives into a machine form suitable for computations. Some concluding remarks are given in Section 7.

2. PROCESSING OF SPECIFICATIONS OF DYNAMIC SYSTEMS

CONTENT is an integrated environment, that is it provides the user with all support and tools needed to investigate dynamical system. In particular, it allows to create, edit, and delete the specification of dynamical systems during a session and does not require the user to run a compiler and a linkage editor (even in any semi-automatic form like running developer-provided batch files). Neither it requires to re-link itself with newly created object files. An implementation that completely hides processing of functions and their derivatives from the user will be discussed later.

The user asks CONTENT to create a new or modify an existing dynamical system by selecting an appropriate item from a menu. In respond to this CONTENT shows a special window with several text controls and allows the user to define or edit a dynamic system by giving names for time, phase (state), and parameter variables along with equations. In the simplest case, each equation is an assignment statement where left-hand side is the name of a phase variable followed by an apostrophe denoting time derivative for ODEs and the image for iterated maps. Right-hand side of the assignment is an expression constructed from previously defined names, operators, constants, and standard function calls. Figure 2.1 shows an example of specification of a dynamical system.
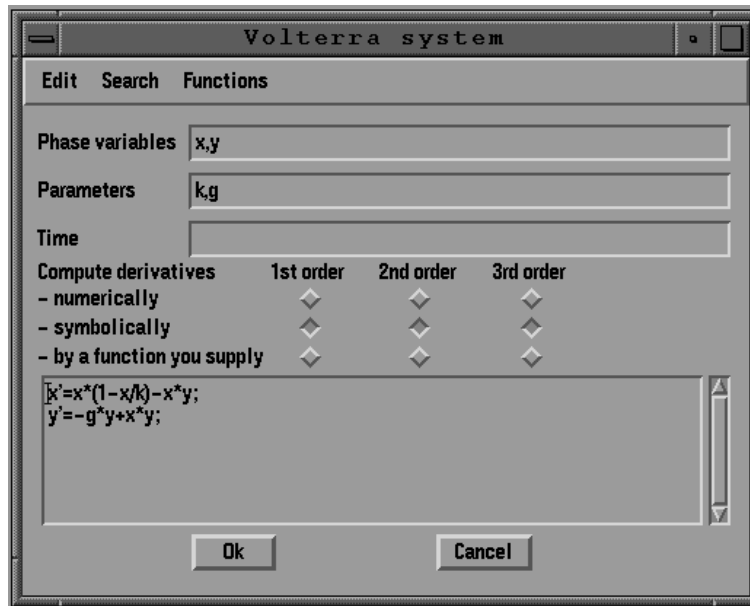
Figure 2.1: A window where the user specify dynamical systems.

In fact the specification given by the user is used by CONTENT to construct C. This is done as follows. A text of equations becomes a body of a C function which will be called by numerical algorithms when they need the values of right-hand sides (RHS) to be computed. As the user deals with one dynamic system at each moment the name of the function and number and types of its parameters are always the same. Thus, the user given text

```
x'=x*(1-x/k)-x*y;
y'=-g*y+x*y;
```

first becomes a C function

```
void _Rhs(double *_phase, double *_par, double *_rhs) {
  x'=x*(1-x/k)-x*y;
  y'=-g*y+x*y;
}
```

where _phase is a pointer to values of phase variables, _par is a pointer to values of parameters of dynamic system, and _rhs is a pointer to output array of computed RHS values. Underscores in the function and parameter names are used to avoid collisions with names given by the user (obviously, the user is not allowed to use underscore in names he provides). Then names given by the user are used to generate #define directives that map these names onto components of _phase, _par, and _rhs. (There is a small technical problem with apostrophes which are not allowed to be parts of names in C. They are merely substituted by underscores.) After this second step the source looks like

```
#define x  (_phase[1])
#define y  (_phase[2])
#define k  (_par[0])
#define g  (_par[1])
#define x_ (_rhs[0])
#define y_ (_rhs[1])
void _Rhs(double *_phase, double *_par, double *_rhs) {
  x_=x*(1-x/k)-x*y;
  y_=-g*y+x*y;
}
```

Note that _phase[0] is reserved for the time variable.

After the source for the RHS computation is built, CONTENT tries to construct the source for derivatives.

3. WAYS TO COMPUTE DERIVATIVES

Algorithms of bifurcation analysis need to compute first and higher order derivatives of the RHS with respect to phase variables and parameters. Let us consider how this could be done.

The first and most obvious way is *numerical differentiation* in which derivatives are approximated by finite differences. This is easy to implement and each particular algorithm may compute only those derivatives it needs, thus reducing computational time and used memory. Another advantage is that derivatives can always be computed regardless of the complexity of the RHS. The well known disadvantage of this approach is its inaccuracy compared with 'exact' derivatives. In some cases it also may need different step sizes for different variables and this may be a problem even when the number of equations is small.

The second possible way of dealing with derivatives of the RHS is *user-supplied derivatives*, i.e. to require the user to provide routine(s) for their computation. This allows one to compute derivatives with the highest possible accuracy and does not require much efforts from the developer. From the user's point of view this approach cannot be regarded as user-friendly. Although it is possible that the user may want to specify the derivatives himself, in general he would like to avoid extra symbolic manipulations either by hand or by means of some software (for example, see Griewank (1989), Griewank & Corliss (1991), Corliss (1992)).

The third solution to the problem in question is to implement some kind of *automatic differentiation*. That means that the source text of routines computing derivatives is constructed from the system's specification given by the user by some embedded algorithm. This is a very good solution for the user because it completely frees him from caring about derivatives and provides accurate values of derivatives. At the same time this approach requires much more effort from the developer to implement it.

The straightforward implementation of automatic differentiation could be a parsing algorithm augmented with routines for each syntax rule. The idea is to construct a source text of derivatives propagating it through a kind of syntax tree in the same way as a compiler parses source text and builds its internal form. Such a parser could be rather complex for a language like C.

We would like to avoid parsing and related problems. This may be achieved by using C++ facility to overload (redefine) such elementary operations as addition, multiplication, etc.

This idea was implemented by Briggs (1993). It could be sketched as follows. Suppose we have a C program that computes some quantities (dependent variables) from other quantities (independent variables) and we would like to transform it to a program that computes their first order partial derivatives, for example:

```
main() {
  double x,y,z,u;
  ...
  u=exp(x/(y*z))+cos(x*y*z);
  ...
}
```

Here `x`, `y`, and `z` are independent variables and `u` is a dependent one.

Let us define a C++ class with its constructor and overloaded functions for operators and standard functions (all fields and functions irrelevant to the discussion are omitted):

```
class AD {
  double value;
  double* grad;
  ...
  friend AD operator*(const AD&, const AD&);
  ...
  friend AD exp(const AD&);
  ...
};
extern int dim;
AD(int varno=0) {   // constructor
  grad=new double[dim+1];
  for (int i=1; i<=dim; i++) grad[i]=0;
  if (varno>=0 && varno<=dim) grad[varno]=1;
};
...
inline AD operator*(const AD& u, const AD& v){
  AD w; int i;
  w.value=u.value*v.value;
  for (i=1; i<=dim; i++)
    w.grad[i]=u.grad[i]*v.value+u.value*v.grad[i];
  return w;
}
...
AD exp(const AD& v){
  AD w; int i;
  w.value=exp(v.value);
  for (i=1; i<=dim; i++)
    w.grad[i]=w.value*v.grad[i];
  return w;
```

```
}
...
```

Here `dim` is a global variable whose value gives the total number of independent variables. Each variable of type `AD` contains its value and values of all its first order derivatives with respect to independent variables. The parameter `varno` of the constructor means 'sequential number of an independent variable' and must be non-zero for such variables. This ensures that the derivatives of an independent variable are all zeros except for the derivative with respect to itself which is one. Overloaded operators (like one showed above for '*') compute their results together with values of all partial derivatives.

