April 26, 1996

VARIATIONAL DATA ASSIMILATION WITH THE LORENZ MODEL

Xiang-Yu Huang and Xiaohua Yang

Danish Meteorological Institute, Copenhagen, Denmark

ABSTRACT

This note is intended as an introduction to various aspects of variational data assimilation using the adjoint model technique, in particular the principles and formulation of an adjoint model. A complete assimilation system for the Lorenz equations is formulated. The Lorenz model is chosen for its simplicity in structure and the dynamic similarities with the nonlinear atmospheric system. Detailed derivations are performed to introduce basic concepts of the adjoint model method, and the general procedure to construct an adjoint variational assimilation is discussed. The coding process to construct the Lorenz assimilation is demonstrated in detail, in which the line-by-line approach for adjoint model coding is introduced. With the Lorenz assimilation model, various numerical experiments including coding checks are performed and results discussed. The FORTRAN source code for the constructed assimilation is given in an appendix. It is believed that by designing, coding and running the variational data assimilation for the Lorenz model, newcomers may gain useful insights and experiences which will be beneficial for the research and development work of a more complicated variational assimilation system, such as the four dimensional data assimilation of an operational weather prediction model.

CONTENTS

1. Introduction 5

2. Formulation of a variational assimilation 6

2.1 The Lorenz model 6

2.2 Formulation of the assimilation problem for the Lorenz model 7

2.3 Adjoint method to estimate the cost function gradient 8

2.4 Minimization procedure using adjoint method 11

2.5 Application of adjoint model approach in assimilation problem 11

2.6 Construction of a variational assimilation system 13

3. Coding of an adjoint variational assimilation system 15

3.1 Nonlinear model 15

3.2 Tangent linear model 16

3.3 Adjoint model 17

4. Results and discussion 22

4.1 True solutions 22

4.2 Coding check 24

4.3 Variational assimilation experiments 26

4.4 Minimization methods 32

5. Summary 36

Acknowledgments 36

Appendix A: An alternative derivation of the cost function gradient 37

Appendix B: FORTRAN code for the Lorenz VDA system 38

References 42

.

1. INTRODUCTION

In recent years variational data assimilation (VDA) techniques have been increasingly applied to the numerical weather prediction (NWP) problem [see, *e.g.*, the review by Ghil & Malanotte-Rizzoli (1991), and a VDA reference list by Courtier *et al* (1993)]. For an operational numerical weather prediction model using the primitive equations, a successful data assimilation should produce initial model states that not only makes optimal use of observational data, but also satisfies physical laws and is free of undesirable noise. Traditional data assimilation approaches such as function fitting, successive corrections, optimal interpolation *etc.* (Daley, 1991), use different kinds of interpolation techniques, subjected to various physical balance constraints, and directly project spatially irregular observational data onto a regular coordinate space. In the VDA approach, the assimilation problem is redefined as an iterative process in which the gap between observed fields and the initial model state is minimized. The so-called adjoint model, which is derived from the forecast model (forward model), is often used in the minimization of a VDA.

For newcomers working with an adjoint VDA problem, it is often difficult to get a precise understanding of the basics, such as the concept of tangent linear operator, adjoint model, cost function gradient, *etc.* It is not a trivial task, either, to formulate a VDA problem and program it correctly. Unfortunately, there exists few suitable textbooks on the adjoint model formulation and coding. Despite the fact that the adjoint method has been introduced into meteorological and oceanographic applications for decades, we still see researchers attaching appendices with coding examples or alike in their journal articles or reports (*e.g.*, Li *et al*, 1991, Navon *et al*, 1992). With that in mind, we write this note on the formulation and coding of a VDA system using the adjoint model method. Detailed description is given to illustrate practical procedures as well as principles to implement the adjoint technique to a forecast system. We hope that in this way, the note will make it easier for beginners to grasp the basics of the VDA approach, especially the practical aspects of the adjoint model coding.

In designing a complete VDA system we use, as an assimilation model, the equation set proposed by Lorenz (1963). The Lorenz model is chosen because it is simple in structure, yet rich in solution patterns, and it has many similarities with the nonlinear dynamics of the atmospheric system. Over the past decades, a wide variety of nonlinear

dynamic studies has been done for this model. In the data assimilation field, it has also been a favored research subject (see, *e.g.*, Gauthier, 1992, Stensrud and Bao, 1992, Pires *et al*, 1996). Our purpose here is to use the Lorenz model to illustrate the procedure to apply the variational method to a data assimilation problem. The Lorenz model contains only three ordinary differential equations, and it is therefore easy to construct a complete VDA system and explain the details in a note. Furthermore, the precise model solution is also readily obtained for verifying the VDA coding.

In Section 2, we formulate the assimilation problem for the Lorenz model and introduce the principles of variational assimilation. The application of the adjoint model approach in a general VDA application is discussed, followed by a description of the general procedure to construct a variational assimilation. In Section 3, we describe in detail the coding procedure for the Lorenz VDA. Various experimental results including coding checks are discussed in Section 4. In Section 5, a brief summary is given. The Fortran source code of the constructed Lorenz VDA system is given in Appendix B.

2. FORMULATION OF A VARIATIONAL ASSIMILATION

2.1. The Lorenz model

In the classical paper by Lorenz (1963), a finite system of deterministic equations (hereafter referred to as the Lorenz model or Lorenz equations) is designed, to represent a forced, dissipative hydrodynamic flow. The system consists of a set of three nonlinear ordinary differential equations, which are derived from a highly truncated spectral model of thermal convection:

$$\frac{dw(1)}{dt} = -pw(1) + pw(2), \tag{1}$$

$$\frac{dw(2)}{dt} = [r - w(3)]w(1) - w(2), \tag{2}$$

$$\frac{dw(3)}{dt} = w(1)w(2) - bw(3), \tag{3}$$

where t is the non-dimensional time, p the Prandtl number, r the modified Rayleigh number, b the aspect ratio, $w(1)$ the intensity of convection, $w(2)$ the maximum temperature difference and $w(3)$ the stratification change due to convection. Despite its simplicity, this nonlinear equation system is chaotic, dissipative and rich in solution patterns, and shares many dynamic features with the atmospheric system. For such an equation system

with bounded solutions, Lorenz found that non-periodic solutions are ordinarily unstable with respect to small modifications, so that slightly differing initial states can evolve into considerably different states. In light of the results, the question about the feasibility of an extended range weather prediction was raised for the first time, which has attracted the attention of the meteorological community over several decades.

