Sensitivity Analysis of a Dynamical System Using C++

DONNA CALHOUN[1] AND ROY OVERSTREET[2]

[1]*Computer Sciences Corporation, Seattle, WA 98115*
[2]*National Oceanic and Atmospheric Administration, Seattle, WA 98115*

ABSTRACT

This article introduces basic principles of first order sensitivity analysis and presents an algorithm that can be used to compute the sensitivity of a dynamical system to a selected parameter. This analysis is performed by extending with *sensitivity equations* the set of differential equations describing the dynamical system. These additional equations require the evaluation of partial derivatives, and so a technique known as the *table algorithm*, which can be used to exactly and automatically compute these derivatives, is described. A C++ class which can be used to implement the table algorithm is presented along with a driver routine for evaluating the output of a model and its sensitivity to a single parameter. The use of this driver routine is illustrated with a specific application from environmental hazards modeling.

1 INTRODUCTION

When modeling complex systems with large numbers of parameters, it is important to know how the model is affected by uncertainties in the parameters. Standard techniques of first order sensitivity analysis such as those described by Frank [1] can be employed to study the effects of parameter changes on a system. This article presents a C++ routine for evaluating the sensitivity of a dynamical system to its parameters. One key feature of this routine is that the partial derivatives needed to estimate the first order sensitivity are computed exactly using the well-known "table algorithm" described by Kalaba and Tischler [2].

First, the system error of a model is defined and a method for computing this error for a dynamical system is presented. Then an algorithm for simultaneously computing the solution to the model and sensitivity equations arising from a dynamical system is developed. The table algorithm used to compute derivatives required by sensitivity equations is illustrated. Finally, a complete C++ implementation of the algorithm used to solve model and sensitivity equations is given, along with an implementation of the table algorithm.

The methods developed in this article arose from the need to understand the uncertainty associated with a model developed by the Modeling and Simulations Studies Branch (MASS) of the National Oceanic and Atmospheric Administration (NOAA). This model estimates ground-level concentrations of toxic vapors resulting from accidental discharges of hazardous chemicals. As a final example in our article, we calculate the uncertainty in the "footprint" of a plume generated by evaporation of phosgene from a puddle, based on estimated uncertainty in environmental and chemical parameters.

Received April 1993
Revised June 1993

Scientific Programming, Vol. 2, pp. 157–169 (1993)
CCC 1058-9244/94/040157-13

2 FIRST ORDER SENSITIVITY ANALYSIS

The mathematical model used to represent a dynamical system is a differential equation of the form

$$y'(t;\alpha) = f(y(t;\alpha), t;\alpha) \quad y(t^0;\alpha) = y^0 \quad (1)$$

where

$$y'(t, y(t;\alpha);\alpha) = \begin{pmatrix} y_1'(t, y(t;\alpha);\alpha) \\ y_2'(t, y(t;\alpha);\alpha) \\ \vdots \\ y_n'(t, y(t;\alpha);\alpha) \end{pmatrix}.$$

Although in general, α may be a vector of parameters in $\mathcal{R}^m$, for this article, it is assumed that α is a scalar in $\mathcal{R}$. The equations given by (1) are called the model equations.

The aim of sensitivity analysis is to quantify how small changes in the value of the parameter α about some nominal value $\alpha_0 \in \mathcal{R}$ affect the output of the model equations. Formally, these effects are expressed as the quantity

$$\Delta y(t;\alpha) = y(t;\alpha_0 + \Delta\alpha) - y(t;\alpha_0)$$

where $y(t;\alpha)$ is the solution to (1). In first order sensitivity analysis, Δy is commonly approximated by

$$\Delta y(t;\alpha_0) \approx \frac{\partial y(t;\alpha_0)}{\partial\alpha}\Delta\alpha$$

where $|\Delta\alpha| \ll |\alpha_0|$. This quantity is referred to as the parameter-induced system error. This formulation is essentially that described by Frank [1]. In what follows, the term *system error* will refer to the parameter-induced system error.

For a model described by the differential equations (1), the partial derivatives needed to compute the system error are not readily available. However, they can be easily computed by applying basic rules of calculus. Differentiating both sides of (1) with respect to α and interchanging the order of differentiation on the left side results in a new differential equation whose solution is $\partial y(t;\alpha)/\partial\alpha$, the term needed to compute the system error. This new differential equation, defined as the sensitivity equation, has the form

$$w'(t, w(t,\alpha);\alpha) = \sum_{k=1}^{n} \frac{\partial f(t, y(t;\alpha);\alpha)}{\partial y_k} w_k(t;\alpha) + \frac{\partial f(t, y(t;\alpha);\alpha)}{\partial\alpha} \quad w_i(t^0) = w_i^0 \quad (2)$$

where w_i^0 is 1 if the initial condition y^0 depends on α and 0 if y^0 does not depend on α. The solution to (2) is