Now transform our C program to another C++ program according to following rules. Declare integer variable `dim` and assign the number of independent variables to it. Replace declaration of each dependent variable `double vd` by `AD vd`. Replace declaration of each independent variable `double vi` by `AD vi(i)`, where i stands for the ordinal number of the variable. In such a way our example would be transformed to:

```
int dim;
main() {
  AD x(1),y(2),z(3),u;
  dim=3;   // we have three independent variables
  ...
  u=exp(x/(y*z))+cos(x*y*z);
  ...
}
```

After an assignment to `u` its `grad` field will contain all partial derivatives with respect to `x`, `y`, and `z`.

It is clear that this implementation could be easily extended to compute derivatives of higher orders. It is also clear that it could be incorporated into any numerical program with moderate amount of effort.

Despite all advantages, this approach has one but very important disadvantage: low efficiency. Although we did not make any extensive measurements, derivatives of several 'randomly' taken functions (of five to ten variables) took more then twenty times to be computed than 'hand-derived' derivatives of these functions. An implementation of this approach in CONTENT would considerably slow down numerical algorithms and therefore was rejected.

4. AUTOMATIC CONSTRUCTING OF DERIVATIVES

In an attempt to combine advantages of the above approach with efficiency we came to the following idea. Let us define another C++ class along with overloaded operations and standard functions in such a way that a modified C++ program does not compute values of derivatives during its run but constructs another C program, which in its turn will compute the derivatives. So the scheme is as follows.

- Modify user given specification of RHS in a way described above (that is, modify declarations of variables and parameters) thus obtaining the source text of a C++ program.

- Compile this C++ program, link it with an object module that contains overloaded operations and standard functions, and then run it once. This results in creating another C program for the derivatives.

- Compile and link the program constructed in the previous step. It will be used by numerical algorithms for computation of derivatives.

Discussion of advantages and disadvantages of this approach will follow after more detailed description of the C++ class and functions associated with it.

4.1 `Gradient` *class*

The declaration of a class and functions associated with it are rather complex and lengthy, so we present it in a simplified and schematic way.

The class called `Gradient` has fields for ASCII texts of an expression and its derivatives:

```
class Gradient {
  char *Expr;     // text of an expression
  char **Der1;    // texts of 1st order derivatives
  char **Der2;    // texts of 2nd order derivatives
  char **Der3;    // texts of 3rd order derivatives
  VarKind Kind;
  int n3,n2;
  ...........
  public:
  Gradient(char *name=NULL,
           int varno=0,
           VarKind kind=DEF);  // constructor
  ...........
  friend Gradient operator*(const Gradient &f,
                            const Gradient &g);
  ...........
};
```

Type `VarKind` is defined as

```
typedef enum {DEF,INDEP,TEMP,DEP} VarKind;
```

and it is used to specify the kind of variables: independent (`INDEP`), dependent (`DEP`), intermediate (`DEF`, these are implicitly created during C++ expressions calculation), or temporary (`TEMP`, discussed later). Integers `n2` and `n3` hold the number of second and third order derivatives, respectively.

A constructor for this class has several parameters: the name of a variable, its kind, and ordinal number for independent variable which must be zero for all other variables. It allocates memory for various strings and assigns values to them (note default values for parameters shown above):

```
Gradient::Gradient(char *name, int varno, VarKind kind) {
  Kind=kind;
  Expr=StringSave(name);
  n3=sub3_(_Niv_-1,_Niv_-1,_Niv_-1)+1;
  Der3=(char **)MemNew(n3*sizeof(char *));
  n2=sub2_(_Niv_-1,_Niv_-1)+1;
  Der2=(char **)MemNew(n2*sizeof(char *));
  Der1=(char **)MemNew(_Niv_*sizeof(char *));
  if (varno) Der1[varno-1]=StringSave("1");
}
```

Here is a brief description of the functions and variables used in the constructor and other functions related to the class. `MemNew` allocates memory (just like standard `malloc` does), fills it with zeros, and returns a pointer to it. All functions that deal with texts of expressions and derivatives treat the `NULL` pointer as string `"0"`. `StringSave` makes a copy of a string using `MemNew`. The global variable `_Niv_` contains the number of independent variables; its value is the sum of the number of phase variables and the number of parameters of a dynamical system. Functions `sub2_` and `sub3_` map pairs and triples of ordinal numbers of independent variables to ordinal numbers of derivatives of second and

third orders, respectively, with respect to these variables. For example, `sub2_(2,3)` gives the subscript in array `Der2` of the second order derivative with respect to the third and forth independent variables. Here these functions are used to obtain total numbers of second and third order derivatives. These functions are defined as follows:

```
int sub2_(int i, int j) {
  return i+j*(j+1)/2;
}
int sub3_(int i, int j, int k) {
  return i+j*(j+1)/2+k*(k+1)*(k+2)/6;
}
```

Note that they map one-to-one sets of pairs `<i,j>` with $i \leq j$ and triples `<i,j,k>` with $i \leq j \leq k$ to continuous intervals `[0,sub2_(_Niv_-1,_Niv_-1)]` and `[0,sub3_(_Niv_-1,_Niv_-1,_Niv_-1)]` respectively.

4.2 Overloaded operators

Overloaded operators for `+`, `-`, `*`, and `/` construct texts of expressions and their derivatives according to well-known rules of differentiation. They use four auxiliary routines `Add_`, `Sub_`, `Mult_`, and `Div_` to construct texts for addition, subtraction, multiplication, and division respectively. Here is a simplified version of `Mult_`:

```
char *Mult_(char *p, char *q, char freeptr) {
  char *r;
  int len;
  r=NULL;
  if (p && q) {
    len=strlen(p)+strlen(q)+2;
    r=MemNew(len);
    sprintf(r,"%s*%s",p,q);
  }
  switch (freeptr) {
    case 'l': MemFree(p); break;
    case 'r': MemFree(q); break;
    case 'b': MemFree(p); MemFree(q);
  }
  return r;
}
```

In fact, these routines perform more processing: they may surround operands by parentheses if needed and also do some simple optimization such as avoiding addition of zero and multiplication by one.

Here are examples of processes performed by overloaded addition and multiplication operators. It should be noted that formulas for second and third order derivatives for multiplication and division are rather complex and their direct implementation as a sequence of calls to `Add_`, `Sub_`, `Mult_`, and `Div_` routines would be very lengthy. Instead a simple language and an interpreter for it were developed. A program in the language is a sequence of simple assigned statements. Left-hand side of an assignment is always one of the symbols `$1`, `$2`, or `$3` which denote interpreter's temporary string variables. Right-hand side specifies a binary operation and its operands. An operand may be a temporary variable (`$1`, `$2`, `$3`), a small integer constant (e.g. `$c2`), an parameter (`$f0`, `$g0`), or its derivative (e.g., `$f3ijk`). For example, three first assignments in the `operator*` function below mean the following:

`$1=$f3ijk*$g0` - take the text of third order derivative of the first parameter with respect to the independent variables with numbers `i`, `j`, and `k` (integer variables `i`, `j`, and `k` are global and

therefore their values are known to the interpreter) and the text of second parameter, construct a text of their product and put a pointer to the text in first temporary variable;

$2=$f0*$g3ijk - similarly, take the text of first parameter and the text of the third order derivative of the second parameter with respect to the independent variables with numbers i, j, and k, construct a text of their product and put a pointer to the text in the second temporary variable;