2.2. Formulation of the assimilation problem for the Lorenz model

Assume that our task is to find the evolution trajectory of the model state $\mathbf{w}_k$ described by the equations (1) - (3), where

$$\mathbf{w} = \begin{pmatrix} w(1) \\ w(2) \\ w(3) \end{pmatrix},$$

and subscript k represents the time step. Obviously, if the initial model state $\mathbf{w}_0$ is known, with a time stepping method, the numerical solutions for the future model state $\mathbf{w}_k$ can be readily obtained by integration.

It is easy to see that the Lorenz model is a deterministic problem in which the future evolution trajectory of the model state is uniquely determined by the initial condition. In fact, the following general equation form may be used to represent such initial value problems,

$$\mathbf{w}_k^f = \mathbf{N}(\mathbf{w}_{k-1}^f), \tag{4}$$

where $\mathbf{N}(\mathbf{w})$ represents all modeled processes for one time interval between k and its neighboring time step $k-1$, and superscript f represents the model solution. Eq. (4) is often referred to as a forward model in data assimilation studies. Obviously, it implies that $\mathbf{w}_k^f$ is ultimately determined by the initial state $\mathbf{w}_0^f$:

$$\mathbf{w}_k^f = \mathbf{N} \cdots \mathbf{N}(\mathbf{w}_0^f).$$

Consider now the situation in which the initial model state $\mathbf{w}_0$ is unknown. Instead, there are some "observations" for the trajectory at certain point in time, $\mathbf{w}_k^o$, where the superscript o stands for observation. This represents a typical data assimilation scenario in which the ultimate goal is to use available observations to obtain model solutions as close to the true evolution trajectory as possible. Among many ways to formulate such a task in a mathematical language, with a variational approach, a cost function J is

introduced to measure the misfit between the model state and observations,

$$J = \frac{1}{2} \sum_{k=0}^{kmax} (\mathbf{w}_k^f - \mathbf{w}_k^o)^T (\mathbf{w}_k^f - \mathbf{w}_k^o), \tag{5}$$

where $(\ )^T$ is the matrix transpose operator, and $\mathbf{w}_k^o$ has the same dimension as $\mathbf{w}_k^f$. The components of $\mathbf{w}_k^o$ are set to be equal to those of $\mathbf{w}_k^f$ in case of missing observations. Since the exact initial state $\mathbf{w}_0$ is unknown in a data assimilation problem, we may only start with a guess state for $\mathbf{w}_0^f$, which is either from observations, or from model solutions formed through iterations.

Eqs. (4) and (5) show that, for a deterministic initial value problem such as the Lorenz model, the essence of the data assimilation is to adjust the initial state, $\mathbf{w}_0^f$, so that J is minimized. The following minimization algorithm may be designed to relate the cost function J to the model states obtained from the iterative process,

$$\mathbf{w}_0^{f,n+1} = \mathbf{w}_0^{f,n} - \alpha \nabla J(\mathbf{w}_0^{f,n}), \tag{6}$$

where n is the number of iterations, α is a parameter to be chosen to achieve effective convergence of the iteration, and ∇J is the gradient of the cost function with respect to the initial state $\mathbf{w}_0^{f,n}$. If the iteration (6) converges, $\mathbf{w}_0^{f,n}$ will approach the desired initial state $\mathbf{w}_0^{f,\infty}$ which satisfies

$$J = min(J).$$

2.3 Adjoint method to estimate the cost function gradient

With the VDA minimization designed as in Eq. (6), the important issue is to estimate the gradient of the cost function with respect to a model state, ∇J. A straightforward way would be to perturb each control variable in turn and estimate the resulting change of the cost function. However, such an estimate yields only an approximate gradient, which is often insufficient to achieve convergence of the minimization. Furthermore, this approach is not feasible if such a VDA algorithm is to be used in a more general application such as an atmospheric model, which is characterized by an enormous model dimension [O(10^6)]. As an alternative, the so-called adjoint model method may be used to achieve an exact cost function gradient, through an integration of the adjoint model backwards in time.

Consider a small perturbation to the model state $\mathbf{w}_0^f$, denoted as $\mathbf{w}_0^{f,tl}$. The perturbation results in a change in the cost function $J(\mathbf{w}_0^f)$, denoted as $J^{tl}(\mathbf{w}_0^f)$:

$$J^{tl}(\mathbf{w}_0^f) = J(\mathbf{w}_0^f + \mathbf{w}_0^{f,tl}) - J(\mathbf{w}_0^f). \tag{7}$$

Taking the limit of $\|\mathbf{w}_0^{f,tl}\| \longrightarrow 0$, the above becomes,

$$J^{tl}(\mathbf{w}_0^f) = [\nabla J(\mathbf{w}_0^f)]^T \mathbf{w}_0^{f,tl}. \tag{8}$$

At the same time, using the definition of $J(\mathbf{w}_0^f)$ in (5), (7) can also be written as

$$J^{tl}(\mathbf{w}_0^f) = \sum_k (\mathbf{w}_k^f - \mathbf{w}_k^o)^T \mathbf{w}_k^{f,tl}. \tag{9}$$

Combining (8) and (9), we get

$$[\nabla J(\mathbf{w}_0^f)]^T \mathbf{w}_0^{f,tl} = \sum_k (\mathbf{w}_k^f - \mathbf{w}_k^o)^T \mathbf{w}_k^{f,tl}. \tag{10}$$

In principle, Equation (10) may now be used to compute the gradient of cost function $J(\mathbf{w}_0^f)$, provided that a relationship between $\mathbf{w}_k^{f,tl}$ and $\mathbf{w}_0^{f,tl}$ is found. To find that relationship, we do the derivation in the next step. For simplicity, the generalized forward model (4) is used.