$$w(t;\alpha) = \frac{\partial y(t;\alpha)}{\partial\alpha}$$

Note that the equation given by (2) describes a system of equations, because

$$\frac{\partial f(t, y(t;\alpha);\alpha)}{\partial y_k} = \begin{pmatrix} \frac{\partial f_1(t, y(t;\alpha);\alpha)}{\partial y_k} \\ \frac{\partial f_2(t, y(t;\alpha);\alpha)}{\partial y_k} \\ \vdots \\ \frac{\partial f_n(t, y(t;\alpha);\alpha)}{\partial y_k} \end{pmatrix}$$

$$\text{and } \frac{\partial f(t, y(t;\alpha);\alpha)}{\partial\alpha} = \begin{pmatrix} \frac{\partial f_1(t, y(t;\alpha);\alpha)}{\partial\alpha} \\ \frac{\partial f_2(t, y(t;\alpha);\alpha)}{\partial\alpha} \\ \vdots \\ \frac{\partial f_n(t, y(t;\alpha);\alpha)}{\partial\alpha} \end{pmatrix}$$

An important observation in (2) is that in order to evaluate the terms

$$\frac{\partial f(t, y(t;\alpha);\alpha)}{\partial y_k} \quad \text{and} \quad \frac{\partial f(t, y(t;\alpha);\alpha)}{\partial\alpha}$$

the values $y(t;\alpha) = (y_1(t;\alpha), y_2(t;\alpha), \ldots, y_n(t;\alpha))^T$ will be required. As a consequence, the model and the sensitivity equations must be solved simultaneously.

3 DESIGN OF ROUTINES FOR COMPUTING MODEL SENSITIVITY

A set of routines to compute model sensitivity should compute the numerical solution to the

model and sensitivity equations and related system error. Ideally, such a set of routines should only require definitions of the model equations, a time interval over which to compute the solution to these equations, a time step, initial conditions, parameters, and $\Delta\alpha$, an estimate of deviation of the parameter α from its expected value. In particular, the user of these routines should not have to be involved in the details of formulating the sensitivity equations.

3.1 Solving the Model and Auxiliary Equations

In general, the model and sensitivity equations must be solved numerically. As was noted above, in order to evaluate the right hand side of the equation (2) at a particular time step t^i, it is necessary to know the solution to (1) at time step t^i. Although it is possible to first compute the solution to (1) on a given time interval, store these results, and then solve (2) on that interval, a better approach is to solve the two equations simultaneously.

To solve (1) and (2) simultaneously, it is necessary to construct a new differential equation:

$$u'(t, u(t; \alpha); \alpha) = F(t, u(t; \alpha); \alpha) \qquad (3)$$

where

$$u(t; \alpha) = \begin{pmatrix} y(t; \alpha) \\ \hline w(t; \alpha) \end{pmatrix} = \begin{pmatrix} y_1(t; \alpha) \\ \vdots \\ y_n(t; \alpha) \\ \hline w_1(t; \alpha) \\ \vdots \\ w_n(t; \alpha) \end{pmatrix}$$

and

$$F(t, u(t; \alpha); \alpha) = \begin{pmatrix} f_1(t, y(t; \alpha); \alpha) \\ \vdots \\ f_n(t, y(t; \alpha); \alpha) \\ \hline \sum_{k=1}^{n} \frac{\partial f_1(t, y(t; \alpha); \alpha)}{\partial y_k} w_k(t; \alpha) + \frac{\partial f_1(t, y(t; \alpha); \alpha)}{\partial \alpha} \\ \vdots \\ \sum_{k=1}^{n} \frac{\partial f_n(t, y(t; \alpha); \alpha)}{\partial y_k} w_k(t; \alpha) + \frac{\partial f_n(t, y(t; \alpha); \alpha)}{\partial \alpha} \end{pmatrix}$$

The differential equation given by (3) can be solved using any suitable numerical differential equation solver. In the algorithm presented below, a generic routine called Integrate is used. The vectors $y^i, w^i \in \mathcal{R}^n$ and $u^i \in \mathcal{R}^{2n}$ represent the numerical solutions to (1), (2), and (3), respectively, at time $t^i = t^0 + i * h$. The vector $syserr^i \in \mathcal{R}^n$ represents the system error at time t^i.

The algorithm used to solve (1) and (2) simultaneously is:

```
for j = 1, . . . , n
  u_j^0 = y_j^0
  u_{j+n}^0 = w_j^0
  syserr_j^0 = u_{j+n}^0 Δα                    (4)
end
t^i = t^0
for i = 0, 1, . . . , N − 1
  u^{i+1} = Integrate(F, t^i, u^i, h)
  for j = 1, . . . , n
    y_j^{i+1} = u_j^{i+1}
    w_j^{i+1} = u_{j+n}^{i+1}
    syserr_j^{i+1} = u_{j+n}^{i+1} Δα
  end
  t^{i+1} = t^i + h
end
```

3.2 Automatic Differentiation

The issue central to the design of routines for sensitivity analysis is that of derivative evaluation. In particular, the derivatives of the right hand side of the model equations must be evaluated with respect to both the dependent variables and parameters. The routines developed here use the table algorithm for computation of these derivatives. This algorithm is discussed briefly and then illustrated on a sample function.