$1=$1+$2 - construct a sum of the two previous expressions and put a pointer on it to the first temporary variable. The interpreter returns a pointer to the last constructed string.

```
Gradient operator+(const Gradient &f, const Gradient &g) {
  Gradient t;
  // f+g
  t.Expr=Add_(f.Expr,g.Expr,0);
  // (f+g)'''=f'''+g'''
  for (i=0; i<f.n3; i++)
    t.Der3[i]=Add_(f.Der3[i],g.Der3[i],0);
  // (f+g)''=f''+g''
  for (i=0; i<f.n2; i++)
    t.Der2[i]=Add_(f.Der2[i],g.Der2[i],0);
  // (f+g)'=f'+g'
  for (i=0; i<_Niv_; i++)
    t.Der1[i]=Add_(f.Der1[i],g.Der1[i],0);
  return t;
}
Gradient operator*(const Gradient &f, const Gradient &g) {
  Gradient t;
  char *desc;
  // f*g
  t.Expr=Mult_(f.Expr,g.Expr,0);
  // (f*g)'''
  for (i=0; i<_Niv_; i++)
    for (j=i; j<_Niv_; j++)
      for (k=j; k<_Niv_; k++) {
        desc="$1=$f3ijk*$g0" "$2=$f0*$g3ijk" "$1=$1+$2"
             "$2=$f2ij*$g1k" "$1=$1+$2"
             "$2=$f2ik*$g1j" "$1=$1+$2"
             "$2=$f2jk*$g1i" "$1=$1+$2"
             "$2=$f1i*$g2jk" "$1=$1+$2"
             "$2=$f1j*$g2ik" "$1=$1+$2"
             "$2=$f1k*$g2ij" "$1=$1+$2";
        t.Der3[sub3_(i,j,k)]=Interpret(f,g,desc);
      }
  // (f*g)''
  for (i=0; i<_Niv_; i++)
    for (j=i; j<_Niv_; j++) {
      desc="$1=$f2ij*$g0" "$2=$f0*$g2ij" "$1=$1+$2"
           "$2=$f1i*$g1j" "$1=$1+$2"
           "$2=$f1j*$g1i" "$1=$1+$2";
      t.Der2[sub2_(i,j)]=Interpret(f,g,desc);
    }
  // (f*g)'
  for (i=0; i<_Niv_; i++) {
    desc="$1=$f1i*$g0"  "$2=$f0*$g1i"  "$1=$1+$2";
```

```
        t.Der1[i]=Interpret(f,g,desc);
      }
    return t;
}
```

Overloaded operators for multiplication and division perform more complex processes than shown. For second and third order derivatives they distinguish cases when differentiation takes place with respect to two or three identical variables (e.g. `i==j`) and use simplified programs which lead in these cases to expressions with a lesser number of operations.

4.3 Overloaded assignment operator

Processing described so far ensures that an expression on the right-hand side of an assignment statement in the modified C++ program during its execution will be represented by an object of type `Gradient` with the `Expr` field pointing to the text of the expression itself and `Der1`, `Der2`, `Der3` fields pointing to texts of all its first, second and third order derivatives. Let us consider now what the overloaded assignment operator does. It simply replaces texts of derivatives of the left-hand side variable by copies of texts of derivatives of the right-hand side expression. Apart from this, a text of an assignment statement is printed for each derivative. These assignments constitute a body of a C function that will compute these derivatives.

```
    Gradient & Gradient::operator=(const Gradient &f) {
      char *p;
      switch (_Hod_) {
        case 3:    // third order derivatives
          for (i=0; i<_Niv_; i++)
            for (j=i; j<_Niv_; j++)
              for (k=j; k<_Niv_; k++) {
                p=f.Der3[sub3_(i,j,k)];
                if (p)
                  fprintf(MStream_," %s[%i]=%s;\n",Expr,sub3_(i,j,k),p);
                MemFree(Der3[sub3_(i,j,k)]);
                Der3[sub3_(i,j,k)]=StringSave(p);
              }
          break;
        case 2:    // second order derivatives
          for (i=0; i<_Niv_; i++)
            for (j=i; j<_Niv_; j++) {
              p=f.Der2[sub2_(i,j)];
              if (p)
                fprintf(MStream_," %s[%i]=%s;\n",Expr,sub2_(i,j),p);
              MemFree(Der2[sub2_(i,j)]);
              Der2[sub2_(i,j)]=StringSave(p);
            }
          break;
        case 1:    // first order derivatives
          for (i=0; i<_Niv_; i++) {
            p=f.Der1[i];
            if (p)
              fprintf(MStream_," %s[%i]=%s;\n",Expr,i,p);
            MemFree(Der1[i]);
            Der1[i]=StringSave(p);
          }
      }
```

```
    return *this;
  }
```

Note that here another global integer variable is used. The value of _Hod_ which stands for 'highest order of derivatives' specifies the order of derivatives to be assigned to. The variable is needed to provide the user with the ability to compute analytical derivatives of any order independently of the way that has been used for the computation of the derivatives of other orders. This will be discussed later.

4.4 Constructing C++ program for derivatives

Let us consider now how the described class and functions associated with it are used to tranform a program given by the user to a C++ program that produces functions to compute derivatives of RHS. Suppose that the declaration of the class is in file `autodif.h`. Remember also that the user specifies his dynamical system by giving names to phase variables and parameters and supplying expressions for the RHS. We will use the same example of RHS.

The process of building C++ program consists of two steps.

A main program is created. It is the same for any dynamical system.

```
#include <stdio.h>
#include "autodif.h"

FILE *MStream_;
int _Niv_,_Hod_;

void main(void) {
  MStream_=fopen("tmp_der.c","wt");
  _Der1();
  _Der2();
  _Der3();
  fclose(MStream_);
}
```

Then for each order of derivatives one function is created. Being called this function prints a text of another C function which computes derivatives of this order. These three functions are very similar so we show only one here. The only differences between them are dimensions of arrays (_d1Type, _d2Type, and _d3Type) and names of variables. Digits in them refer to the order of derivatives. Expressions for RHS given by the user are also used in these functions:

```
void _Der2(void) {
  fprintf(MStream_,"typedef double _d2Type[2][10];\n");
  fprintf(MStream_,"static _d2Type _d2;\n");
  fprintf(MStream_,"double *  _Der2(double *_x, double *_p) {\n");
  _Niv_=4;
  _Hod_=2;
  IndepVar(x,1);
  fprintf(MStream_,"#define x (_x[1])\n");
  IndepVar(y,2);
  fprintf(MStream_,"#define y (_x[2])\n");
  IndepVar(k,3);
  fprintf(MStream_,"#define k (_p[0])\n");
  IndepVar(g,4);
  fprintf(MStream_,"#define g (_p[1])\n");
```

```
      GradVar(x_);
      fprintf(MStream_,"#define x_ (*(_d2+0))\n");
      GradVar(y_);
      fprintf(MStream_,"#define y_ (*(_d2+1))\n");
      // User specified RHS
      x_=x*(1-x/k)-x*y;
      y_=-g*y+x*y;

      fprintf(MStream_,"#undef x\n");
      fprintf(MStream_,"#undef y\n");
      fprintf(MStream_,"#undef k\n");
      fprintf(MStream_,"#undef g\n");
      fprintf(MStream_,"#undef x_\n");
      fprintf(MStream_,"#undef y_\n");

      fprintf(MStream_,"return _d2[0];\n");
      fprintf(MStream_,"}\n");
  }
```

Let us comment on this function. Variable _Niv_ contains the number of independent variables which is the sum of numbers of phase variables and parameters. _Hod_ specifies the required order of derivatives. `IndepVar` and `GradVar` are macros defined in `autodif.h`. They are used to create variables for phase variables, parameters, and RHS values:

```
  #define IndepVar(name,varno) Gradient name(#name,varno,INDEP)
  #define GradVar(name)        Gradient name(#name,0,GRAD)
```

A call to `fprintf` is generated for each phase variable, parameter and RHS. This is done in the same way as desribed for the _Rhs function above.