We linearize (4) around the basic state $\mathbf{w}_k^f$:

$$\mathbf{w}_{k+1}^{f,tl} = \frac{\partial \mathbf{N}(\mathbf{w}_k^f)}{\partial \mathbf{w}_k^f} \mathbf{w}_k^{f,tl} = \mathbf{L}(\mathbf{w}_k^f)\mathbf{w}_k^{f,tl}, \tag{11}$$

where $\mathbf{L}(\mathbf{w}_k^f)$ is the tangent linear operator (TL) which depends on the basic state $\mathbf{w}_k^f$ and time step k. Eq. (11) is known as the tangent linear equations of the forward model (4). Using the tangent linear model (11) iteratively, the relation between $\mathbf{w}_k^{f,tl}$ and $\mathbf{w}_0^{f,tl}$ is established:

$$\begin{aligned} \mathbf{w}_k^{f,tl} &= \mathbf{L}(\mathbf{w}_{k-1}^f)\mathbf{w}_{k-1}^f = \mathbf{L}(\mathbf{w}_{k-1}^f)\mathbf{L}(\mathbf{w}_{k-2}^f)\mathbf{w}_{k-2}^{f,tl} = \cdots \\ &= \mathbf{L}(\mathbf{w}_{k-1}^f)\mathbf{L}(\mathbf{w}_{k-2}^f)\cdots\mathbf{L}(\mathbf{w}_1^f)\mathbf{L}(\mathbf{w}_0^f)\mathbf{w}_0^{f,tl}, \end{aligned}$$

or

$$\mathbf{w}_k^{f,tl} = \mathbf{L}_k \mathbf{w}_0^{f,tl}, \tag{12}$$

where,

$$\mathbf{L}_k = \mathbf{L}(\mathbf{w}_{k-1}^f)\mathbf{L}(\mathbf{w}_{k-2}^f)\cdots\mathbf{L}(\mathbf{w}_1^f)\mathbf{L}(\mathbf{w}_0^f). \tag{13}$$

Substituting (12) into (10), we get:

$$[\nabla J(\mathbf{w}_0^f)]^T \mathbf{w}_0^{f,tl} = \sum_k (\mathbf{w}_k^f - \mathbf{w}_k^o)^T \mathbf{L}_k \mathbf{w}_0^{f,tl},$$

that is,

$$[\nabla J(\mathbf{w}_0^f)]^T = \sum_k (\mathbf{w}_k^f - \mathbf{w}_k^o)^T \mathbf{L}_k,$$

or

$$\nabla J(\mathbf{w}_0^f) = \sum_k \mathbf{L}_k^T (\mathbf{w}_k^f - \mathbf{w}_k^o), \tag{14}$$

where $\mathbf{L}_k^T$ is the transpose of $\mathbf{L}_k$

$$\mathbf{L}_k^T = \mathbf{L}^T(\mathbf{w}_0^f)\mathbf{L}^T(\mathbf{w}_1^f)\cdots\mathbf{L}^T(\mathbf{w}_{k-2}^f)\mathbf{L}^T(\mathbf{w}_{k-1}^f). \tag{15}$$

The transpose of the tangent linear operator is called the adjoint operator. Notice that in (15) the order of the time index of the adjoint operator is reversed, compare to (13). By defining the adjoint model (AD) as

$$\mathbf{w}_{k-1}^{f,ad} = \mathbf{L}^T(\mathbf{w}_{k-1}^f)\mathbf{w}_k^{f,ad}, \tag{16}$$

the solution at time 0 is

$$\mathbf{w}_0^{f,ad} = \mathbf{L}_k^T \mathbf{w}_k^{f,ad}, \tag{17}$$

where $\mathbf{w}_k^{f,ad}$ denotes the adjoint variable at time step k. The adjoint equation (17) has the same form as the tangent linear equation (12) but with two differences: the linear operator in (17) is the transpose of the linear operator in (12); and the time sequence is reversed in (17). If we now initialize the adjoint model with the difference between the basic state $\mathbf{w}_k^f$ and the observation $\mathbf{w}_k^o$ at time step k, that is, assuming

$$\mathbf{w}_k^{f,ad} = \mathbf{w}_k^f - \mathbf{w}_k^o, \tag{18}$$

and integrate the adjoint model (16) to time step 0, the result will be

$$\mathbf{L}_k^T(\mathbf{w}_k^f - \mathbf{w}_k^o),$$

which is exactly the expression in the summation on the right hand side of (14). In other words, one adjoint integration with $(\mathbf{w}_k^f - \mathbf{w}_k^o)$ as input gives the contribution of $(\mathbf{w}_k^f - \mathbf{w}_k^o)$ to the gradient of the cost function at time step k. Due to the linearity of the adjoint model, it is not necessary to compute $(\mathbf{w}_k^f - \mathbf{w}_k^o)$ one by one for the summation in (14).

The gradient of the cost function is the final output at time 0 after only one integration of the adjoint model (16), starting from zero and adding $(\mathbf{w}_k^f - \mathbf{w}_k^o)$ during the integration. It is easy to see that a backwards adjoint model integration is of a complexity similar to a single integration of the forward model.

In fact, the above is only one of the many methods to derive the cost function gradient using the adjoint model. In Appendix A, an alternative derivation is presented.