The well-known table algorithm, developed by Kalaba and described by others [2–8], is a chain rule-based technique used to obtain numerical values for the derivatives of a function simultaneously with the function evaluation without appealing to numerical or symbolic differentiation [3]. To compute the derivative of a function using the table algorithm, it is necessary to decompose the function into a finite sequence of algebraic operators and elementary functions. A step in the algorithm consists of applying a function or operator to results obtained in a previous step. In prin-

ciple, this is no different from normal procedure used to evaluate a function. Where the table algorithm differs from routine function evaluation is that at each step in the computation, not only is the value of the operator or function computed, but the derivative of that operator or function is also calculated.

Table 1 shows the steps needed to apply the table algorithm to the sample function

$$f(x) = \alpha \sin(x) + e^{x^2} \tag{5}$$

Computations in the first column of Table 1 correspond to the steps needed to evaluate (5). Computations in the second column of Table 1 correspond to the steps needed to evaluate derivatives of the operation or function applied in the first column.

The results of applying the table algorithm to (5) are stored in F and F_x, which correspond to f(x) and f'(x), respectively. The table algorithm can be applied to any function that can be decomposed into a finite sequence of elementary functions and operations. For a complete discussion of the conditions under which the table algorithm can be applied, see Rall [6].

The table algorithm can be implemented by constructing length 2 vectors of the form $x = (x_1, x_2)$, which can be used to store the results of column 1 and column 2 of Table 1. The first element of this vector stores the value of the operation or function and the second element stores the derivative of the operation or function. Algebraic operators and elementary functions can then be defined for this vector so that the computations required by both columns of Table 1 are carried out in a single application of the operator or function.

Table 2 shows how operations in Table 1 can be redefined for length 2 vectors $A = (A_1, A_2)$, $B = (B_1, B_2)$, . . . , $F = (F_1, F_2)$. Column 1 of Table 2 is identical to column 1 of Table 1, except that operations and functions have been redefined for the length 2 vectors. The manner in which these operations and functions are redefined is illustrated by the remaining two columns of Table 2. The variables x and α are real valued scalars, as in Table 1. Other operators and elementary functions are defined in a similar manner. For a complete set of rules used to define several more elementary operations and functions, see Jerrell [3] and [8].

Table 1. Table Algorithm Applied to $f(x) = \alpha \sin(x) + e^{x^2}$

Sequence of Steps	Derivative d/dx
$A = x$	$A_x = 1$
$B = \sin(A)$	$B_x = \cos(A)*A_x$
$C = \alpha*B$	$C_x = \alpha*B_x$
$D = A*A$	$D_x = 2*A*A_x$
$E = \exp(D)$	$E_x = \exp(D)*D_x$
$F = E + C$	$F_x = E_x + C_x$

Table 2. Table Algorithm in Vector Form

Operations and Functions Redefined for A = (A1, A2) etc.	Operations and Functions Defined as:	
$A = x$	$A_1 = x$	$A_2 = 1$
$B = \sin(A)$	$B_1 = \sin(A_1)$	$B_2 = \cos(A_1)*A_2$
$C = \alpha*B$	$C_1 = \alpha*B_1$	$C_2 = \alpha*B_2$
$D = A*A$	$D_1 = A_1*A_1$	$D_2 = 2*A_1*A_2$
$E = \exp(D)$	$E_1 = \exp(D_1)$	$E_2 = \exp(D_1)*D_2$
$F = E + C$	$F_1 = E_1 + C_1$	$F_2 = E_2 + C_2$

4 IMPLEMENTATION

The algorithm given by (4) has been implemented in C++. The derivatives required by the differential equation (3) are computed using the table algorithm, which has been implemented as a C++ class.

4.1 Array Types

In the routines implemented here, one-dimensional double arrays (or vectors) are represented by the class *DoubleVec**. Two-dimensional double arrays (or matrices) are represented by the class *DoubleGenMat*. It is beyond the scope of this article to present a full description of *DoubleVec* and *DoubleGenMat*, so it is suggested that the interested reader consult Keffer [9] and Vermeulen et al. [10] for more information on these classes.