It is clear that execution of the assignments to `x_` and `y_` will result in printing assignment statements for second order derivatives for the RHS. Here is the resulting C program. `#define` and `#undef` directives were removed from functions to shorten the listing.

```
  typedef double _d1Type[2][4];
  static _d1Type _d1;
  double *_Der1(double *_x, double *_p) {
    x_[0]=1-x/k+x*(-1/k)-y;        /* x */
    x_[1]=-x;                      /* y */
    x_[2]=x*(-(-x/k)/k);           /* k */
    y_[0]=y;                       /* x */
    y_[1]=0-g+x;                   /* y */
    y_[3]=(-1)*y;                  /* g */
    return _d1[0];
  }
  typedef double _d2Type[2][10];
  static _d2Type _d2;
  double *_Der2(double *_x, double *_p) {
    x_[0]=2*(-1/k);                /* x x */
    x_[1]=-1;                      /* x y */
    x_[3]=x*(-(-1)/k/k)-(-x/k)/k;  /* x k */
    x_[5]=x*(-x*2/k/k/k);          /* k k */
    y_[1]=1;                       /* x y */
    y_[7]=-1;                      /* y g */
```

```
     return _d2[0];
  }
  typedef double _d3Type[2][20];
  static _d3Type _d3;
  double *_Der3(double *_x, double *_p) {
     x_[4]=2*(-(-1)/k/k);              /* x x k */
     x_[7]=x*(-2/k/k/k)-x*2/k/k/k; /* x k k */
     x_[9]=x*(-6*(-x/k)/k/k/k);     /* k k k */
     return _d3[0];
  }
```

Arrays _d1, _d2, and _d3 are used to store values of derivatives of RHS. The number of rows is equal
to the dimension of the dynamical system (which is two in our example). The number of columns
is equal to the number of different derivatives in respect to phase variables and parameters. There
are two phase variables and two parameters in our example so there are four first order derivatives,
ten second order derivatives, and twenty third order derivatives. In general, these numbers are _Niv_,
sub2_(_Niv_-1,_Niv_-1)+1, and sub3_(_Niv_-1,_Niv_-1,_Niv_-1)+1.

   Note also that only non-zero values are assigned to elements of the arrays. Here we exploit a
property of static variables in C of being set to zero during startup. For each derivative names of
variables with respect to which it was calculated are shown in comments.

5. COMPLICATIONS: FUNCTIONS, VARIABLES, ARRAYS, AND STATEMENTS

The process of transforming a specification of a dynamical system to a C++ program that produces
another C program for computing derivatives was described with some simplifying assumptions made
about the structure of the specification. Namely, it was assumed that a dynamical system is specified
by a sequence of assignments with each right-hand side being an expression constructed from names
of phase variables and parameters. This is enough in many cases. Nevertheless the ability to use
sometimes other features of the programming language is desirable. These are functions (both standard
and user-defined), local variables, arrays, and control statements. The possibility of using these
features brings many technical complications to the transformation process. Let us briefly schetch the
modifications required to support them.

5.1 Standard functions

Each standard mathematical function such as `exp` or `sqrt` is also overloaded in such a way that it
returns a variable of type `Gradient` which describes a text of a function call along with texts of
all its derivatives with respect to formal parameters. The derivatives of the function with respect
to independent variables are constructed from these derivatives and derivatives of actual parameters
using the chain rule (e.g., $\frac{\partial \sin f}{\partial x} = \frac{\partial \sin f}{\partial f} \cdot \frac{\partial f}{\partial x}$).

   The main problem is that in a program that computes derivatives a function produces not only
one number but values of all its derivatives. This means that a function should store the values in
memory reserved for this purpose. It may be several calls to functions within one RHS so we have
used an array instead of a simple variable. Another important issue is efficiency. Straightforward
implementation of the calculation of derivatives of standard functions would be inefficient. Indeed if
the number of independent variables is 4 as in our example then a single call to `sqrt` will produce
4+10+20=34 derivatives each of which requires a call to `sqrt`. We cannot rely on a possibility that an
optimizing compiler could eliminate these common subexpressions. Usually compilers do not do this
assuming that functions may have side-effects. On the other hand derivatives of standard functions
may be expressed as a combination of this and another standard functions. This allows one to reduce
number of calls to standard functions to one or two. The implementation is as follows.

   We allocate two arrays `sv_` and `sd_`. Their dimension equals to maximum numbers of calls to
standard functions in one RHS. An appropriate statement that prints decalrations of the arrays is
placed to the `main` function in C++ program. One element (starting from the first) from each array

is allocated to each call to a standard function made in the right-hand side of an assigment statement (we omit a description of this process). Overloaded operator for each standard function prints two assignment statements like this

```
sv_[0]=sqrt(x*y/k);
sd_[0]=1/(2*sv_[0]);
```

where right-hand side of the first assignment is the original call the the function made from RHS and right-hand side of the second assignment is the first derivative of the function with respect to its own formal parameter. Then texts of all derivaties of the function with respect to independent variables are produced and associated with temporary variable of type `Gradient` which is the result of overloaded standard function. The trick is that elements of this arrays, `sv_[0]` and `sd_[0]` in this example, are used in texts of derivatives instead of explicit expressions for the function call and its derivatives.

To illustrate the processing let us slightly modify RHS in our example by adding a call to `sqrt` function

```
x'=x*(1-x/k)-x*y+sqrt(x*y/k);
```

Then the body of _Der1 function will be

```
sv_[0]=sqrt(x*y/k);
sd_[0]=1/(2*sv_[0]);
x_[0]=1-x/k+x*(-1/k)-y+sd_[0]*y/k;      /* x */
x_[1]=-x+sd_[0]*x/k;                    /* y */
x_[2]=x*(-(-x/k)/k)+sd_[0]*(-x*y/k)/k; /* k */
y_[0]=y;                                /* x */
y_[1]=0-g+x;                            /* y */
y_[3]=(-1)*y;                           /* g */
```

5.2 User-defined functions

Processing of user-defined functions is more complex then that of standard functions. Let us illustrate it on our example with another minor modification. Suppose there is a function

```
double hyp(double u, double v) {
  return sqrt(u*u+v*v);
}
```

