Available online at www.sciencedirect.com
ScienceDirect
Progress in Natural Science 19 (2009) 397–406
Progress in Natural Science
www.elsevier.com/locate/pnsc

Differentiation transforming system

Qiang Cheng [a,b,c,*], Haibin Zhang [d], Bin Wang [a], Yonghua Zhao [c]

[a] LASG, Institute of Atmospheric Physics, Chinese Academy of Sciences, Beijing 100029, PR China
[b] LSEC, Institute of Computing Mathematics, Chinese Academy of Sciences, Beijing 100080, PR China
[c] SC, Institute of Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, PR China
[d] College of Applied Sciences, Beijing University of Technology, Beijing 100124, PR China

Received 15 April 2008; received in revised form 10 July 2008; accepted 14 July 2008

Abstract

The differentiation transforming (DFT) system is developed to produce the tangent linear codes, which is used to calculate the Jacobian- and the Hessian-vector products with no truncation errors. This paper first gives the introduction of the functionality and features of the DFT system, and then discusses several techniques for the implementation of automatic differentiation tools, including data dependence analysis, singular differentiation and code optimization. Finally, the codes generated with DFT used in several applications have been demonstrated.

Keywords: Jacobian; Hessian; Tangent linear model; Automatic differentiation

1. Introduction

A large number of algorithms are closely related to the calculations of derivatives in solving the nonlinear problems. To solve an unconstrained nonlinear optimization problem, it is much indispensable to calculate the gradients of the objective function, and sometimes the Hessian or the Hessian-vector products. To solve the nonlinear equations, it is much indispensable to calculate the Jacobian of the vector-valued functions, or the Jacobian-vector products within the Jacobian-Free Newton–Krylov (JFNK) method [1]. In some other algorithms such as the Halley-like algorithms [2], the calculation of three and higher order derivatives is required. As we know, the traditional finite differencing (FD) method is inherited with truncation error whose value is much dependent on the way the values of the small increments are selected.

Automatic differentiation (AD) [3] is a technique through which the derivatives of a function that is defined by a number of computer program lines can be calculated without truncation errors. To calculate the first-order derivatives, there are two typical differentiation models, the tangent linear model and the adjoint model, which calculate derivatives in a natural way around the sequence of running the program and in a reverse way, respectively. Using the chain-rule law, the tangent linear model can be derived from a top–down accumulation of the underlying functions, while the adjoint model can be derived from a bottom–up accumulation. In practice, the tangent linear model can be directly used to calculate the Jacobian-vector products, while the adjoint model can be used to calculate gradients with ideal computational cost. By a combination of both the methods, the second-order adjoint model can be simply derived, which can be used to calculate the Hessian-vector products with the cost in terms of operations count and memory as roughly twice that of the adjoint model. So far, AD has been used in a variety of applica-

* Corresponding author. Tel.: +86 10 58812132; fax: +86 10 58812115.
E-mail address: walls@sccas.cn (Q. Cheng).

1002-0071/$ - see front matter © 2008 National Natural Science Foundation of China and Chinese Academy of Sciences. Published by Elsevier Limited and Science in China Press. All rights reserved.
doi:10.1016/j.pnsc.2008.07.012

tions such as variational data assimilation [4], parameter recognition, sensitivity analysis [5], singular vector and singular value decomposition [6], and computational fluid dynamics (CFD).

Apart from the calculation of the Jacobian-vector products, the tangent linear model can also be used to calculate Jacobian and gradients with the computational cost proportional to the number of independent variables, which is roughly twice that by using the finite differencing approximation. However, as referred to in most documents, the derivates calculated with the tangent linear model have the advantage of no truncation errors, such that the number of convergence steps in the Newton algorithm system could be dramatically reduced to nearly the optimum point. In addition, the tangent linear model is also employed in testing the correctness of the adjoint model.

Since the values of the required variables in the calculation of each derivative term can be derived from the above calculations, it is relatively easy for the implementation of the tangent linear model. However, one could meet some difficulties in the large-scale applications, such as the accuracy uncertainty of the tangent linear model and the unacceptable cost both in running time and in memory. To check the correctness of the tangent linear model, the knowledge in terms of inputs and outputs of the argument parameter list is required in advance before making a reliable testing function. Although any variable, which is independent or not, can be viewed as an input and output variable mathematically, the memory requirement of the testing function in the large-scale applications could be far beyond the limitation of the system memory if each of the argument parameters is uniformly taken as an input and output variable in this way. At the same time, the input/output relationship of global variables employed in a number of procedures cannot be calculated accurately if they are used in an implicit way, which is typically declared in a common block in Fortran 77 and used as an external variable in C/C++. In such a case, the input/output relationship of global variables is either dependent on the values of the inputs, or cannot be found out through simple static dependence analysis. For this reason, the dependence analysis in terms of inputs and outputs is one of the most challenging problems in reconstructing the tangent linear models for the large-scale applications.

Another important aspect is the data dependence analysis, which is required in seeking the sparse structure of Jacobian and Hessian, as well as in deriving the left/right seed matrix [7] as the inputs for the adjoint model and the tangent linear model, respectively. However, there are at least two aspects of difficulties. As we know, the running behavior of a program cannot be predicted in those cases including switch and selective structures. In other words, the reliability of the sparse structure of the Jacobian that is derived through static data dependence analysis can be acceptable only in smooth problems. In terms of memory and the complexity of data dependence analysis, the costs for calculating the sparse structure of the Jacobian are not acceptable if the scale of a specific application is terrifically large. Another difficulty is the data dependence analysis for array data, especially, which index set is dependent on the values of input variables, or which elements are used in different ways. Fortunately, in most instances, it is helpful to take a variety of array indexes as the uniform one in a simple way.