4.2 Automatic Differentiation Using the Class *doubleTT*

To compute the derivatives required by the differential equation F given by (3), a class named *doubleTT* (for *double table type*) has been imple-

* The classes *DoubleVec* and *DoubleGenMat* are taken from Rogue Wave's Math.h++ Library. More information about the Rogue Wave math libraries can be obtained from Rogue Wave Software, P.O. Box 2328, Corvallis, OR 97339, (503) 754-3010.

mented. The class presented here can be used to compute the derivatives of a variety of functions encountered in dynamical systems. It should be noted, however, that this class was not designed as a general implementation of the table algorithm. It is not, for example, implemented for a single precision type.

```
// Special functions
friend doubleTT cos(doubleTT);
friend doubleTT exp(doubleTT);
// sin, tan, pow, And other needed special functions
// Algebraic operators
friend doubleTT  operator+(doubleTT, doubleTT);
friend doubleTT  operator+(double , doubleTT);
friend doubleTT  operator+(doubleTT, double );
// etc.
```

The class *doubleTT* has two private member variables:

```
class doubleTT
{
    private :
        double v;
        double d;
    public :
    :
    :
};
```

The class member variables v and d store the *value* and *derivative*, respectively, of a function returning a variable of type *doubleTT*.

There are two constructors for *doubleTT*. They are:

```
doubleTT();
doubleTT(double val, double dv = 0.0)
```

The first sets class member variables v and d to 0. The second sets class member variable v to val and the class member variable d to dv. This second constructor allows a *double* object to be promoted to a *doubleTT* object in assignment operators.

Two access member functions and two global functions are defined for *doubleTT*:

```
double&   value();
double&   deriv();
friend double value(doubleTT);
friend double deriv(doubleTT);
```

The first two are used to access the class member variables v and d, respectively, of a *doubleTT*. The second two are needed to compute the *value* and *derivative* of an expression.

Finally, all operators and elementary functions are overloaded so that they are defined for *doubleTT*.

Special functions are overloaded in a manner illustrated by this example using sin(x):

```
doubleTT     sin(doubleTT x)
{
    doubleTT t;

    t.v = sin(x.v);
    t.d = cos(x.v)*x.d;
    return t;
}
```

Similarly, arithmetic operators are defined as illustrated by this example:

```
doubleTT      operator*(doubleTT
  a, doubleTT b)
{
    doubleTT t;
    t.v = a.v*b.v;
    t.d = a.v*b.d + a.d*b.v;
    return t;
}
```

Note that for binary operators it is necessary to consider the three combinations (*double*, *doubleTT*), (*doubleTT*, *double*), and (*doubleTT*, *doubleTT*).

Relational operators are defined only for the *value of a doubleTT* object. For example, operator<=() is defined as:

```
// Relational operators.
friend short     operator <=(doubleTT
  a,  doubleTT b) { return (a.value()
  <= b.value()); }
```

The following example illustrates how to use the class *doubleTT* to compute the derivative of a scalar valued function. First, the function to be differentiated must be defined using the type *doubleTT*.

```
doubleTT     f(doubleTT t)
{
    return 3.0*sin(t) + exp(t*t);
}
```

Then a variable a of type *doubleTT* is declared:

```
doubleTT a(1.2,1);
```

Note that to compute the derivative of f with respect to a, it is necessary to set $deriv(a) = 1$. The function f is then called and the result stored as a *doubleTT*:

```
doubleTT result = f(a);
```

Finally, the values f(a) and f′(a) are obtained from the variable `result` using the *doubleTT* member functions *value()* and *deriv()*:

```
double fa = result.value();
  // fa = f(a)
double dfa = result.deriv();
  // dfa = f'(a)
```

4.3 The Class *DoubleTTVEc*

To store an array of objects of type *doubleTT*, a class *DoubleTTVec* has been implemented.† This class, which is an array class for the type *doubleTT*, has all of the functionality of the analogous class *DoubleVec*. Two extra functions are added to complete the class *DoubleTTVec*.

† The class *DoubleTTVec* was generated using Rogue Wave's template for creating vectors of an arbitrary arithmetic type.

The two functions are:

```
DoubleVec value(DoubleTTVec& V)
DoubleVec deriv(DoubleTTVec& V)
```

and are just vector versions of the functions *value()* and *deriv()* defined for the base type *doubleTT*.

4.4 Model Equations

To take advantage of the table algorithm, model equations must be defined in terms of the type *doubleTT* and *DoubleTTVec*. The equations given in (1) are specified by the user in the following general form:

```
DoubleTTVec UserF(doubleTT t,
  DoubleTTVec& y)                    (6)
{
    long n = y.length();
    DoubleTTVec f(n);
    f(0) = ... // f1(t,y)
    f(1) = ... // f2(t,y)
       :
       :
    f(n-1) = ... // fn(t,y)
    return f;
}
```

where the vector `f` returned from `UserF` is the value of the right hand side of the model equations evaluated at `y` and `t`. A pointer to a function of the form given above will be named a *diffEqTTVec*. A pointer to the analogous real-valued function of the form

```
DoubleVec f(double t, DoubleVec y);
```

will be named a *diffEqVec*.

Absent from function definitions represented by the types of *diffEqTTVec* and *diffEqVec* is an expression of the dependency of model equations on their parameters. The reason for this is that parameters are declared as global to model functions and therefore the dependency of model functions on parameters is not explicit.