2.4 Minimization procedure using adjoint method

With the method to calculate the cost function gradient established, the minimization step (6) may be performed to achieve the optimal initial state $\mathbf{w}_0$ through iteration. Figure 1 demonstrates schematically the process to apply the adjoint model technique to the VDA minimization. Assume we have an observation $\mathbf{w}_k^o$, $k = 0, 1, \ldots, kmax$. Start with an initial guess state, $\mathbf{w}_0^{f,0}$. Integrate the forecast model $\boxed{\text{NL}}$ (1) - (3) from $\mathbf{w}_0^{f,0}$ at time step 0, to time step $kmax$. This integration results in a basic state $\mathbf{w}_k^{f,0}$, $k = 0, 1, \ldots, kmax$. Then, integrate the adjoint model $\boxed{\text{AD}}$, which is in form of (16), from time step $kmax$ to time 0. During the integration, add the difference between the basic state $\mathbf{w}_k^{f,0}$ and the observation $\mathbf{w}_k^o$, $(\mathbf{w}_k^{f,0} - \mathbf{w}_k^o)$, to the adjoint variables. According to (14) and (16), this integration will result in the gradient of the cost function with respect to the initial state, $\nabla J(\mathbf{w}_0^{f,0})$. Put $\mathbf{w}_0^{f,0}$ and $\nabla J(\mathbf{w}_0^{f,0})$ into the minimization procedure $\boxed{\text{MIN}}$, as defined in (6). The minimization will yield a new initial state $\mathbf{w}_0^{f,1}$. The above procedure may then be repeated until the desired minimum of J is achieved.

In the Lorenz VDA study here, we take the parameter α in the minimization equation (6) as a constant. The simple minimization method with a constant α value is known as the steepest descent approach. In practical applications such as an operational weather forecast system, more efficient minimization methods, in which α is a non-constant and determined by other considerations, are necessary.

2.5 Application of adjoint model approach in assimilation problem

Obviously, the above adjoint model technique also has a potential in application to more sophisticated assimilation problems. In fact, the method has been used in many data assimilation applications in the NWP field [see, *e.g.*, the extensive adjoint VDA reference list compiled by Courtier *et al*, (1993)]. The procedure shown in (14) and

(18) provides an easy way to incorporate any observational data collected within the assimilation period. The method is particularly useful for the four dimensional variational data assimilation (4DVAR) of an operational weather forecast model, in which temporally as well as spatially irregular observations may be conveniently used.

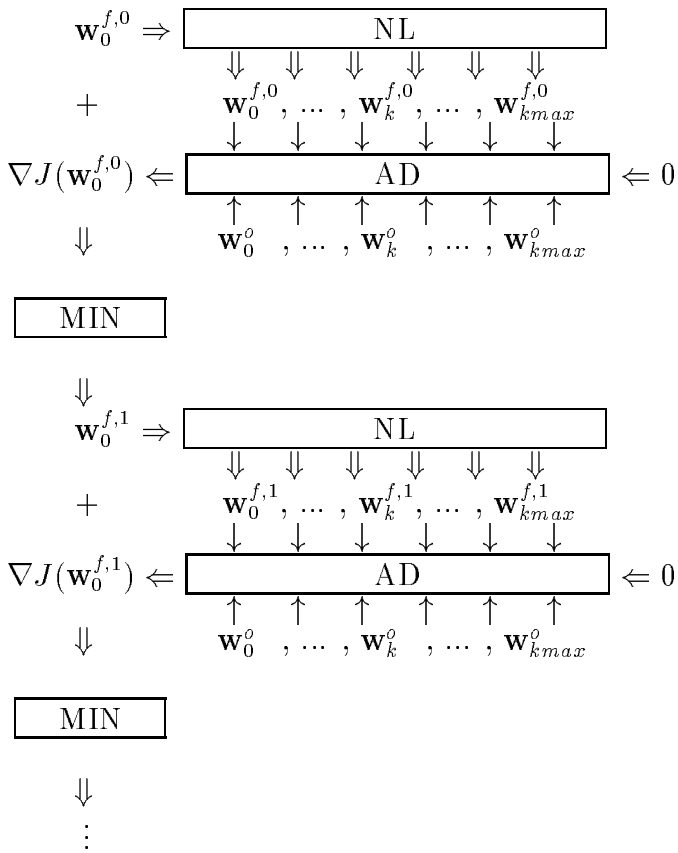

Figure 1: Flow chart of the minimization process.

In many practical applications, the cost function J, as a measure of the misfit between model solutions and observations, may be defined in various ways, reflecting certain constraints and conditions [see, *e.g.*, the discussions in Daley (1991)]. As an example, in a more general application, the cost function may be redefined as below, in order to take into account errors in observations, and also the situation in which the observation sites do not coincide with the model grid mesh,

$$J = \frac{1}{2} \sum_{k=0}^{kmax} (H_k \mathbf{w}_k^f - \mathbf{w}_k^o)^T O_k (H_k \mathbf{w}_k^f - \mathbf{w}_k^o), \tag{19}$$

where the model vector $\mathbf{w}_k^f$ is of a dimension N, the observation $\mathbf{w}_k^o$ has a dimension M, and O is a diagonal matrix with dimension $M \times M$, containing weighting coefficients

measuring observation errors. Here the definition of the cost function has been modified to reflect the fact that the observation is normally done over an irregular network which is often different from the model grid mesh. Hence, an additional matrix H in (19), with a dimension of $M \times N$, is needed to interpolate the model grid values to the observation grid so as to make a comparison. Notice that in a realistic assimilation problem, most often $M < N$, that is, the degrees of freedom of the model state is normally larger than those of the observations, so that the problem is under-determined. In case of missing observations at some points, the corresponding component in O should be set to zero. The individual component of O for each observation must be determined to reflect the data quality, which is often a complicated process in applications such as NWP.

With the definition of the cost function J modified, the gradient ∇J may be derived in a similar way as described in Subsection 2.3. For the cost function (19), the corresponding gradient equation for (14) becomes,

$$\nabla J(\mathbf{w}_0^f) = \sum_k \mathbf{L}_k^T H_k^T O_k (H_k \mathbf{w}_k^f - \mathbf{w}_k^o), \tag{20}$$

and for (18),

$$\mathbf{w}_k^{f,ad} = H_k^T O_k (H_k \mathbf{w}_k^f - \mathbf{w}_k^o).$$

To calculate the cost function gradient according to (20), similar to the procedure discussed in Subsection 2.3, the weighted difference between the model state and the observation is used as the input for the adjoint model integration. Notice that $O_k(H_k \mathbf{w}_k^f - \mathbf{w}_k^o)$, which is calculated on the observation sites, should be multiplied by H^T, so as to be transformed back to the model grid mesh.