It will be much helpful to develop a special compiler of a specific tool for generating the tangent linear models for Fortran 77 codes with YACC [8]. Different from any other language compiler, the lexical/syntax analyzer of an AD tool is relatively smaller concentrating on the analysis of the data differentiability. Except for extracting detailed information from the process of lexical analyzing and syntax parsing, one should make a productive syntax parser for producing optimized derivative codes, which is not easy in the cases in which the structure of a program object is terrifically complicated or in which a statement is terrifically long.

So far, a number of tools have been developed for generating the tangent linear models, or the forward model in different forms, such as TAMC/TAF [9], Odyssee [10], and ADIC/ADIFOR [11]. Generally, there are two strategies for producing the tangent linear codes, i.e., operator overloading and code transformation. For the purpose of better performance of the tangent linear model, most of the tools are designed in the latter way. Different from other tools, ADIC/ADIFOR is implemented with several advanced techniques such as XAIF and SparseLinC, which can be used to calculate the entire or part of the Jacobian in an efficient way.

This paper is organized as follows: in Section 2, we first introduce the functionality and several features of the DFT software, and then analyze the structure and the style of the tangent linear model generated with the DFT software by illustrating a tutorial example. In Section 3, we concentrate on discussing several dependence analysis techniques employed in the implementation of AD tools. And in Section 4, we further discuss the details of techniques for the implementation of the DFT software. In Section 5, some differentiation costs in several applications are presented, and in Section 6, a summary of this paper is given.

2. DFT software

DFT is a source-to-source tool for generating the tangent linear model of a program defined by a number of subroutines and functions. Designed with YACC, DFT is implemented in C/C++ and supports Fortran 77, partly Fortran 90/95 extensions. DFT can also be used for generating the testing functions and analyzing the structure of the sparse Jacobian. In practice, the tangent linear model can be used for calculating the Jacobian-vector products, gradients and the Jacobian.

2.1. Functionality

2.1.1. Generation of the tangent linear model

The elementary program object transferred with DFT is a subroutine or a function, in which interface is, respectively, by default of the following forms:

- SUBROUTINE PROC(X_IN, X_OUT, X_IN_OUT, OTHERS)
- FUNCTION FUNC(X_IN, X_OUT, X_IN_OUT, OTHERS)

where the real-typed parameters X_IN, X_OUT and X_IN_OUT, respectively, denote the input variables, the output variables and the input and output variables, and OTHERS denotes a number of integer, character and logical variables. Correspondently, the interfaces of the tangent linear codes generated with DFT are by default of the following forms:

- SUBROUTINE Diff_PROC(Diff_X_IN, X_IN, Diff_X_OUT, X_OUT, Diff_X_IN_OUT, X_IN_OUT, OTHERS)
- FUNCTION Diff_FUNC(Diff_X_IN, X_IN, Diff_X_OUT, X_OUT, Diff_X_IN_OUT, X_IN_OUT, OTHERS)

where each of the real-typed variables is followed by its correspondent differentiation which is headed with the prefix "Diff_" by default. The standard structure of the tangent linear model generated with the DFT software will be illustrated in detail in Section 2.3.

2.1.2. Generation of the testing codes

There are a number of approaches to check the correctness of the tangent linear model, but the most rigid one is the gradient testing. In the gradient testing, each element of the gradient of each output variable that is calculated at any point by the tangent linear model should be well approximated to that obtained by divided differencing approximation if the norm of the increments is small enough. However, it is impossible under the limitation both in memory and in the running time if the number of the inputs and outputs is terrifically large. An alternative approach implemented in the DFT software is to check the misfit at a random point between the differentiation results that are calculated with the tangent linear model and by the difference of calling the underlying functions twice.

Without the loss of generality, and given a vector-valued function F: $R^n \to R^m$, if it is differentiable at a random point X_0, the Jacobian $F'(X_0)$ exists and

$$\frac{||F(X_0+\Delta X)-F(X_0)||}{||F'(X_0)\Delta X||} \to 1, \quad ||\Delta X|| \to 0 \tag{1}$$

To check the correctness of the tangent linear model in a statistical way, DFT can generate the testing codes of the interface as the following forms:

- SUBROUTINE Check_Diff_PROC(X_IN, X_OUT, X_IN_OUT, OTHERS)
- FUNCTION Check_Diff_FUNC(X_IN, X_OUT, X_IN_OUT, OTHERS)

where parameters are just the duplication of the underlying subroutine or function. As referred to in relation (1), the values of the input and output variables X_IN_OUT should be stored in advance before calling the underlying functions and the tangent linear model. Since the memory requirement is limited in a specific application, it is not impossible to take each of the variables in the parameter list as both an input variable and an output variable in a simple way. From this point, the IO dependence analysis is indispensable anyway.

2.1.3. Detection of sparsity

In specific applications, the sparse Jacobian matrix of a vector-valued function can be effectively calculated by using the tangent linear model or the adjoint model. By carefully analyzing the global dependence of all variables, we can derive the sparse structure of the Jacobian, as well as the left/right seed matrix which can be taken as the inputs of the tangent linear model row-by-row or the inputs of the adjoint model column-by-column, yielding a compressed Jacobian matrix with less time cost.