4.5 Implementation of Algorithm (4)

It is now possible to present a C++ implementation of the algorithm given by (4). First, it is assumed that a subroutine is available that can solve a differential equation numerically. The routine used here is a fourth order Runge-Kutta method and is called `Rk4Step`. Any suitable differential equation solver can be used, however.

Next, it is assumed that the user has supplied a function of the form given by (6) that can be used to evaluate the right hand side of the model equations. This function is referenced by the global pointer `gUserF`. Finally, it is assumed that the user defined function depends on a global parameter that is referenced by the global pointer variable `gAlpha`.

The function F given by (3) can now be defined

```
diffEqTTVec *gfUser;      // Pointer to user defined function
doubleTT    *gAlpha;      // pointer to parameter upon which user defined
                             function depends.
                          // These globals are assigned in a driver routine.
DoubleVec   F(double t, DoubleVec& u)                                    (7)
{
    long n = u.length()/2;
    DoubleVec y(n), w(n);

    for(short i = 0; i < n; i++)
    {
        y(i) = u(i);
        w(i) = u(i+n);
    }
    // Evaluate model equations
    DoubleVec uModel = value((*gfUser)(t,y));    // (7a)

    // Now evaluate sensitivity equations.
    DoubleTTVec yTT = y;                          // (7b)
    DoubleVec fSum(n,0);
    for(i = 0; i < n; i++)                        // (7c)
    {
        yTT(i).deriv() = 1.0;
        fSum += deriv((*gfUser)(t,yTT))*w(i);
        yTT(i).deriv() = 0.0;
    }

    // Comput ∂f/∂a                               // (7d)
    gAlpha->deriv() = 1.0;
    DoubleVec dfda = deriv((*gfUser)(t,yTT));
    gAlpha->deriv() = 0;
    DoubleVec uSens = fSum + dfda;

    DoubleVec U(2*n);                             // (7e)
    for(i = 0; i < 2*n; i++)
    {
       U(i) = (i < n) ? uModel(i) : uSens(i-n);
    }
    return U;
}
```

Since F is to be passed to a differential equation solver, which will not in general be defined for *doubleTT* and *doubleTTVec*, F must take and return real-valued scalars and vectors. Hence, in (7a) the results from the user defined function must be passed to *value()* to obtain a real-valued vector. In (7b), a *DoubleTTVec* must be constructed from the *DoubleVec* y extracted from the input vector u. In (7c) and (7d), results from the user defined function are passed to *deriv()* again to obtain a real-valued vector. Finally, in (7e) a real-valued vector U is constructed, assigned values from `uModel` and `uSens`, and returned from the procedure.

To set up the global pointers `gAlpha` and `gfUser` and to call the differential equation solver `Rk4Step`, the following driver routine has been written:

```
void DESolve(diffEgTTVec& f,double t0,doubleTTPtrs& Ya,double tN,
                          double h, doubleTT* parms, double dp,
                          DoubleGenMat **y, DoubleGenMat **w,
                          DoubleGenMat** sysErr)
{
   double ti;
   long    i,j,N,fNum;

   N = (long) ((tN-t0)/h + 1.0);

   fNum = Ya.length();

   *y = new DoubleGenMat(N,fNum);
   *w = new DoubleGenMat(N,fNum);
   *sysErr = new DoubleGenMat(N,fNum);

   gfUser = &f;
   gAlpha = parms;

   DoubleVec ui(2*fNum);

   // Initialize matrices needed locally.
   for (j = 0; j < fNum; j++)
   {
       (**y)(0,j) = ui(j) = (*Ya(j)).value();
       (**w)(0,j) = ui(j+fNum) = (parms == Ya(j));
       (**sysErr)(0,j) = ui(j+fNum)*dp;
   }
   for(i = 0; i < (N-1); i++)
   {
       ti = t0 + i*h;

       DoubleVec uip1 = RK4Step(F,ti,ui,h);

       for(j = 0; j < fNum; j++)
       {
           (**y)(i+1,j) = uip1(j);
           (**w)(i+1,j) = uip1(j+fNum);
           (**sysErr)(i+1,j) = uip1(j+fNum)*dp;
       }
       ui = uip1;
    }
}
```

where *doubleTTPtrs* is an array of pointers to the type *doubleTT*.

The numerical solutions to (1) and (2) are stored in the matrices referenced by `y` and `w`, respectively. The system error is stored in the matrix referenced by `sysErr`.