2.6 Construction of a variational assimilation system

In light of the above discussions, generally, in constructing a VDA system, the following steps may be considered:

Step I : restructure the nonlinear forecast model (NL)

In most applications, the forecast model, which is often nonlinear, already exists. However, the model may have been under development for a long time, so that the computer code contains various standards, and it thus needs to be restructured, especially if some kind of automatic code generator is to be used.

Step II : construct the tangent linear model (TL)

In Subsection 2.3, we introduced the concept of a tangent linear model in deriving the adjoint model. The TL model is not directly used in the VDA minimization as formulated in this paper. However, since an AD operator is the transpose of an TL operator, when coding an AD model, it is often useful to first construct an TL model. The TL could also be used to check the correctness of the adjoint code, as will be discussed in Step III.

To develop the TL model from the forward model, one could start from the forward equations to derive the linearized TL model. Alternatively, one can start directly from the existing NL code and write TL code line by line. The latter is a more practical approach for full scale models, but users have to be very careful to avoid coding errors.

To verify a TL code is important and straightforward. Running the forward model NL from two slightly different initial states, $\mathbf{w}$ and $\mathbf{w}+\Delta\mathbf{w}$, we get two nonlinear solutions, $\mathbf{N}(\mathbf{w})$ and $\mathbf{N}(\mathbf{w}+\Delta\mathbf{w})$. Then, run the TL model from $\Delta\mathbf{w}$, using $\mathbf{N}(\mathbf{w})$ as the basic state, to produce a TL solution $\mathbf{L}(\Delta\mathbf{w})$. If the TL model is correctly coded, the following relation should hold approximately,

$$\mathbf{N}(\mathbf{w}+\Delta\mathbf{w}) - \mathbf{N}(\mathbf{w}) \approx \mathbf{L}(\Delta\mathbf{w}). \tag{21}$$

Step III : construct the adjoint model (AD)

To get an adjoint model, the easiest way is to start from the TL model in matrix form and transpose the TL matrix, and the AD model is readily obtained. Alternatively, one may use the line-by-line approach to derive the model, which will be illustrated later.

Unlike the TL code, the correctness of an adjoint code can be checked exactly. If the adjoint code has been formulated correctly, the following should hold [1]:

$$(\mathbf{LX})^T(\mathbf{LX}) = \mathbf{X}^T[\mathbf{L}^T(\mathbf{LX})], \tag{22}$$

where $\mathbf{X}$ is an arbitrary model state vector. Both $\mathbf{L}$ and $\mathbf{L}^T$ are linear operators and may be comprised of many linear operators on their own, representing different processes, such as subroutines, functions, DO loops or the whole program. During the development of an

[1] in fact, the above condition is merely necessary, but not sufficient(Hans Hvas, DMI, personal communication). Instead, as a sufficient condition, for two arbitrary model states, $\mathbf{X}$ and $\mathbf{Y}$, there is the relationship, $< \mathbf{X}, \mathbf{LY} > = < \mathbf{L}^T\mathbf{X}, \mathbf{Y} >$, where $< \, , \, >$ is the inner product.

adjoint code, one generally needs to test each individual segment of the code first, and the whole code afterwards, using (22).

Step IV : gradient check

As shown in Subsection 2.3, the backward integration of the adjoint model, with the difference between observation and model solution added in, yields the gradient of the cost function, ∇J. In a sophisticated model system, one may have to write the adjoint code for each individual part of the model. Experience has shown that serious coding mistakes may occur with the data exchange among different portions of the adjoint codes. Even when the AD code for individual operator has been verified, the final gradient of the cost function with respect to the control variables might still be incorrect due to improper linking of AD segments. Therefore, the accuracy of the cost function gradient ∇J needs to be checked. For that purpose, a function $\Psi(\alpha)$ may be used (Navon et al, 1992):

$$\Psi(\alpha) = \frac{J[\mathbf{w} - \alpha \nabla J(\mathbf{w})] - J(\mathbf{w})}{< -\alpha \nabla J(\mathbf{w}), \nabla J(\mathbf{w}) >} = 1 + O(\alpha), \tag{23}$$

where $< \, , \, >$ denotes the inner product. For values of α which are small but not too close to machine zero, the above value is expected to be close to unity.

3. CODING OF AN ADJOINT VARIATIONAL ASSIMILATION SYSTEM

Having formulated the variational data assimilation system for the Lorenz equations, we now need to code the numerical models.

3.1. Nonlinear model

A simple FORTRAN subroutine can be constructed for (1) - (3), using a similar notation:

```
subroutine Lorenz(w,dwdt,n,p,r,b)
dimension w(n),dwdt(n)
dwdt(1) = -p*w(1) + p*w(2)
dwdt(2) = w(1)*(r-w(3)) -w(2)
dwdt(3) = w(1)*w(2) - b*w(3)
return
end
```

Note that in the above subroutine, following the usual practice in NWP model, we choose to use the time derivative $\frac{d\mathbf{w}}{dt}$ explicitly in the code (Alternatively, one may avoid using this

term by discretizing it). Given an initial model state of $w(1)$, $w(2)$, and $w(3)$, with a time stepping method, the numerical solutions for the equations are achieved by integration.