Besides the above descriptions, there are several auxiliary functions such as flexible running optional parameters, printing the calling tree structure of a specific subroutine and function, global data dependence analysis, code optimization by the binary reducing method, partly supporting MPI and PETSc [12], and statistical analysis for the complexity of each program objects.

2.2. Features

Apart from several auxiliary functions such as generating the testing functions, optimizing the code through the binary reducing technique, and printing the calling tree structures, DFT is much different from other tools in the following aspects.

2.2.1. Static dependence analysis

We focus on three kinds of dependence analyses, which include the input/output (IO) dependence analysis, the data dependence analysis and the procedure dependence analysis. For the effective implementation of softwares, all further discussions are limited to the static analysis approach. In addition, the iteration dependence analysis is another challenge in specific applications such as numerical iterations and recursive functions.

Apparently, the data dependency is closely related to the IO dependency. However, the former concentrates on the

dependence relationship between any two variables, while the latter is a relative notation that is employed to describe the input/output behavior of a variable within a program object defined by a segment of program lines.

By recording the IO knowledge of argument parameters in each procedure, DFT can calculate the final IO relationship of any argument parameter through deep recursive dependence analysis. This process can be represented as an iteration $A_{k+1} = A_k \oplus A_k^2$, where A_0 is the initial dependence matrix, which is proved to be of logarithmical convergence in the next section.

Through the global dependence analysis between procedures, DFT can be used to print the calling tree structure of any subroutine or function (see Fig. 1), which is much helpful in analyzing and checking the differentiation models in complicated applications.

2.2.2. Structural differentiation

The tangent linear model generated using the DFT software is of the same structure as the underlying functions, such as DO/ENDDO/CONTINUE, IF/THEN/ELSE/ENDIF, and GOTO.

The differentiation of an active variable in the tangent linear model is always accompanied with itself, as well as the differentiation statement accompanied with the underlying one. Since all the differentiation transformation is performed from structure to structure and from statement to statement, the differentiation codes are of the same structure and features such as vectorized and parallelism, data precision, and data type.

2.3. Tutorial examples

In the following, we introduce the structure of the tangent linear model by illustrating an example generated with the DFT software. Perigee is an independent subroutine in the GPS/MET Rayshooting model without calling any subroutine or function, see Fig. 2.

As illustrated by this example, the reconstruction of the tangent linear model is just a differentiation process from structure to structure and statement by statement. Specifically, each evaluation statement is followed by its differentiation correspondence. And either in the argument list or in the declarations of variable list, each real-typed variable, which is called the active variable in most documents, is always followed by its differentiation perturbation.

In the second circulation structure of the underlying subroutine shown in Fig. 2, the right-hand side expression is fairly complicated such that its differentiation correspondence in the tangent linear model is also fairly long and complicated, which causes a poor running performance. To solve this problem, the binary reducing method is employed for addressing this problem. Detailed discussion will be presented in Section 4.3.

The include statement is an implicit quotation of a segment of program lines, whose differentiation correspondence is also generated with the DFT software and encapsulated in a file with the name of the default prefix "Diff_" in the same directory. For the purpose of the IO dependence analysis, DFT extends all the include lines into the current routine in advance.

Except for the headline descriptions in each generated file, all the comment lines in the tangent linear models are removed by default. The current version of DFT supports various styles of comment lines from Fortran 77 to Fortran 90/95.

A large number of statements, such as STOP, RETURN/END, and GOTO, have the correspondent differentiation codes as simple duplications of themselves. As well as in the cases of nonlinear expressions, IF/ELSE IF conditions and subroutine/function calling statements, it is of great significance that all the values of each controlling variable in these statements should be full in accordance with those in the underlying functions.

Besides the above descriptions, the differentiation correspondence of a subroutine calling statement is just the call-