and it is called from RHS

```
x'=x*(1-x/k)-x*y+hyp(k*x,x*y/g);
```

The source text of the function is transformed to the following C++ function (there are three different versions of the function for derivatives of first, second, and third orders but we show only one of them; digits in names reflect the order of derivatives):

```
#define hyp D2_hyp
int f2_hyp=1;
static Gradient D2_hyp(Gradient _u, Gradient _v) {
  Gradient _t_;
  char _b_[20],*_p_;
```

```
int _sub_,_i_;
int _j_; char *_q_;
int s_Niv_=_Niv_;
// Create C source text corresponding to hyp function.
// It computes second order derivatives with respect
// to its formal parameters (u and v).
_Niv_=2;
if (f2_hyp) {
  Gradient _t_;
  f2_hyp=0;
  rtFuncEntry_();
  fprintf(MStream_,"static double d2_hyp[3];\n");
  fprintf(MStream_,"static double D2_hyp(double u, double v) {\n");
  IndepVar(u,1,1);
  IndepVar(v,2,1);
  _t_= sqrt(u*u+v*v);
  fprintf(MStream_,"  d2_hyp[0]=%s;\n",GetDer2_(_t_,0,0));
  fprintf(MStream_,"  d2_hyp[1]=%s;\n",GetDer2_(_t_,0,1));
  fprintf(MStream_,"  d2_hyp[2]=%s;\n",GetDer2_(_t_,1,1));
  fprintf(MStream_,"  return D1_hyp(");
  fprintf(MStream_,"u,");
  fprintf(MStream_,"v)");
  fprintf(MStream_,";\n}\n\n");
  rtFuncExit_();
}
_Niv_=s_Niv_;
// Create C source text of assignment statements
// which compute second order derivatives of
// hyp function with respect to phase variables
// and parameters using the chain rule.
_sub_=rtGetSubscript_();
sprintf(_b_,"v_[%i]",_sub_);
fprintf(MStream_,"  %s=D2_hyp(",_b_);
fprintf(MStream_,"%s,",GetExpr_(_u));
fprintf(MStream_,"%s",GetExpr_(_v));
fprintf(MStream_,");\n");
_t_.SetExpr(_b_);
for (_i_=0; _i_<_Niv_; _i_++) {
  _p_=0;
  _p_=Add_(_p_,Mult_("d1_hyp[0]",GetDer1T_(_u,_i_),0),'b');
  _p_=Add_(_p_,Mult_("d1_hyp[1]",GetDer1T_(_v,_i_),0),'b');
 if (_p_) {
    sprintf(_b_,"d1_[%i][%i]",_sub_,_i_);
    fprintf(MStream_,"  %s=%s;\n",_b_,_p_);
    MemFree_(_p_);
    _t_.SetDer1(_i_,_b_);
  } else _t_.SetDer1(_i_,0);
  for (_j_=_i_; _j_<_Niv_; _j_++) {
    _p_=0;
    _p_=Add_(_p_,Mult_("d1_hyp[0]",GetDer2T_(_u,_i_,_j_),0),'b');
    _q_=0;
    _q_=Add_(_q_,Mult_("d2_hyp[0]",GetDer1T_(_u,_j_),0),'b');
    _q_=Add_(_q_,Mult_("d2_hyp[1]",GetDer1T_(_v,_j_),0),'b');
    _p_=Add_(_p_,Mult_(_q_,GetDer1T_(_u,_i_),0),'b');
```

```
        _p_=Add_(_p_,Mult_("d1_hyp[1]",GetDer2T_(_v_,_i_,_j_),0),'b');
        _q_=0;
        _q_=Add_(_q_,Mult_("d2_hyp[1]",GetDer1T_(_u_,_j_),0),'b');
        _q_=Add_(_q_,Mult_("d2_hyp[2]",GetDer1T_(_v_,_j_),0),'b');
        _p_=Add_(_p_,Mult_(_q_,GetDer1T_(_v_,_i_),0),'b');
        if (_p_) {
          sprintf(_b_,"d2_[%i][%i]",_sub_,sub2_(_i_,_j_));
          fprintf(MStream_," %s=%s;\n",_b_,_p_);
          MemFree_(_p_);
          _t_.SetDer2(_i_,_j_,_b_);
        } else _t_.SetDer2(_i_,_j_,0);
      }
    }
    return _t_;
  }
```

Although this function looks rather complex it has very clear structure with two main parts. The first part consists of statements inside if (f2_hyp) statement. When executed they print a source C text of the function D2_hyp which computes second order derivatives of original hyp function with respect to its formal parameters which are u and v in our example. A static array (d2_hyp) is allocated for values of the second order derivatives of hyp. Note that _Niv_ is set to the number of formal parameters just prior to the if statement. Another important thing is that produced C function calls its own version for first order derivatives to compute them. Functions rtFuncEntry_ and rtFuncExit_ are used to switch output stream to temporary file and restore it. This is needed to avoid placing the text at the point from which D2_hyp was first called. GetDer2_ function is used to retrive the text of appropriate second order derivatives; it returns "0" if the derivative is zero. Note that this part is executed only once at first call to the function made from RHS or another function. Appropriate #define directive allows to use original function name hyp at all places from which the function is called. In particular, it means that the _Der2 function needs not be modified.

The purpose of the second part is to print statements which compute values of the function and all its derivatives of the second order with respect to independent variables using the chain rule. Global arrays v_, d1_, d2_, and d3_ used to store the values produced by each call to a user-defined function. rtGetSubscript_function reserves particular elements of the arrays for each call; this will be discussed later. Assignment statements to compute first order derivatives are printed during execution of outer for loop. They just implement the chain rule for first order derivatives: $\frac{\partial hyp}{\partial x} = \frac{\partial hyp}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial hyp}{\partial v}\frac{\partial v}{\partial x}$ etc. Note that assignment is not generated if a derivative is zero. Similary, statements to compute second order derivatives are constructed during execution of inner for loop.

As a result of execution the C++ program the following text will be produced for functions computing derivatives of the original hyp function:

```
  static double d1_hyp[2];
  static double D1_hyp(double u, double v) {
    sv_[0]=sqrt(u*u+v*v);
    sd_[0]=1/(2*sv_[0]);
    d1_hyp[0]=sd_[0]*2*u;
    d1_hyp[1]=sd_[0]*2*v;
    return sv_[0];
  }
  static double d2_hyp[3];
  static double D2_hyp(double u, double v) {
    sv_[0]=sqrt(u*u+v*v);
    sd_[0]=1/(2*sv_[0]);
    d2_hyp[0]=sd_[0]*(2*u*2*u*(-1/(2*(u*u+v*v)))+2); /* u u */
```

```
    d2_hyp[1]=sd_[0]*2*u*2*v*(-1/(2*(u*u+v*v)));      /* u v */
    d2_hyp[2]=sd_[0]*(2*v*2*v*(-1/(2*(u*u+v*v)))+2); /* v v */
    return D1_hyp(u,v);
  }
  static double d3_hyp[4];
  static double D3_hyp(double u, double v) {
    sv_[0]=sqrt(u*u+v*v);
    sd_[0]=1/(2*sv_[0]);
    d3_hyp[0]=sd_[0]*(2*u*2*u*2*u*3/((u*u+v*v)*(u*u+v*v)*4)+
             3*2*u*2*(-1/(2*(u*u+v*v))));    /* u u u */
    d3_hyp[1]=sd_[0]*(2*u*2*u*2*v*3/((u*u+v*v)*(u*u+v*v)*4)+
             2*v*2*(-1/(2*(u*u+v*v))));       /* u u v */
    d3_hyp[2]=sd_[0]*(2*u*2*v*2*v*3/((u*u+v*v)*(u*u+v*v)*4)+
             2*u*2*(-1/(2*(u*u+v*v))));       /* u v v */
    d3_hyp[3]=sd_[0]*(2*v*2*v*2*v*3/((u*u+v*v)*(u*u+v*v)*4)+
             3*2*v*2*(-1/(2*(u*u+v*v))));    /* v v v */
    return D2_hyp(u,v);
  }
```

_Der2 function that computes second order derivatives of RHS now looks as follow (we omit all #define and #undef directives):

```
  typedef double _d2Type[2][10];
  static _d2Type _d2;
  double *  _Der2(double *_x, double *_p) {
    v_[0]=D2_hyp(k*x,x*y/g);
    d1_[0][0]=d1_hyp[0]*k+d1_hyp[1]*y/g;
    d2_[0][0]=(d2_hyp[0]*k+d2_hyp[1]*y/g)*k+(d2_hyp[1]*k+
             d2_hyp[2]*y/g)*y/g;
    d2_[0][1]=d2_hyp[1]*x/g*k+d1_hyp[1]*1/g+d2_hyp[2]*x/g*y/g;
    d2_[0][3]=d1_hyp[0]+d2_hyp[0]*x*k+d2_hyp[1]*x*y/g;
    d2_[0][6]=d2_hyp[1]*(-x*y/g)/g*k+d1_hyp[1]*(-y)/g/g+
             d2_hyp[2]*(-x*y/g)/g*y/g;
    d1_[0][1]=d1_hyp[1]*x/g;
    d2_[0][2]=d2_hyp[2]*x/g*x/g;
    d2_[0][4]=d2_hyp[1]*x*x/g;
    d2_[0][7]=d1_hyp[1]*(-x)/g/g+d2_hyp[2]*(-x*y/g)/g*x/g;
    d1_[0][2]=d1_hyp[0]*x;
    d2_[0][5]=d2_hyp[0]*x*x;
    d2_[0][8]=d2_hyp[1]*(-x*y/g)/g*x;
    d1_[0][3]=d1_hyp[1]*(-x*y/g)/g;
    d2_[0][9]=d1_hyp[1]*x*y*2/g/g/g+d2_hyp[2]*(-x*y/g)/g*(-x*y/g)/g;
    x_[0]=2*(-1/k)+d2_[0][0];                  /* x x */
    x_[1]=-1+d2_[0][1];                        /* x y */
    x_[3]=x*(-(-1)/k/k)-(-x/k)/k+d2_[0][3]; /* x k */
    x_[6]=d2_[0][6];                           /* x g */
    x_[2]=d2_[0][2];                           /* y y */
    x_[4]=d2_[0][4];                           /* y k */
    x_[7]=d2_[0][7];                           /* y g */
    x_[5]=x*(-x*2/k/k/k)+d2_[0][5];            /* k k */
    x_[8]=d2_[0][8];                           /* k g */
    x_[9]=d2_[0][9];                           /* g g */
    y_[1]=1;                                    /* x y */
    y_[7]=-1;                                   /* y g */
```

```
    return _d2[0];
  }
```