3.2. Tangent linear model

To formulate the tangent linear model of (1) - (3), we start from the equation set, using $w(n)$ as basic state and $w^{tl}(n)$ as tangent linear variables, and derive the following equations,

$$[\frac{dw(1)}{dt}]^{tl} = -pw^{tl}(1) + pw^{tl}(2), \tag{24}$$

$$[\frac{dw(2)}{dt}]^{tl} = [r - w(3)]w^{tl}(1) - w^{tl}(2) - w(1)w^{tl}(3), \tag{25}$$

$$[\frac{dw(3)}{dt}]^{tl} = w(2)w^{tl}(1) + w(1)w^{tl}(2) - bw^{tl}(3). \tag{26}$$

In addition, remembering that a variational perturbation is always done to the **whole** model state vector, the following relationships are implied in the tangent linear operation,

$$w(1)^{tl} = w(1)^{tl}, \tag{27}$$

$$w(2)^{tl} = w(2)^{tl}, \tag{28}$$

$$w(3)^{tl} = w(3)^{tl}. \tag{29}$$

The equations above may also be written in matrix form,

$$\begin{pmatrix} w(1)^{tl} \\ w(2)^{tl} \\ w(3)^{tl} \\ \frac{dw(1)}{dt}^{tl} \\ \frac{dw(2)}{dt}^{tl} \\ \frac{dw(3)}{dt}^{tl} \end{pmatrix} = \mathbf{L} \begin{pmatrix} w(1)^{tl} \\ w(2)^{tl} \\ w(3)^{tl} \\ \frac{dw(1)}{dt}^{tl} \\ \frac{dw(2)}{dt}^{tl} \\ \frac{dw(3)}{dt}^{tl} \end{pmatrix}, \tag{30}$$

where

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ -p & p & 0 & 0 & 0 & 0 \\ r - w(3) & -1 & -w(1) & 0 & 0 & 0 \\ w(2) & w(1) & -b & 0 & 0 & 0 \end{bmatrix}.$$

The corresponding FORTRAN subroutine can be written as follows,

```
      subroutine lorenz_tl(w,n,p,r,b,w_tl,dwdt_tl)
      dimension w(n)
```

```
dimension w_tl(n),dwdt_tl(n)
w_tl(1)    =          w_tl(1)
w_tl(2)    =                        w_tl(2)
w_tl(3)    =                                        w_tl(3)
dwdt_tl(1) =       -p*w_tl(1)    + p*w_tl(2)
dwdt_tl(2) = (r-w(3))*w_tl(1)      - w_tl(2) - w(1)*w_tl(3)
dwdt_tl(3) =     w(2)*w_tl(1) + w(1)*w_tl(2)    - b*w_tl(3)
return
end .
```

Obviously, the first three statements in the above code is unnecessary for the tangent linear model and is omitted in the real code. However, they become meaningful in the adjoint model coding, as will be discussed below.

Having understood the concept of a tangent linear operation, it is rather straightforward to derive a TL from the forward model NL. An experienced user may be able to do it directly from the forward code.

3.3. Adjoint model

3.3.1. Adjoint code derived with matrix operation

According to the relationship (16), the adjoint model (AD) can be most conveniently derived from the matrix form of the TL model (30), by simply transposing the matrix $\mathbf{L}$,

$$\begin{pmatrix} w(1)^{ad} \\ w(2)^{ad} \\ w(3)^{ad} \\ \frac{dw(1)}{dt}^{ad} \\ \frac{dw(2)}{dt}^{ad} \\ \frac{dw(3)}{dt}^{ad} \end{pmatrix} = \mathbf{L}^T \begin{pmatrix} w(1)^{ad} \\ w(2)^{ad} \\ w(3)^{ad} \\ \frac{dw(1)}{dt}^{ad} \\ \frac{dw(2)}{dt}^{ad} \\ \frac{dw(3)}{dt}^{ad} \end{pmatrix},$$

where ad represents adjoint vector, and $\mathbf{L}^T$ is,

$$\mathbf{L}^T = \begin{bmatrix} 1 & 0 & 0 & -p & r-w(3) & w(2) \\ 0 & 1 & 0 & p & -1 & w(1) \\ 0 & 0 & 1 & 0 & -w(1) & -b \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

This gives,

$$w(1)^{ad} = w(1)^{ad} - p[\frac{dw(1)}{dt}]^{ad} + [r-w(3)][\frac{dw(2)}{dt}]^{ad} + w(2)[\frac{dw(3)}{dt}]^{ad},$$

$$w(2)^{ad} = w(2)^{ad} + p[\frac{dw(1)}{dt}]^{ad} - [\frac{dw(2)}{dt}]^{ad} + w(1)[\frac{dw(3)}{dt}]^{ad},$$

$$w(3)^{ad} = w(3)^{ad} - w(1)[\frac{dw(2)}{dt}]^{ad} - b[\frac{dw(3)}{dt}]^{ad},$$

$$[\frac{dw(1)}{dt}]^{ad} = 0,$$

$$[\frac{dw(2)}{dt}]^{ad} = 0,$$

$$[\frac{dw(3)}{dt}]^{ad} = 0.$$

The corresponding FORTRAN subroutine of the adjoint model is,

```
subroutine lorenz_ad(w,n,p,r,b,w_ad,dwdt_ad)
dimension w(n)
dimension w_ad(n),dwdt_ad(n)
w_ad(1) = w_ad(1) -p*dwdt_ad(1) + (r-w(3))*dwdt_ad(2) + w(2)*dwdt_ad(3)
w_ad(2) = w_ad(2) +p*dwdt_ad(1)          - dwdt_ad(2) + w(1)*dwdt_ad(3)
w_ad(3) = w_ad(3)                   - w(1)*dwdt_ad(2)    - b*dwdt_ad(3)
dwdt_ad(1) = 0
dwdt_ad(2) = 0
dwdt_ad(3) = 0
return
end
```

Notice that, corresponding to (27) - (29), we get the last three assignment statements in the above routine to set zero value to the $\frac{d\mathbf{w}}{dt}^{ad}$ vector. This may look tricky, but omitting them will be a serious mistake. We will come back to that in the following.

3.3.2. Line-by-line approach in adjoint coding

The above procedure of deriving adjoint code is rather straightforward in principle. However, with a complex system such as an operational weather forecasting model, it is very hard to work with a TL or AD matrix with a large number of rows and columns. In recent years automatic adjoint code generators have been developed for atmospheric data assimilation purposes (*e.g.*, Rostaing *et al*, 1993). However, there still seems to be some time until the automatic technique can be used to routine applications. The majority of researchers working with adjoint coding today still have to do it manually. An often used approach is the so-called line-by-line approach, in which one starts from a constructed TL routine which is derived from the forward (NL) model, and code the corresponding AD model in the reverse order, line by line. We illustrate the principle with simple examples below.