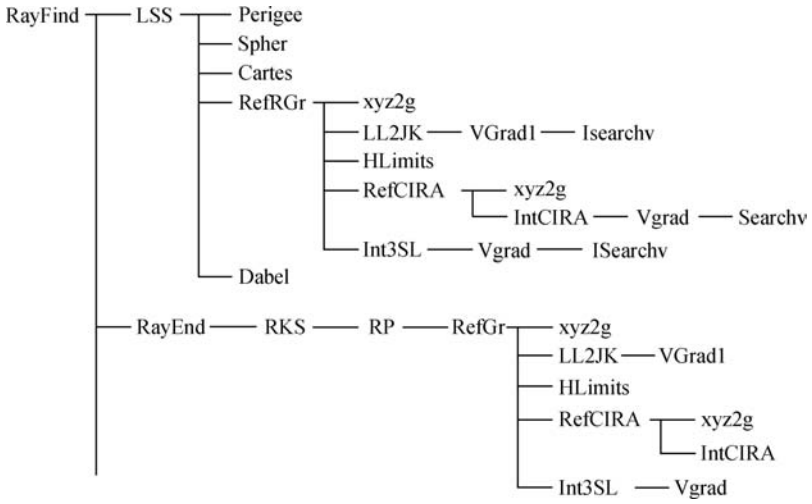

Fig. 1. A piece of the calling tree structure of the GPS/MET Rayshooting model.

```
      Subroutine perigee (X1, X2, P, NV)

*     Subroutine perigee (vectorized) calculates the ray
perigee
      vector for a straight ray connecting two given points.
      Author: M.E. Gorbunocv

 #include "./include/defaults.inc"
*Input parameters:
      Dimension X1(3,NV), X2(3, NV)
*Output parameters:
      Dimension P(3, NV)
*Local parameters:
      Parameter (NOMax = 1000)
      Dimension U(3,MaxSize), XU(MaxSize)

      Do 10 i=1,3
      Do 10 n=1, NV
      U(i,n n) = X1(i, n) - X2(i, n)
10    Continue

      Do 20 n=1, NV
      XU(n) = (X1(1,n)*U (1, n) + X1(2,n)*U (2, n) +
     >          X1(3, n)*U(3, n))/
     >          (U(1, n)**2 + U(2, n)**2 + U(3, n)**2)
20    Continue

      Do 30 i=1,3
      Do 30 n=1, NV
      P(i,n) = X1(i,n) - U(i, n)*XU(n)
30    Continue

      Return
      End
```

```
      Subroutine Diff_Perigee (Diff_X1,X1,Diff_X2,X2,
     >  Diff_P,P,NV)

#include "./include/Diff_defaults.inc"
      Dimension Diff_X1(3, NV),X1(3, NV),
     >  Diff_X2(3, NV),X2(3, NV)
      Dimension Diff_P(3, NV), P(3, NV)
      Parameter (NOMax = 1000)
      Dimension Diff_U(3,MaxSize), U(3,MaxSize),
     >  Diff_XU(MaxSize),XU(MaxSize)

      Do 10 i=1,3
      Do 10 n=1, NV
      Diff_U(i,n) = Diff_X1(i,n) – Diff_X2(i,n)
      U(i,n) = X1(i,n) - X2(i,n)
10    Continue

      Do 20 n =1, NV
      Diff_XU(n) =(((X1(1,n)*Diff_U(1,n) +Diff_X1(1,n)
     > *U(1,n) +(X1(2,n)*Diff_U(2,n) +Diff_X1(2,n)*U(2,n))
     > +(X1(3,n)*Diff_U(3,n) +Diff_X1(3,n)*U(3,n)))
     > *(U(1,n)**2 +U(2,n)**2 +U(3,n)**2) -(2.0*Diff_U(1,n)
     > *U(1,n) +2.0*Diff_U(2,n)*U(2,n) +2.0*Diff_U(3,n)
     > *U(3,n))*(X1(1,n)*U(1,n) +X1(2,n)*U(2,n) +X1(3,n)
     > *U(3,n)))/(((U(1,n)**2 +U(2,n)**2 +U(3,n)**2)*(U(1,n)
     > **2 +U(2,n)**2 +U(3,n)**2))
      XU(n) =(X1(1,n)*U(1,n) +X1(2,n)*U (2,n) +X1(3,n)
     > *U(3,n))/(U(1,n)**2 +U(2,n)**2 +U (3,n)**2)
20    Continue

      Do 30 i =1,3
      Do 30 n =1,NV
      Diff_P(i,n) =Diff_X1(i,n) -(U(i,n)*Diff_XU(n)
     > +Diff_U(i,n)*XU(n))
      P(i,n)=X1(i,n) -U(i,n)*XU(n)
30    Continue

      Return
      End
```

Fig. 2. The tangent linear model generated with the DFT software.

ing of its tangent linear subroutine, in which the calculations of all the independent variables are always followed by the calculations of their differentiation perturbations. Things are a little different in the case of function calling. Apart from the calling of the tangent linear function in the differentiation correspondence of a numerical expression, another calling of the underlying function is performed in the calculation of the underlying expression, which results in poor performance, in particular, if the running cost of the underlying function is terrifically large. Fortunately, this performance inefficiency can be avoided as the binary reducing method is applied. In addition, as to be discussed in Section 4.2, the tangent linear codes of singular functions are different, but one can quote their differentiation correspondences from the generated differentiation library in a simple way.

3. Dependence analysis

3.1. Data dependence analysis

Given a real-typed variable set $\boldsymbol{X} = \{X_1, X_2, \cdots, X_n\}$, we define an n-dimensional dependence matrix $\boldsymbol{A}$ at a given point X as follows:

$$\boldsymbol{a}_{ij} = \begin{cases} 0, & X_i \text{ is independent of } X_j \\ 1, & X_i \text{ is dependent on } X_j \end{cases}$$

where X_i is independent of X_j, which denotes that to arbitrary values of $\boldsymbol{X}$, we have $\partial X_i/\partial X_j = 0$, and X_i is dependent on X_j, which denotes that there exists at least one point in the definition space of $\boldsymbol{X}$, and we have $\partial X_i/\partial X_j \neq 0$. For further discussions, we specially define the addition operation and the multiplication operation on $\{0,1\}$, which satisfy the following relations $\forall a, b \in \{0,1\}$, we have

(1) $1 \oplus 0 = 1, \quad 0 \oplus 0 = 0, \quad 1 \oplus 1 = 1$
(2) $a \oplus b = b \oplus a$
(3) $1 \cdot 0 = 0, 0 \cdot 0 = 0, 1 \cdot 1 = 1$
(4) $a \cdot b = b \cdot a$
(5) $a \oplus (a \cdot b) = a$