Note that call to D2_hyp function is followed by a sequence of assignment statements that compute first and second order derivatives of the function with respect to independent variables. Derivatives of the function with respect to its formal parameters and derivatives of actual parameters are used in these calculations.

rtGetSubscript_ function assigns elements of v_, d1_, d2_, and d3_ arrays to each call of each user-defined function. Basically, it assigns the value of a special counter (starting from zero) to each call made from an assignment statement and then increase the counter. This is accompanied by overloaded assignment operator which resets the counter. This processing ensures that values of all functions and their derivatives computed in a particular assignment statement are 'alive' up to the its end. A complication arises from the fact that elements of the arrays assigned to calls made from functions must be reserved forever and should not be assigned anymore. This is because a function may be called from several points and therefore values computed by functions it calls must be stored at the same places independently from which point the function is called. In fact, rtGetSubscript_ maintains a stack of lists with each stack element corresponding to appropriate level of function call; elements of this stack are pushed and poped by rtFuncEntry_ and rtFuncExit_ mentioned above.

5.3 Temporary variables
Temporary variables may be used in RHS or functions to simplify expressions. One could introduce such a variable to our example to reduce the number of operations:

```
double t;
t=x*y;
x'=x*(1-x/k)-t;
y'=-g*y+t;
```

During the transformation of a source C program to its C++ variant each declaration of a temporary variable, assignments to it, and its usage are modified in such a way that derivatives of the variable with respect to independent variables (or formal parameters if it declared inside a function) are computed and used.

To support usage of temporary variables in specification of dynamical systems following modifications is done to the scheme described so far.

- The constructor for the Gradient class performs special processing of a temporary varible to produce a declaration of arrays associated with it which are used to store the values of its derivatives.

  ```
  if (kind==TEMP) {
    fprintf(MStream_," double");
    switch (_Hod_) {
      case 3:
        fprintf(MStream_," d3_%s[%i],",name,n3);
      case 2:
        fprintf(MStream_," d2_%s[%i],",name,n2);
      case 1:
        fprintf(MStream_," %s,d1_%s[%i];\n",name,name,_Niv_);
    }
  }
  ```

- Overloaded assignment statement behaves slightly different for temporary variables. Apart from printing appropriate assignments to elements of arrays associated with such a variable it

also records that values of its derivatives are in these elements. Here is how an assignment to temporaty variable is processed. Compare it with the one for dependent variables (in fact overloaded assignment statement contains a `switch` statement which forks processing in depend on the kind of a variable).

```
p=f.Expr;
fprintf(MStream_,"  %s=%s;\n",Expr,p ? p : "0");
switch (_Hod_) {
  case 3:
    ...
  case 2:
    for (i=0; i<_Niv_; i++)
      for (j=i; j<_Niv_; j++) {
        p=f.Der2[sub2(i,j)];
        MemFree(Der2[sub2(i,j)]);
        if (p) {
          fprintf(MStream_,"  d2_%s[%i]=%s;\n",Expr,sub2(i,j),p);
          sprintf(buf,"d2_%s[%i]",Expr,sub2(i,j));
          Der2[sub2(i,j)]=StringSave(buf);
        } else Der2[sub2(i,j)]=NULL;
      }
  case 1:
    ...
}
fprintf(MStream_,"\n");
```

This ensures that in subsequent computations derivatives of a temporary variable will be taken from arrays associated with it.

- Each declaration of a temporary variable made in RHS or a function is transformed to declaration of a variable of type `Gradient`:

```
double t;
```

in the source C program becomes

```
TempVar(t);
```

in the C++ program and during its execution produces the following declarations in `_Der1`, `_Der2`, and `_Der3` functions respectivly:

```
double t,d1_t[4];
double d2_t[10],t,d1_t[4];
double d3_t[20],d2_t[10],t,d1_t[4];
```

`TempVar` is another macro defined in the `autodif.h` file as

```
#define TempVar(name)  Gradient name(#name,0,TEMP)
```

A result of the processing is illustrated by the `_Der2` function that computes second order derivatives of RHS (again, we omit all `#define` and `#undef` directives):

```
    typedef double _d2Type[2][10];
    static _d2Type _d2;
    double *  _Der2(double *_x, double *_p) {
      double d2_t[10],t,d1_t[4];
      t=x*y;
      d2_t[1]=1;
      d1_t[0]=y;
      d1_t[1]=x;
      x_[0]=2*(-1/k);            /* x x */
      x_[1]=-d2_t[1];            /* x y */
      x_[3]=x*(-(-1)/k/k)-(-x/k)/k; /* x k */
      x_[5]=x*(-x*2/k/k/k);      /* k k */
      y_[1]=d2_t[1];            /* x y */
      y_[7]=-1;                 /* y g */
      return _d2[0];
    }
```

5.4 Arrays

CONTENT allows any variable in a specification of a dynamical system be declared as an array, that is any phase variable, a parameter of the system, or temporary varibale may be one dimensional array. This brings further complications to the `Gradient` class and functions associated with it.

- One field, `Vardim`, to store the dimension of a variable is added to the `Gradient` class and one extra parameter is added to the constructor; its default value is one. One extra parameter is also added to each of `IndepVar`, `GradVar`, and `TempVar` macros.

- Fields `Expr`, `Der1`, `Der2`, and `Der3` all have one more level of indirection now and become pointers to arrays of pointers:

```
    char **Expr;
    char ***Der1;
    char ***Der2;
    char ***Der3;
```

- The constructor allocates these vectors according to the dimension of a variable even if it is one:

```
    Gradient::Gradient(char *name, int varno, int vardim, VarKind kind) {
      char buf[100];
      Vardim=vardim;
      Kind=kind;
      Expr=(char **)MemNew(vardim*sizeof(char *));
      *buf='\0';
      if (vardim>1)
        for (i=0; i<vardim; i++) {
          if (name) sprintf(buf,"%s[%i]",name,(int)i);
          Expr[i]=StringSave(buf);
        }
      else Expr[0]=StringSave(name);
      Der3=(char ***)MemNew(vardim*sizeof(char *));
      n3=sub3(_Niv_-1,_Niv_-1,_Niv_-1)+1;
      for (i=0; i<vardim; i++)
        Der3[i]=(char **)MemNew(n3*sizeof(char *));
      ...
    }
```

- One more field, `Subscript`, is added to `Gradient` class. It is set to zero by the constructor and changed by overloaded `[]` operator to store the value of a subscript expression:

```
Gradient operator[](int subscript) {
  Gradient t=*this;
  t.Subscript=subscript;
  return t;
}
```

Access to components of `Expr`, `Der1`, `Der2`, and `Der3` arrays are always made using the value of `Subscript` field as the subscript. For example, overloaded `+` operator looks like this:

```
Gradient operator+(const Gradient &f, const Gradient &g) {
  Gradient t;
  // f+g
  t.Expr[0]=Add_(f.Expr[f.Index],g.Expr[g.Index],0);
  // (f+g)'''=f'''+g'''
  for (i=0; i<f.n3; i++)
    t.Der3[0][i]=Add_(f.Der3[f.Index][i],g.Der3[g.Index][i],0);
  // (f+g)''=f''+g''
  for (i=0; i<f.n2; i++)
    t.Der2[0][i]=Add_(f.Der2[f.Index][i],g.Der2[g.Index][i],0);
  // (f+g)'=f'+g'
  for (i=0; i<_Niv_; i++)
    t.Der1[0][i]=Add_(f.Der1[f.Index][i],g.Der1[g.Index][i],0);
  return t;
}
```