First look at a copy statement in the NL model,

```
A = B     ,
```

where A and B are scalars. Here the model state vector comprises two variables A and B. For this assignment statement, the corresponding TL is,

$$\begin{pmatrix} B^{tl} \\ A^{tl} \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} * \begin{pmatrix} B^{tl} \\ A^{tl} \end{pmatrix},$$

hence the AD is,

$$\begin{pmatrix} B^{ad} \\ A^{ad} \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} * \begin{pmatrix} B^{ad} \\ A^{ad} \end{pmatrix},$$

thus the FORTRAN code is,

```
B_ad =     A_ad  + B_ad
A_ad = 0     .
```

Next, for a statement

```
C = coef1 * A + coef2 * B + coef3 * D     ,
```

where C, D are also scalar variables, and $coef1$, $coef2$ and $coef3$ are coefficients that do not change themselves with the operation, it can be shown that the adjoint code should be,

```
A_ad =    A_ad                    + coef1 * C_ad
B_ad =               B_ad         + coef2 * C_ad
D_ad =                            + coef3 * C_ad     +  D_ad
C_ad = 0
```

If we combine the above two forward statements together in the following order,

```
C = coef1 * A + coef2 * B + coef3 * D
A = B
```

The matrix form of the TL for the combined operation will be,

$$\begin{pmatrix} B^{tl} \\ A^{tl} \\ D^{tl} \\ C^{tl} \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ coef2 & coef1 & coef3 & 0 \end{bmatrix} * \begin{pmatrix} B^{tl} \\ A^{tl} \\ D^{tl} \\ C^{tl} \end{pmatrix},$$

the corresponding AD operation is,

$$\begin{pmatrix} B^{ad} \\ A^{ad} \\ D^{ad} \\ C^{ad} \end{pmatrix} = \begin{bmatrix} 1 & 1 & 0 & coef2 \\ 0 & 0 & 0 & coef1 \\ 0 & 0 & 1 & coef3 \\ 0 & 0 & 0 & 0 \end{bmatrix} * \begin{pmatrix} B^{ad} \\ A^{ad} \\ D^{ad} \\ C^{ad} \end{pmatrix},$$

hence the adjoint code is,

```
B_ad =    B_ad      + A_ad                      + coef2 * C_ad
A_ad =                                          + coef1 * C_ad
D_ad =                            + D_ad        + coef3 * C_ad
C_ad = 0
```

It is easy to see that these statements are identical to the combination of the two individual adjoint codes given above, provided that the order of those two statements is reversed. Note that attention needs to be given to the order of each adjoint statement appearing in the code above.

Indeed, with a piece of TL code at hand, it is feasible to construct the adjoint code in the way demonstrated above, that is, to start from the last line of the tangent linear code, and construct the corresponding adjoint code line by line in the reverse order. This means that, for a DO loop, the operation should normally be run in the reversed index order. Roughly speaking, for each TL statement in the form of,

```
Xj_tl = c1 * X1_tl + c2 * X2_tl +  ... + ci * Xi_tl + ... cm * Xm_tl    ,
```

where $c1, c2, \ldots, ci, \ldots, cm$ are coefficients, and $i = 1, 2, \ldots, m$, $1 \leq j \leq m$, there should be a corresponding segment of adjoint code in the following form,

```
Xm_ad = Xm_ad      + cm * Xj_ad
...
Xi_ad = Xi_ad      + ci * Xj_ad
...
X2_ad = X2_ad      + c2 * Xj_ad
X1_ad = X1_ad      + c1 * Xj_ad
Xj_ad =              cj * Xj_ad    .
```

that is, for each TL variable Xi^{tl} that has appeared on the right hand side (r.h.s) of the TL statement, there should be an AD statement in which the corresponding adjoint Xi^{ad} appears on the left hand side (l.h.s), and Xi^{ad} equals to the adjoint itself (Xi^{ad}) plus the

product of the coefficient (ci) and the adjoint of the l.h.s variable in TL statement (Xj^{ad}). In order to memorize this, one just has to remember, in writing AD code, to switch the positions of the l.h.s variable Xj with that of the r.h.s variable Xi of the TL equation. Notice that the AD statement for the l.h.s variable of the TL model Xj^{ad} should always appear at the end of the AD segment, and if Xj^{tl} does not appear on the r.h.s of the TL at the same time, (that is, if $cj = 0$ in the above example), a statement should still be written to assign zero value to its adjoint, otherwise it may cause errors in other parts of the program.

It should be pointed out that, unlike a derivative operation, the variational operation used in connection with the adjoint model derivation always implies a perturbation around the *whole* model state vector at a certain time step. Therefore, when constructing a TL model from a NL model, whether it is for a simple statement, a DO loop, a subroutine, or an entire program segment, one needs to make sure that the whole model state is included in the model state vector, when the TL operator matrix is constructed. For instance, in the first example given above, the simple copy statement assigns the value of B to variable A. If we omit B from model state vector just because B did not change its value in that operation, the AD code will be wrong.

3.3.3. Adjoint code derived with line-by-line approach

Back to the adjoint coding for the Lorenz equations, following the line-by-line approach discussed above, we may write the corresponding adjoint statement in the reversed order, as compared to the TL code.

First, the adjoint code for the third equation is,

```
w_ad(1) = w_ad(1) + w(2)*dwdt_ad(3)
w_ad(2) = w_ad(2) + w(1)*dwdt_ad(3)
w_ad(3) = w_ad(3)    - b*dwdt_ad(3)
dwdt_ad(3) = 0.
```

then, for the second equation,

```
w_ad(1) = w_ad(1) + (r-w(3))*dwdt_ad(2)
w_ad(2) = w_ad(2)           - dwdt_ad(2)
w_ad(3) = w_ad(3)     - w(1)*dwdt_ad(2)
dwdt_ad(2) = 0.
```

and for the first equation,

```
w_ad(1) = w_ad(1) -p*dwdt_ad(1)
w_ad(2) = w_ad(2) +p*dwdt_ad(1)
dwdt_ad(1) = 0.
```

It is easy to see that after some restructuring, the above adjoint code derived with line-by-line approach becomes identical with the one derived from operations with transposed TL operator.