Assume that $\boldsymbol{F}$ is defined as a mapping from $\boldsymbol{X}$ to $\boldsymbol{X}$, F: $X \to X$, the dependence matrix $\boldsymbol{A}$ is just a representation of the sparse structure of the extended Jacobian matrix of $\boldsymbol{F}$. According to the implicit function theory, to arbitrary X_i and X_j, $1 \leqslant i, j \leqslant n$, we have

$$\frac{\partial X_i}{\partial X_j} = \left(\frac{\partial F}{\partial X_i} \cdot \frac{\partial X_i}{\partial X_j} + \frac{\partial F}{\partial X_j}\right) + \sum_{k \neq i,j} \frac{\partial X_i}{\partial X_k} \cdot \frac{\partial X_k}{\partial X_j} \tag{2}$$

where the first terms that are bracketed on the right-hand-side are the direct derivatives of X_i with respect to X_j. From the above relation we can derive the following result

$$\begin{aligned} a_{ij} &= a_{ij} \oplus \sum_{k \neq i,j} a_{ik} \cdot a_{kj} \\ &= a_{ij} \oplus (a_{ii} \cdot a_{ij}) \oplus (a_{ij} \cdot a_{jj}) \oplus \sum_{k \neq i,j} a_{ik} \cdot a_{kj} \\ &= a_{ij} \oplus \sum a_{ik} \cdot a_{kj} \end{aligned} \tag{3}$$

If we define the initial dependence matrix $\boldsymbol{A}_0$ that only contains information about the direct derivatives of X_i with

respect to X_j, $1 \leqslant i, j \leqslant n$, we can calculate the final dependence matrix in finite steps with the following iteration relations:

$$a_{ij}^{m+1} = a_{ij}^{m} \oplus \sum a_{ik}^{m} \cdot a_{kj}^{m}. \tag{4}$$

Rewriting the above relation in matrix formation, we have

$$A_{m+1} = A_m \oplus A_m^2. \tag{5}$$

Algorithm 1 (*Static data dependence analysis*).

(1) Given the initial dependence matrix A_0 from the direct analysis the dependency between each pair of variables in $\boldsymbol{X}$;
(2) Update with $A_{k+1} = A_k \oplus A_k^2$;
(3) If $A_{k+1} = A_k$, stop; else $k = k + 1$ and go to (2)

Lemma 1. *Let D denote the maximum number of the nodes on the computational path with respect to any two nodes on the computational graph. To calculate the final dependence matrix through the above iteration relation, the number of iteration steps will be no more than* $\log_2 D$.

Proof. If X_i is dependent on X_j, the proof should only derive $a_{ij} = 1$ after finite steps by the above algorithm. Suppose $X_{l_1} \to X_{l_2} \to \cdots \to X_{l_m}$ is the shortest computational path from X_i to X_j, where $l_1 = i$, $l_m = j$, $m \leqslant D$. From the initial dependence matrix, we have $a_{l_k,l_{k+1}} = 1, 1 \leq k \leq m-1$. After an iteration step with (5), to arbitrary $1 \leqslant k \leqslant m-2$, we have

$$a_{l_k,l_{k+2}} = a_{l_k,l_{k+2}} \oplus \sum_{p \neq k,k+2} a_{l_k,l_p} \cdot a_{l_p,l_{k+2}} = a_{l_k,l_{k+1}} \cdot a_{l_{k+1},l_{k+2}} = 1 \tag{6}$$

That is, we derive $X_{l_1} \to X_{l_3} \to X_{l_5} \to \cdots \to X_{l_m}$ as another shortest computational path from X_i to X_j, while the number of nodes in the path is only half of that in the last step. Thus, if we repeat the above process for at most $\log_2 m$ steps, we can finally derive $a_{l_1,l_m} = 1$. □

3.2. IO dependence analysis

3.2.1. Definitions

Within a specific segment of program lines, a variable is called the input variable if it has never been evaluated and is quoted for one or more than one time; a variable is called the output variable if it has been evaluated before it is quoted; a variable is called the input and output variable if it has been quoted for one or more than one time before it has been evaluated.

From this point of view, a variable may have different IO types in different segments of program lines. In the static dependence analysis with respect to IO, suppose that we have the knowledge about all IO types of a specific variable within all the subroutines and functions, it would be a problem to find out the final IO relationship of this variable from all these IO information. To address this problem, we first present a basic algorithm in the following.

3.2.2. Basic algorithm

Here, we present an algorithm to calculate the IO type of a specific variable statement by statement. For simplicity, the IO type of an input/output/input and output variable is again marked with IN/OUT/IN_OUT, and at first marked by default UNKNOWN. Suppose that we have known the IO type of a variable before and in the current statement, then we can calculate its final IO relationship through the following rules, see Table 1.

As presented in Table 1, if a variable is used as an output variable or an input and output variable before the current statement, its final IO type is always output or input and output, respectively, no matter what relation it is in the following program lines.

3.2.3. Variables in the argument list

The IO type of variables that are defined in the argument list can be calculated through the above algorithm in a recursive way. With respect to each of the local lines within a specific subroutine and function, we can calculate the IO type of a specific argument variable statement by statement. However, if the current line contains a subroutine or function calling, we can calculate the IO type of each argument variable in the same way, which is correspondent with the same position as in the parameter list of the called subroutine or function. By repeating this process until the bottom nodes in the calling tree, we can finally find out the final IO relationship of this argument variable.

However, there is an efficient way to calculate the IO types of all the argument variables within each of the subroutines or functions in a calling tree if we perform this process from bottom to top. Actually, in the implementation of the DFT software, the IO knowledge of all the argument variables within each subroutine and function is calculated after the first sweep of the underlying functions, as well as of those variables defined in the common blocks.

3.2.4. Variables in common blocks

A variable defined in the common block shares the same memory space throughout the program, whose value could be rewritten and quoted in one or more subroutines and functions. Since the trace of a common block variable cannot be located in most instances, the

Table 1
Basic rules for calculation of the IO relationship.