5 AN APPLICATION

The methods of model sensitivity are applied to a practical problem in which liquid phosgene spills onto the ground from a ruptured tank, creating an evaporating puddle. The vapors from this puddle are carried downwind in the form of a toxic plume. A diffusion-advection equation is used to estimate the size of the plume as a function of meteorological conditions, properties of the chemical, and the rate at which the pool evaporates. The ground-level distribution of the concentration of a plume at time t is given by $P(x,y,t)$, the solution to the partial differential equation

$$\frac{\partial P}{\partial t} + U(t)\frac{\partial P}{\partial x} = D\frac{\partial^2 P}{\partial x^2} + D\frac{\partial^2 P}{\partial y^2} + f(x, y, t) \quad (8)$$

where $x, y \in \Omega$ and

$$f(x, y, t) = \delta(x - x_{source})\delta(y - y_{source})Q(t)$$

is the point source located at (x_{source}, y_{source}). U(t) is the wind speed, which in this case is assumed to be constant and D is the diffusion coefficient. The region Ω over which the equation holds is assumed to be large enough so that the concentration of the plume at the boundary of Ω is 0.

The source is assumed to be equal to the rate at which mass is lost from the puddle by evaporation. For simplicity, it is assumed that the puddle neither spreads nor shrinks during the time of interest. Hence,

$$Q(t) = E(t)A_0 \quad (9)$$

where E is the evaporation flux (kg m^{-2} s^{-1}) and A_0 is the nominal puddle area (m^2). The direct integration of (8) requires independent determination of E. The evaporation mass flux is given by

$$E(T_p, t) = K_m \frac{M_w P_v(T_p)}{RT_p} \quad (10)$$

where M_w = molecular weight, (kg/kmol); P_v = vapor pressure of chemical, (Pa); T_p = pool temperature, (K); R = universal gas constant, (J/kmol K); K_m = mass transfer coefficient, (m/s). The mass transfer coefficient K_m is determined from the solution given by

$$K_m = -u_* \frac{k}{Sc_T}(1 + n)\frac{P_a}{P_v(T_p)} \ln\left(1 - \frac{P_v(T_p)}{P_a}\right)\bar{G}(\xi) \quad (11)$$

where

$$\bar{G}(\xi) = \frac{1}{2} - \frac{\bar{g}_0}{\pi}\tan^{-1}\left(\frac{\ln \xi}{\pi}\right) + \frac{\bar{g}_1}{\ln^2 \xi + \pi^2} + \frac{\bar{g}_2 \ln \xi}{(\ln^2 \xi + \pi^2)} + \frac{\bar{g}_3(\ln^2 \xi - \frac{1}{3}\pi^2)}{(\ln^2 \xi + \pi^2)^3} \cdots$$

u_* is the friction velocity, k is von Kármán's constant, Sc_T is the turbulent Schmidt number, n is the atmospheric stability parameter, and ξ is a nondimensional form of the pool diameter (m). The constants $(\bar{g}_0, \bar{g}_1, \bar{g}_2, \bar{g}_3)$ are (1, 0.4228, 2.824, 1.025) [11].

The term $M_w P_v(T_p)/RT_p$ in (10) is the saturation vapor concentration at the pool's surface. The functional form of $P_v(T_p)$ is known and T_p is determined through the use of an energy conservation equation. It is assumed that the loss of mass of the puddle is negligible with respect to the size of the puddle and so a mass conservation equation is not used. Evaporation occurs when the net energy flux across the pool's top and bottom surfaces overcomes the heat of vaporization. The formulation of each flux is known. Hence, an energy budget can be constructed in the form

$$\frac{dT_p}{dt} = F[Fluxes(T_p; \text{ environmental parameters}; E(T_p)); \text{ chemical properties}] \quad (12)$$

from which T_p and $E(T_p)$ are found by iteration. Substitution of $E(T_p)$ into (9) allows integration of (8) for the vapor concentration.

5.1 Determining Sensitivity of Plume Concentration

The routines developed in Section 4 can be used to compute the sensitivity of the plume concentration to model parameters. The sensitivity will be tested with respect to Sc_T, the turbulent Schmidt number.

Using a Discrete Space-Continuous Time differing scheme (also known as the Method of Lines), equation (8) can be converted to a system of ordinary differential equations. To do this, the region Ω over which equation (8) is assumed to hold is discretized on a regular grid whose nodes are:

$$(x_i, y_j),\ i = 0, \ldots, n+1, j = 0, \ldots, m+1.$$

The resulting ordinary differential equations are:

$$\frac{dP_{ij}}{dt} = -U\left(\frac{P_{i+1,j} - P_{i-1,j}}{2h_x}\right) + D\left(\frac{P_{i+1,j} - 2P_{ij} + P_{i-1,j}}{h_x^2}\right) + D\left(\frac{P_{i,j+1} - 2P_{ij} + P_{i,j-1}}{h_y^2}\right) + Q_{ij}(x_i, y_j, t)$$

where P_{ij} is the plume concentration at node (x_i, y_j) and h_x and h_y are the mesh sizes in the x and y direction, respectively. $Q_{ij}(t)$ is defined as

$$Q_{ij}(x_i, y_j, t) = \begin{cases} Q(x_i, y_j, t), & i = i_{source}, j = j_{source} \\ 0 & \text{otherwise} \end{cases}$$

where the node $(x_{i_{source}}, y_{j_{source}})$ is the source of the plume.