- To allow overloaded assignment operator to access its left-hand side variable when assignment to an element of an array occures the `Parent` field of type `Gradient*` is used (the problem is that when overloaded assignment operator is called to process an assignment like `v[exp]=...` it has access only to a varibale implicitly created by overloaded `[]` operator while it has change fields of the left-hand side variable). This field is set to the value of `this` pointer by the constructor. Its value is used by overloaded assignment operator when it changes fields of its left-hand side variable, for example reference to components of `Der1` field looks like this:

```
Parent->Der1[Index][i]
```

- For temporary arrays the constructor produce declarations of two-dimensional arrays. For example, array declared as

```
double t[7];
```

becomes

```
TempVar(t,7);
```

in C++ program and produces the following declarations in `_Der1`, `_Der2`, and `_Der3` functions respectively:

```
double t[7],d1_t[7][4];
double d2_t[7][10],t[7],d1_t[7][4];
double d3_t[7][20],d2_t[7][10],t,d1_t[7][4];
```

Note also that for arrays the constructor sets `Expr[0]` fields to `"t[0]"`, `Expr[1]` to `"t[1]"`, etc.

5.5 Statements

There is no much sense in declaring arrays without possibility to process them in loops. It is also desirable to allow conditional statements in specifications of dynamical systems. Unfortunately, C++ does not allow for overloading of statements. That means in particular that it is not possible to 'tranfer' statements like

```
if (x>0) then x'=sqrt(x); else x'=x*x;
```

via a C++ program to functions for derivatives: there is no way to determine that the two assignments belong to different clauses of the `if` statement. Conditional operator `?:` also cannot be overloaded.

Despite this it is possible to use flow-control statements in a limited way. Standard C++ function $iff(cond, exp_1, exp_2)$ is intended to be used in place of conditional operator `?:`. Its derivatives are defined as $\frac{\partial\ iff}{\partial x} = cond\ ?\ \frac{\partial\ exp_1}{\partial x}\ :\ \frac{\partial\ exp_2}{\partial x}$, etc. It is supported by a set of overloaded relational, equality, and logical operators. A C version of `iff` is a simple macro:

```
#define iff(c,e1,e2) ((c) ? (e1) : (e2))
```

Loops that process components of arrays are also allowed. If some phase variables or parameters are declared as an array, say, `x[5]` then a loop like this may be used to compute the values of RHS:

```
int i;
for (i=1; i<4; i++) {
  x'[i]=x[i-1]*x[i+1];
}
```

It should be noted that execution such a loop in C++ program unrolls it in resulting C program:

```
x_[1][0]=1/x[2];
x_[1][2]=(-x[0]/x[2])/x[2];
x_[2][1]=1/x[3];
x_[2][3]=(-x[1]/x[3])/x[3];
x_[3][2]=1/x[4];
x_[3][4]=(-x[2]/x[4])/x[4];
```

In fact all iteration statements are allowed provided that no phase variables, parameters, or temporary variables used in their conditional expressions.

6. RUN-TIME COMPILATION

After the user has specified his dynamical system and source text of derivatives of functions was constructed CONTENT has to convert C text to a form suitable for computations, i.e. C text must be compiled and linked. Let us consider possible solutions of the problem.

6.1 Batch files

The worst, although the wide-spread solution, would be to ask the user to quit and run a batch file to compile the functions and link them with the rest of the software. This approach was implemented in AUTO86 (Doedel & Kernévez 1986), GRIND (Boer 1992) and DSTOOL (Back, Guckenheimer, Myers, Wicklin & Worfolk 1992), A better, but still insufficient one, would be to run batch file automatically from within the software and then 'restart' it without user's help, like in the version 1.0 of LOCBIF (Khibnik 1990) and in AUTO94 (Doedel, Wang & Fairgrive 1994). These kinds of dealing with the problem have several disadvantages. The first is that for each dynamical system a copy of executable file containing the whole software is build. This wastes disk space in a case the user creates more then one system. A copy of all object files must also be kept on a disk. Another problem is efficiency. As software for studing dynamical systems tend to be large enough link time may be quite long. There is no easy way to switch from one system to another and changing even one character in RHS requires rebuilding of all system.

6.2 Interpreters

Another approach to the problem is to convert functions to some kind of internal form such as Polish notation, trees, etc. and interpret it when the functions are to be computed. DSTOOL and PHASER (Koçak 1986) implemented it. This solution is machine-independent and may be implemented with a resonable amount of efforts. The language for specification of dynamical systems needs not be one of standard such as Fortran or C; it may provide more natural way of description of dynamical systems. The only drawback is slowdown of computations. Nevertheless this approach may be used in educational software.

6.3 Built-in compilers

An alternative to interpreting would be compiling functions by a built-in compiler as in LOCBIF (Khibnik, Kuznetsov, Levitin & Nikolaev 1993) and TRAX (Levitin 1989). This is the best solution from the user's point of view: it does not require any effort from him and provides high efficiency of computations. It is also the fastest one (no linking and no disk input-output) and it does not require any additional software. The main disadvantage is the cost of its implementation. A compiler is much harder to implement than an interpreter. It is also machine-dependent so porting a software to a computer of another architecture would mean reimplementing a part of the compiler.

6.4 Dynamic link libraries

Widely-used operating systems (OS) such as Unix and Microsoft Windows provide a facility which allows to solve our problem in an elegant and efficient way. It is called Dynamic Link Libraries in Windows and Shared Objects in Unix. We will use the term *dynamic libraries*. Although there are some differences between these facilities functionally they are equivalent. The idea behind them is very simple. A source program containing a set of routines is compiled and linked in a special mode producing an executable file in the form of a dynamic library which cannot be executed directly. Instead, an operating system provides three function which allow:

- to load a dynamic library to the memory,

- to retrieve the address of a routine by its name from previously loaded dynamic library, and

- to remove a dynamic library form the memory.

A program that uses a dynamic library does not refer to a routine in it by routine's name. Instead, it calls retrive-the-address function of OS and passes the name to it as a character string. In such a case there is no usual reference from the program to the routine and thus linker does not look for it at link time.

The advantages of using this facility are obvious:

- each specification of a dynamical system with functions for derivatives may be compiled to a separate dynamical library;

- the user needs not quit the software to create new or change existing dynamic system;

- interface to functions in dynamic libraries may be the same for all libraries because at each particular time only one of them is loaded;

- such dynamical libraries are produced faster because there is virtually no link step;

- such dynamical libraries are small and need not much disk space;

- dynamical libraries may be created, loaded, unloaded, and deleted independently from each other providing flexability in manipulation of dynamical systems;

- all users in a multiuser environment share the only copy of executable file of the software and it has not to be changed or linked during its life time;

- it allows to provide the user with ability to specify functions in some other circumstances such as functions along axes in graphics windows and functional parameters of algorithms.

CONTENT employs dynamic link facility in the way described above. Let us consider now the scheme used to produce dynamic libraries. As it was mentioned, they are constructed during four-step process.

- After the soure texts of the C program for dynamical system and the C++ program for derivatives of RHS are constructed and placed into two separate files, the C program is compiled. If there is an error in the specification of dynamical system the process halts and the user is notified about this.

- The C++ program is compiled and linked with the object file which contains functions associated with `Gradient` class. This step follows compilation of dynamical system because in a case of errors in RHS the C++ program also has errors.

- The C++ program created at the previous step is run. It creates a C source file with functions which compute derivatives of RHS.

- The source C file for RHS and the newly created C source file for derivatives are compiled together and then linked as dynamic library.

An error at any step halts the process and with notification the user that automatically constucted derivatives cannot be used in computations.