IO relationship		Current			
		UNKNOWN	IN	OUT	IN_OUT
Before	UNKNOWN	UNKNOWN	IN	OUT	IN_OUT
	IN	IN	IN	IN_OUT	IN_OUT
	OUT	OUT	OUT	OUT	OUT
	IN_OUT	IN_OUT	IN_OUT	IN_OUT	IN_OUT

above algorithm as per the argument parameter list could meet difficulties when it is directly used for analyzing the IO relationship.

Roughly speaking, there are generally two sorts of communication schemes in terms of the common blocks. The first one is the explicit scheme, in which a common block defined in a specific subroutine or function marked A is also defined in those subroutines or functions that call A in a direct way. Actually, we can simply calculate the final IO relationship of variables in the common block in the same way as variables in the argument parameter list.

The other one is the implicit scheme, in which the common block defined in a specific subroutine or function marked A is also defined in those subroutines or functions that do not call A in a direct way. Suppose that we have known information about the local calling tree of each subroutine or function, we can simply duplicate the information of the common blocks within the current subroutine or function to all its father nodes, and perform the same algorithm as in the explicit scheme. However, a large number of duplications occur if the number of subroutines and functions that contain the common blocks is relatively small within the calling tree structure. In this way, it must result in less efficiency.

Another approach is to record the traces of all the common blocks during a sweep of the calling tree structure. With regard to a specific common block, we can derive a list of subroutines and functions in the nature sequence from such trace information, for instance, a probable list as "RayFind → Perigee → Perigee → RefRGr → VGrad → VGrad → RefRGr → RKS → VGrad → VGrad →RKS → RayFind" within the calling tree as depicted in Fig. 1. From the list, we can derive a sub-list for the common block within a specific subroutine or function, for instance, "RKS → VGrad → VGrad → RKS". With the help of the sub-list, we can find out the final IO relationship of the common block variables within this subroutine or function through the above approaches. Based on the above discussions, we present an algorithm in the following. For the purpose of efficiency, we should similarly perform this algorithm on a calling tree form bottom to top.

Algorithm 2 (*Static IO dependence analysis for common block variables*)**.**

(1) Given a specific subroutine or function, produce its local calling tree structure;
(2) Given a specific common block, derive a list of subroutines and functions by analyzing its trace in the calling tree;
(3) Given a specific subroutine or function, derive a sub-list of subroutines and functions from the above list;
(4) Given a specific variable in the common block within the subroutine or function, derive a list of IO types from a sweep of the sub-list of subroutines and functions;
(5) Perform the above rules on the list of IO types step by step and finally we can find out the final IO relationship of this variable;
(6) Repeat the above steps if necessary.

4. Implementation details

4.1. Architecture design

The DFT software is implemented on the YACC platform and run on LINUX/UNIX, which is composed of five major parts, i.e., the lexical analyzer, the syntax parser, static dependence analysis, the differentiation library and differentiation code transformation, see Fig. 3. The following descriptions are intended to address the running mechanism for helping the users to produce efficient differentiation codes by using the DFT software productively.

4.1.1. Lexical analyzer

A typical lexical analyzer is specially designed for extracting identifiers from flows of program statements, as well as for recognizing the types of these identifiers. Identifiers are generally divided into three sorts. One is the system keywords, such as IF/ELSE/ENDIF, DO/ENDDO, MAX/MIN, and GOTO; the other two are constants and variables, i.e., differentiable and undifferentiable.

In most instances, it is essential to extract information about logical and character constants/variables from a number of statements. For this, we consider about four types of identifiers, which include differentiable, undifferentiable, character and logical constants/variables. To avoid ambiguousness, the type of the undifferentiable identifiers is specially referred to both the integer type and the unknown type. Apart from the digit numbers and the system keepwords, the types of the other identifiers can be recognized from the information list extracted from the complicated static data dependence analysis.

4.1.2. Syntax parser

Unlike the tools implementation for the reverse differentiation models, the design of the syntax parser for reconstructing the forward models is mainly focused on the differentiation of variables, expressions and statements.

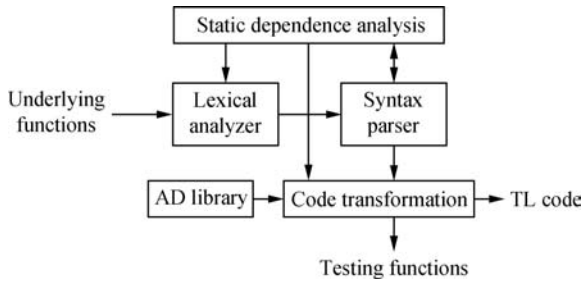

Fig. 3. Structure of the differentiation transforming system.

From the viewpoint of computer science, a typical syntax parse is specially designed for shifting/reducing different syntax elements of a specific language grammar including advanced identifiers, phrases, and expressions, statements. According to the classification of identifiers, each of these syntax elements is uniformly divided into four types, which include differentiable, undifferentiable, character and logical.

Roughly speaking, the most simplified syntax parser that can only recognize the numerical expressions and a small number of statements such as headlines and include lines can nearly deal with everything in the reconstruction of the tangent linear model. For the purpose of good performance, it is essential to design a relatively complete language compiler for extracting information from the underlying functions as much as we can. The extracted information includes the different complexities of the underlying functions, the IO knowledge of variables, and the types of different program objects. In addition, a refined syntax parser can also help to produce terse codes. Thus, a good syntax parser can not only remove all the syntax shift/reduce conflicts, but also recognize various types of each program object in a correct way.