The following program illustrates how to set up model equations and arguments for the driver routine DESolve.

```
#include "puddleScenario.h"
#include "diffEqTT.h"

long          gNodesN, gNodesM, gSourceX, gSourceY;
double  gXMin, gYMin, gHX, gHY, gD = 0.1;

doubleTT dTdt(doubleTT t, DoubleTTVec& tp)
{
    long n = tp.length();
    doubleTT temp = tp(n-1);
    return (1.0/(gLiquidDensity*gPuddleDepth*gSpecificHeat)*
           (FLUX_S() + FLUX_UP(temp) + FLUX_DWN() + FLUX_E(temp) +
            FLUX_H(temp) + FLUX_G(t,temp)));
}

DoubleTTVec PlumeVec(doubleTT t, DoubleTTVec& y)
{
    long n = y.length(), Np1 = gNodesN+1, Mp1 = gNodesM+1, sourceXidx,
      sourceYidx;
    long Np2 = gNodesN+2, Mp2 = gNodesM+2;
    doubleTT Pij, Pip1j, Pim1j, Pijp1, Pijm1, dP2dx2, dP2dy2, dPdx;
    sourceXidx = (long) (gSourceX - gXMin)/gHX;
    sourceYidx = (long) (gSourceY-gYMin)/gHY;
    DoubleTTVec P(n);
    for(long j = 1; j < Np1; j++)
    {
        for(long i = 1; i < Mp1; i++)
        {
            Pij = y(j*Np2 + i);
            Pip1j = y(j*Np2 + i+1);
            Pim1j = y(j*Np2 + i-1);
            Pijp1 = y((j+1)*Np2 + i);
            Pijm1 = y((j-1)*Np2 + i);
            dPdx = (Pip1j-Pim1j)/(2*gHX);
            dP2dx2 = (Pip1j - 2*Pij + Pim1j)/(gHX*gHX);
            dP2dy2 = (Pijp1 - 2*Pij + Pijm1)/(gHY*gHY);
```

```
            P(j*Np2 + i) = -gWindSpeed*dPdx + gD* (dP2dx2 + dP2dy2);

            if (i == sourceXidx && j == sourceYidx)
            {
                P(j*Np2+i) += FLUX_E(y(n-1));
            }
        }
    }
    P(n-1) = dTdt(t,y);
    return P;
}

void main()
{
    // Read in chemical properties and global parameters.
    ReadChemical(phosgene);
    PuddleDefaults();
    gPoolDiameter = 10;
    gPuddleDepth = 0.005;
    gZ0 = 0.03;
    gWindSpeed = 6;
    gAirTemperatureK = 273.15;
    gPuddleTemperature = 273.15;
    gLatitude = 45.0;        // St. Paul Minnisota
    gLongitude = 93.1;

    // Set up initial conditions.
    gNodesN = gNodesM = 29;  // Number of interior nodes
    double width = 1000, length = 1000;
    gHX = gHY = length/double(gNodesN+1);
    gSourceX = gSourceY = 0;
    gXMin = -length*0.25; gYMin = -width/2.0;
    long numberOfNodes = (gNodesN+2)*(gNodesM+2);
    doubleTTPtrs Y0(1+numberOfNodes);
    DoubleTTVec YStart(numberOfNodes,0);
    Y0(numberOfNodes) = &gPuddleTemperature;
    for(long i = 0; i < numberOfNodes; i++)
    {
        Y0(i) = &YStart(i);
    }

    double t0 = 0, tN = 60, h = 0.05;
    double dp = gPuddleTemperature.value()*0.25;

    DoubleGenMat *Y, *W, *sysErr;
    DESolve(PlumeVec,t0,Y0,tN,h,&gPuddleTemperature,dp,&Y,&W,&sysErr);
    long N = Y->rows();
    FILE *resultsH = fopen("results.out","w");
    FILE *sensH = fopen (*Sensitivity.out*,"w");
    for(long j = 0; j < gNodesM+2; j++)
    {
        for(i = 0; i < gNodesN+2; i++)
        {
            fprintf(resultsH,"%12.4e\t",(*Y)(j*(N-1, gNodesN+2) + i));
            fprintf(sensH,"%12.4e\t",(*sysErr)(j*(N-1, gNodesN+2) + i));
        }
```