6.5 Compiling

An OS ability to execute commans issued by running programs is used to compile and link functions (unfortunately compilers and linkers do not exist in the form of dynamic libraries on either system). A call is made to the `system` function in Unix while `WinExec` is called in Windows. Here are Unix and Windows versions of batch files which do essensialy the same. Each file has three parameters two of which are names of source and target files and the third one indicates whether automatically constructed derivatives have be compiled (it is set to `"n"` when the user does not ask to do so or when functions other then RHS and derivatives are compiled).

Unix version for SGI (SUN version require `-G` option in `ld` command instead of `-shared`):

```
cc -c -o delete.o  -I./ $1.c >ErrLog 2>&1
ld -shared -o $2.so delete.o -ldl -lc -lm >>ErrLog 2>&1
if [ $3 = 'y' ]; then
 if [ -r $2.so ]; then
  rm tmp_der
  rm tmp_fun.c
  rm tmp_fun1.h
  CC -o tmp_der  -I./ tmp_der.cc autodif.o -ldl -lc -lm >ErrLog1 2>&1
  if [ -r tmp_der ]; then
   tmp_der >>ErrLog 2>&1
  fi
  cc -c -o delete.o  -I./ $1.c >ErrLog 2>&1
  ld -shared -o $2.so delete.o -ldl -lc -lm >>ErrLog 2>&1
 fi
fi
rm -f delete.o
```

Windows version for Borland C:

```
bcc -WDE @Respond.con -e%2.dll %1.c >delete.me
if not .%3 == .y goto end
if not exist %2.dll goto end
del tmp_der.exe
del tmp_fun.c
del tmp_fun1.h
```

```
bcc -tDe -P @Respond.con -etmp_der.exe tmp_der.cpp autodif.obj >ErrLog1
if not exist tmp_der.exe goto compile
tmp_der.exe
:compile
bcc -WDE @Respond.con -e%2.dll %1.c >delete.me
:end
del %1.obj
rename delete.me ErrLog
```

Some comments should be made on the latter version. While Unix's `system` function returns after a command represented by its argument has completed, its counterpart in Windows, `WinExec`, *does not* wait the completion of the command and returns *before* this moment. There is no legal way in Windows to determine when the command has completed so the following trick was implemented in CONTENT. After call to `WinExec` CONTENT installs a system timer via `SetTimer` function with a one second time-out value. From this moment Windows calls a callback function associated with the timer each second. The callback function checks whether `ErrLog` file exists (via `access` function). If it does not the callback does nothing and returns to Windows. Otherwise it kills the timer via `KillTimer` and proceeds with created dynamic library. The `rename` command in the batch file insures that `ErrLog` file appears only *after* the library has been created.

6.6 Processing of errors in RHS
If there are errors in a specification of RHS given by the user ErrLog file will contain messages about them issued by a compiler. In such a case CONTENT shows a special window with these messages and source text and allow the user to correct it (see Figure 6.1). After this it performs recompilation as is
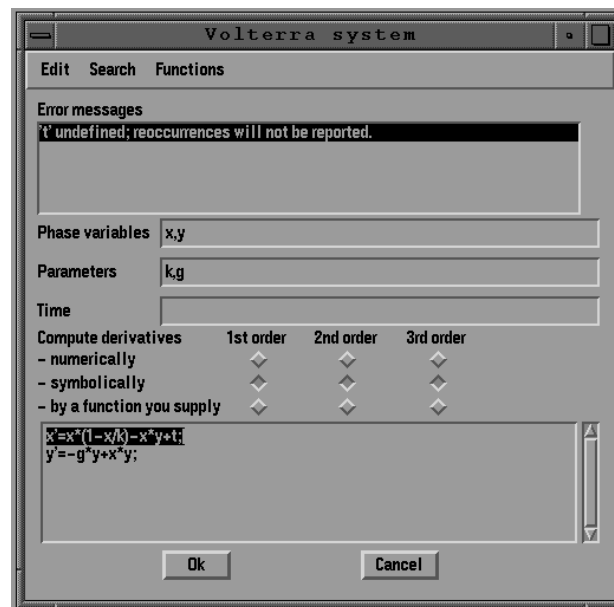

Figure 6.1: A window with a list of error messages.

described above.

7. DISCUSSION AND CONCLUDING REMARKS
Two problems were addressed in this work:

1. how to construct derivatives of RHS of dynamical systems, and

2. how to convert source code of functions and their derivatives into machine code and how access it during run-time.

The solution to the first problem is to provide the user with the possibility to select a way of construction derivatives. For derivatives of first, second, and third orders CONTENT allows to choose one of the following methods of computing derivatives:

- by finite diferences,

- by call a user-supplied function,

- by call a function that is automatically constructed by CONTENT.

Each method have its advantages, disadvantages, and overhead and none of them has absolute superiority over the others. It is up to the user to decide which one suits his particular needs. The software provides him with the choice and implements at least one method that does not require any efforts from the user and works well for 'typical' systems. Fourth and fifth order derivatives are not currently implemented although some algorithms need to compute them. The reasons were complexity of expressions for derivatives (with higher possibility to break compiler's complexity limits for expressions) and huge memory requirments for arrays. This problem is still under investigation.

Run-time compilation of functions coupled with using of dynamic libraries seem a good solution to the second problem. It does not require any efforts from the user and provides flexibility in dealing with functions.

Traditionally, developers of scientific software for investigating of dynamical systems spent much of their efforts on implementing numerical algorithms; such things as visualizing, interface with the user, storage of data, etc. were of second priority. The experience gained from the development of TraX, locbif, and content clearly shows importance of all 'non-mathematical' parts of software. Increasing productivity of users, more comfortable and easy-to-use environment worth extra efforts of developers.

REFERENCES

Back, A., Guckenheimer, J., Myers, M., Wicklin, F. & Worfolk, P. (1992), *dstool: Dynamical Systems Toolkit with Interactive Graphic Interface User's Manual*, Center for applied mathematics, Cornell University, USA.

Boer, R. (1992), *GRIND:GReat INtegrator of Differential equations*, Theoretical Biology Group, Utrecht University, The Netherlands.

Briggs, K. (1993), Simple automatic differentiation package, a C++ class declaration, version 1.0, Department of Applied Mathematics, University of Adelaida 5005, Australia.

Corliss, G. (1992), Automatic differentiation bibliography compiled by G.F. Corliss, Technical Report 167, Argonne National Laboratory. Mathematical and Computer Science Division.

Doedel, E. & Kernévez, J. (1986), *AUTO: Software for continuation problems in ordinary differential equations*, California Institute of Technology, Pasadena, CA. Applied Mathematics Report.

Doedel, E., Wang, X. & Fairgrive, T. (1994), *AUTO94: Software for continuation problems in ordinary differential equations*, California Institute of Technology, Pasadena, CA. Applied Mathematics Report.

Griewank, A. (1989), On Automatic Differentiation, *in* M. Iri & K. Tanabe, eds, 'Mathematical Programming', Kluwer, pp. 83–107.

Griewank, A. & Corliss, G., eds (1991), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, Penn.

Khibnik, A. (1990), LINLBF: A program for continuation and bifurcation analysis of equilibria up to codimension three, *in* D. Roose, B. d. Dier & A. Spence, eds, 'Continuation and Bifurcations: Numerical Techniques and Applications', Kluwer, Dordrecht, pp. 283–296.

Khibnik, A., Kuznetsov, Y., Levitin, V. & Nikolaev, E. (1993), 'Continuation techniques and interactive software for bifurcation analysis of ODEs and iterated maps', *Physica D* **62**, 360–371.

Koçak, H. (1986), *Differential and Difference Equation through Computer Experiments*, Springer-Verlag, New York.

Kuznetsov, Y. & Levitin, V. (1995), *CONTENT: Continuation Environment for Analysis of Dynamical Systems. User's Manual*, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam. In preparation.

Levitin, V. (1989), *TraX: Simulation and Analysis of Dynamical Systems*, Exeter Software, New York.