For the purpose of terse codes, we also pay attention to another characteristic of expressions particularly, which can effectively avoid the bracket redundancies in complicated differentiation expressions. In this methodology, expressions are generally divided into three types, which include additive expressions, half-additive expressions and unadditive expressions. Detailed discussions have been presented in the users' manual.

4.1.3. Static dependence analysis

Due to the procedure dependency between subroutines and functions, the DFT software should be running twice. In the first running, it is essential to extract, find out and record the necessary information about each program object at any position. As described above, all kinds of dependence analyses are not only essential in analyzing different types of program objects in a correct way, but are also helpful in producing efficient resulting codes in the second running. Besides the dependence analysis discussed above, there are other related things including the data type analysis and the object complexity analysis.

The data type analysis involves the finding out of the data type of each identifier at any position, which is used in these cases including shifting and reducing a series of strings, analyzing the data types of parameters in the calling subroutines or functions, and producing the testing functions. In addition, the data type of each variable can be dynamically inquired from the variable information list, and all the final data types of both the argument variables and the common block variables in each subroutine or function are stored into files after the first running.

The object complexity analysis is helpful in dramatically reducing the memory requirement of the DFT software in the second running, which concerns a number of program characteristics such as the number of argument variables, the number of the common blocks and the number of the local variables, the number of subroutines and functions in the underlying functions, and the maximum length of variables.

4.1.4. AD library

The AD library is a mechanism that can dynamically produce the differentiation correspondences of all the singular functions and the nonsingular functions, useful interfaces, and general testing codes.

4.1.5. Code transformation

In the second running of the DFT software, the underlying function is transformed into the tangent linear model statement by statement. Concretely, the lexical analyzer first reads a complete statement and partitions it into a number of identifiers, and finds out the data type of each identifier through the data dependence analysis. Then the syntax parser shifts/reduces these identifiers and their differentiation correspondences, and finally forms into the differentiation statement that is followed by the original one.

4.2. Singular differentiation

There are a number of singular functions in Fortran, which is also called the on/off problems in some other documentations. For simplicity, we particularly present the equivalent differentiation of five singular functions as listed in Table 2. Besides, the differentiation correspondences of the other singular functions, including sqrt, asin, and acos, are provided in the differentiation library, which are uniformly headed with the prefix "Diff_" by default. In addition, the differentiation correspondence of the max/min function that is of more than two parameters is also presented in the differentiation library.

With respect to the differentiation values at the singular points, one can simply set it at zero to keep the calculation satiability. However, one can also set different values in a specific application. Some interesting documentations can be found in related documents.

4.3. Code optimization

For the purpose of better performance, several techniques are used in the optimization of the tangent linear model, based on the following considerations. First, compared with the underlying functions, the number of operations in the tangent linear model is roughly doubled anywhere. As a result, the data locality of the original program lines could be ruined, since each statement is followed by its differentiation correspondence. As we know, it is really a big challenge to arrange the data locality of a specific program in the best way. However, one cannot neglect this problem if the running cost of a segment of program lines is very time-consuming.

Table 2
Equivalent differentiation of singular functions.

Singular function	Equivalent differentiation	Singular points
abs(x)	sign$(1.0, x)\,\mathrm{d}x$	$x = 0.0$
sign(x, y)	sign$(1.0, xy)\,\mathrm{d}x$	$xy = 0.0$
dim(x, y)	$(1.0 - \mathrm{sign}(1.0, y - x)) \cdot 0.5(\mathrm{d}x - \mathrm{d}y)$	$x \leqslant y$
max(x, y)	$(\mathrm{d}x + \mathrm{d}y + (\mathrm{d}x - \mathrm{d}y) \cdot \mathrm{sign}(1.0, x - y)) \cdot 0.5$	$x = y$
min(x, y)	$(\mathrm{d}x + \mathrm{d}y - (\mathrm{d}x - \mathrm{d}y) \cdot \mathrm{sign}(1.0, x - y)) \cdot 0.5$	$x = y$

Second, if a variable is locally used as a constant, some related calculations are unnecessary. By default, each variable declared as a real type is taken as the active variable, which is always accompanied by its differentiation elsewhere. However, if a variable is declared as a constant before running the DFT software, its differentiation as well as the related calculations will be removed from the tangent linear model.

Third, if we perform the differentiation on a complicated expression in the usual way, the differentiation correspondence is not acceptable in some cases. By representation of a segment of evaluation statements with the computational graph, the best implementation could be found. In the following, we specially discuss the binary reducing method.

We first compare two different differentiation approaches that are depicted in Fig. 4. The underlying function is composed of a typical statement that calculates the product of five independent variables. The differentiation calculation in the left implementation needs operations including 18 mult and 4 add, while that in the right implementation needs 12 mult and 4 add. However, if we calculate the product of n independent variables, the operations in the differentiation calculation will be $(n-1)\cdot(n+4)/2$ mult and $n-1$ add in the left implementation and $3n-3$ mult and $n-1$ add in the right implementation, respectively.

Roughly speaking, the binary reducing method is a technique which reduces each basic operation such as add, sub, mult, divide and power into a uniform identifier at the stage of syntax parsing. This technique can be automatically selected with respect to the complexity of a statement, or can be activated from the running parameter options.