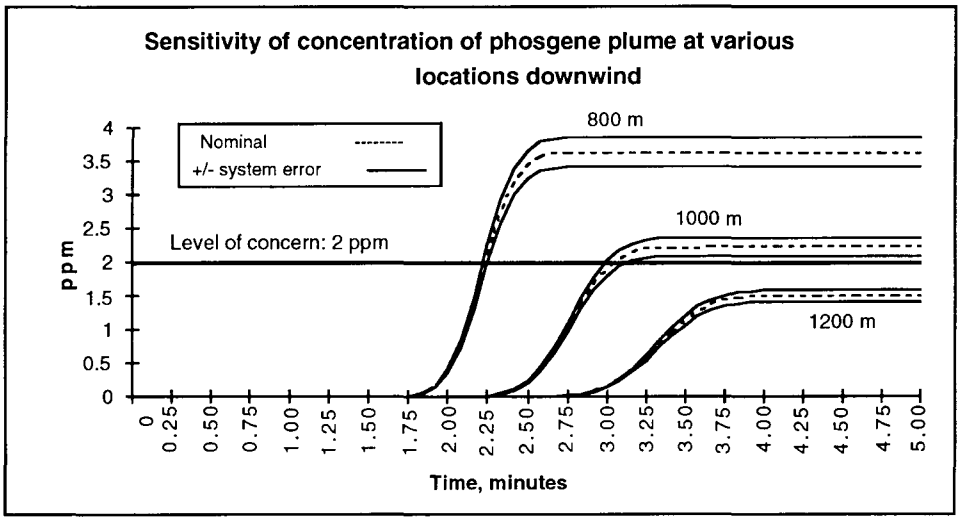

FIGURE 1 The solution to the model equations for a plume for three different downwind locations.

```
        fprintf(resultsH, "\n");
        fprintf(sensH, "\n");
    }
    fclose(resultsH);
    fclose(sensH);
}
```

Figure 1 illustrates how the output from this program can be presented. It shows the trajectories and their sensitivity to an initial puddle temperature.

6 CONCLUSION

The method presented here simplifies the task of computing the sensitivity of a dynamical system to its parameters. The user of these routines does not need to be involved in formulating the sensitivity equations and furthermore, does not need to supply any method for computing necessary derivatives. By using the table algorithm, necessary derivatives are computed exactly, and hence the first order sensitivity is computed to the accuracy of the underlying numerical differential equation solver.

ACKNOWLEDGMENTS

The authors especially wish to thank Andrzej Lewandowski, visiting scientist at NOAA from Wayne State University, Detroit, Michigan. Dr. Lewandowski's ideas provided the initial inspiration for this article and his suggestions and comments on draft versions helped immensely in clarifying ideas presented here.

REFERENCES

[1] P. M. Frank, *Introduction to System Sensitivity Theory*. New York: Academic Press, 1978.

[2] R. Kalaba and A. Tischler, "A computer program to minimize a function with many variables using computer evaluated exact higher-order derivatives," *Appl. Math. Comput.*, vol. 13, pp. 143–172, 1983.

[3] M. E. Jerrell, "Automatic differentiation in C++," *JOOP* vol. 3, pp. 17–24, 1990.

[4] R. Kalaba, L. Tesfatsion, and J. L. Wang, "A finite algorithm for the exact evaluation of higher order partial derivatives of functions of many variables," *J. Math. Anal. Appl.*, vol. 92, pp. 552–563, 1983.

[5] R. Kalaba and L. Tesfatsion, "Automatic differentiation of functions of derivatives," *Comput. Math. Appl.*, vol. 12A, pp. 1091–1103, 1986.

[6] L. B. Rall, *Automatic Differentiation Techniques and Applications* (Lecture Notes in Computer Science) Berlin: Springer, 1981.

[7] L. C. Rich and D. Hill, "Automatic differentiation in MATLAB," *Appl. Numerical Math.* vol. 9, pp. 33–43, 1992.

[8] R. D. Wilkens, "Investigation of a New Analytical Method for Numerical Derivative Evaluation," *Comm. ACM*, Vol. 7, pp. 465–471, 1964.

[9] T. Keffer, "Why C++ will replace Fortran," *Dr. Dobb's J.*, vol. 195, pp. 39–47, 1992 (Special Suppl.).

[10] Rogue Wave Software, Inc., *Rogue Wave Math.h++ Class Library Version 4.1.* Corvallis, OR: Rogue Wave Software, 1992.

[11] P. W. M. Brighton, "Evaporation from a plane liquid surface into a turbulent boundary layer," *J. Fluid Mech.*, pp. 323–345, 1985.

[12] P. I. Kawamura and D. Mackay, "The evaporation of volatile liquids," *J. Hazardous Materials*, vol. 15, pp. 343–364 (1987).

[13] E. Palazzi, M. De Faveri, G. Fumarola, and G. Ferraiolo, "Diffusion from a steady source of short duration," *Atmospheric Environment*, vol. 16, pp. 2785–2790, 1982.

[14] F. Pasquill and F. B. Smith, *Atmospheric Diffusion.* Ellis Horwood Limited, 1983.

[15] R. E. Wengert, "A simple automatic derivative evaluation program," *Comm. ACM*, vol. 7, pp. 463–464, 1964.