5. Applications

In the following, we only present the differentiation costs of the tangent linear models in five applications, which include the 3D-VAR for the GPS/MET Raytracing model [4], the 4D-VAR for the GPS/MET Rayshooting model [13], sensitivity analysis for the AGCM model [14], implicit solving of the spherical barotropic atmospheric shallow water equations [15] and the VB orbital optimization with the XIAMEN software [16]. The costs shown in Fig. 5 are roughly 1.35–2.44 times that of the underlying functions.

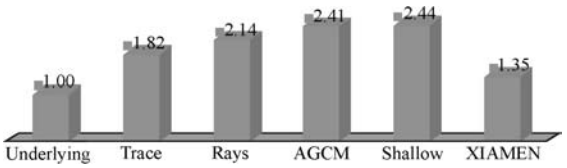

Fig. 5. The running costs of the tangent linear models in several applications.

We use the Jacobian-free Newton–Krylov method to solve the shallow water equations, in which the tangent linear model is directly used to calculate the Jacobian-vector products. In the other four applications, the tangent linear models are mainly used for testing the correctness of the adjoint models. In the XIAMEN software, the tangent linear model is also used instead of the finite differencing method to calculate the gradients of the cost function.

6. Summary and conclusions

The tangent linear models generated with the DFT software can be used in calculating the Jacobian- and the Hessian-vector products, and in testing the correctness of the adjoint models and sensitivity analysis. Some auxiliary functions of the DFT software are helpful in most applications such as automatically generating the testing functions, printing the calling tree structures, and detecting the structure of the sparse Jacobian. From the initial dependence matrix, the final dependence matrix can be calculated with logarithmical convergence complexity through the algorithm presented in Section 3.1. Another algorithm that is presented in Section 3.2 can be used to find out the final IO relationship of a specific variable, no matter what it is used in the local part of a subroutine or function, or declared in the argument parameter list, or defined in the common block. In terms of code optimization, several techniques are applied such as the binary reducing method, variable used as a constant parameter, and equivalent differentiation for singular functions. Finally, we have pre-

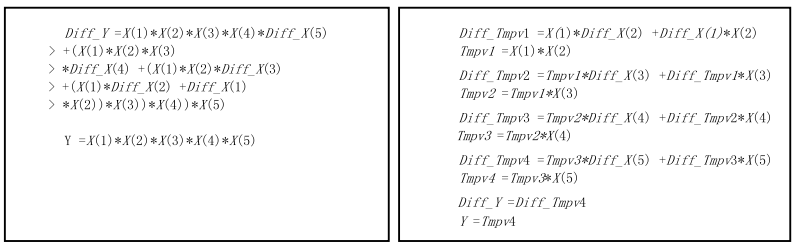

Fig. 4. Code optimization through the binary reducing method.

sented the differentiation costs of five applications to verify the performance of the DFT software.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (Grant No. 60503031) and the National Basic Research Program of China (2004CB418304). The authors thank Dr. Xing Xie, Dr. Zaizhong Ma and Xiaoyu Chen for their helpful suggestions and work.

References

[1] Knoll DA, Keys DE. Jacobian-free Newton–Krylov methods: a survey of approaches and applications. J Comput Phys 2004;193(2):357–97.
[2] Yamamoto T. On the method of tangent hyperbolas in Banach spaces. J Comput Appl Math 1998;21:75–86.
[3] Griewank A. Evaluating derivatives: principles and techniques of automatic differentiation. 1st ed. Philadelphia: SIAM; 2000.
[4] Zou XL. Use of GPS/MET refraction angles in three-dimensional variational analysis. Quart J Roy Meteor Soc 2000;126: 3013–40.
[5] Bischof C, Pusch G, Knoesel R. Sensitivity analysis of the MM5 weather model using automatic differentiation. Comput Phys 1996; 89:605–12.
[6] Mu M. Nonlinear singular vectors and nonlinear singular values. Sci China D 2000;43:375–85.
[7] Newsam GN, Ramsdell JD. Estimation of sparse Jacobian matrices. SIAM J Algebraic Discrete Methods 1983;4:404–17.
[8] Levine J, Mason T. Lex and Yacc. 2nd ed. O_Reilly and Associate Inc. 1992, Sebastopol, CA.
[9] Giering R, Kaminski T. Recipes for adjoint code construction. ACM Trans Math Software 1998;24(4):437–74.
[10] Rostating N, Dalmas S, Galligo A. Automatic differentiation in odyssee. Tellus 1993;45(A):558–68.
[11] Bischof C, Carle A, Corliss G, et al. ADIFOR: generating derivative codes for Fortran programs. Sci Program 1992;1(1):11–29.
[12] Balay S. PETSc 2.0 users manual. Technical Report ANL-95/11-revision 2.1.3. Argonne National Laboratory; May 2003.
[13] Zou XL, Vandenberghe F, Wang B, et al. A raytracing operator and it's adjoint for the use of GPS/MET refraction angle measurements. J Geophys Res Atmos 1999;104(D18):22301–18.
[14] Wu GX, Liu H, Zhao YC, et al. A nine-layer atmospheric general circulation model and its performance. J Adv Atmos Sci 1996;13(1):1–18.
[15] Wang B, Ji ZZ. Construction and numerical tests for the multi-conservation difference scheme. Chinese Sci Bull 2003;48(10):1016–20.
[16] Song LC, Luo Y, Dong KM, et al. Paired-permanent approach for VB theory (II). Sci China B 2001;44(6):561–70.