4. RESULTS AND DISCUSSIONS

4.1 True solutions

Using a simple forward time scheme, the solution of the Lorenz equations can be readily obtained by specifying an initial model state and a suitable time step. In case 1, we choose the parameters in the Lorenz equation set (1) – (3) as follows,

```
p  = 10., r  = 32., b  = 2.66666667,
```

and assume the initial model state to be

```
w(1) = 1.00 ,   w(2) =   3.00 ,   w(3) =  5.00.
```

For a forward time integration, we adopt a non-dimensional time increment of 0.01. The integration length is taken as 500 time steps, which corresponds to five non-dimensional time units. The model solution is plotted in Figure 2(a) as thick solid lines for all three variables $w(1)$, $w(2)$ and $w(3)$.

It should be noted that, according to Lorenz (1963), at the given value of p and b as above, the corresponding critical Rayleigh number is 24.74, which means that any Rayleigh number r that is larger than 24.74 will make the system a chaotic one, *i.e.*, the resulting model state trajectories may differ significantly from other ones with slightly different initial values. With a r value of 32 as chosen here, the system solution is expected to be in that way, which is also similar to the behavior of an atmospheric system. To illustrate this point more clearly, we make another run, case 2, where the initial condition differs from case 1 with an offset of 10 % for all model variables,

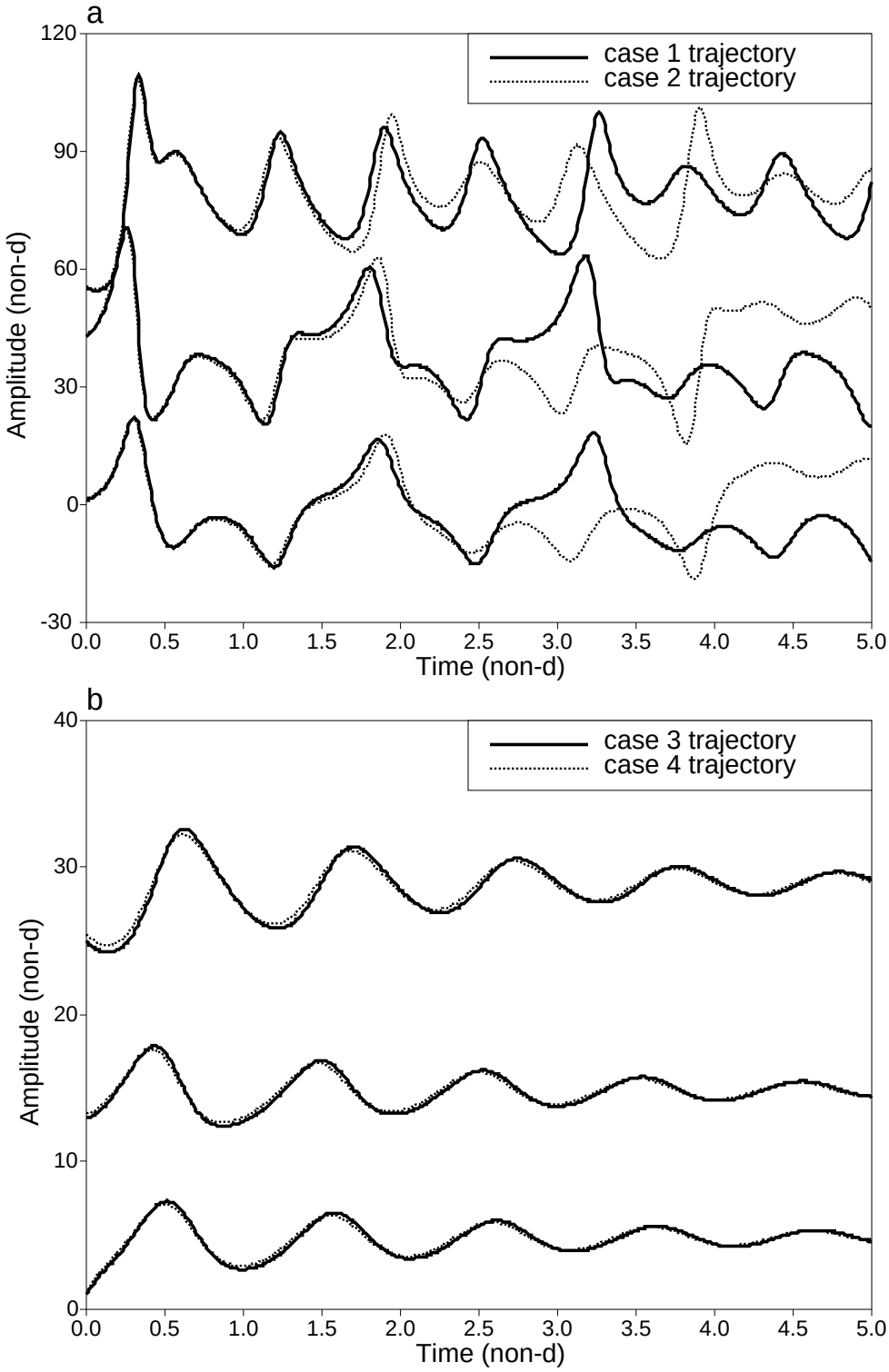

Figure 2: (a) Solutions of the Lorenz equations with $p = 10$, $b = 8/3$, and $r = 32$. r is larger than the critical Rayleigh number. Note that the lower curves are for $w(1)$, the middle curves for w(2) with an offset +40, the upper curves w(3) with an offset +50. (b) Same as in (a) but $r = 10$, which is smaller than the critical Rayleigh number. The $w(2)$ curves have been offset by +10, and $w(3)$ by +20.

```
w(1) = 1.10 ,   w(2) =   3.30 ,   w(3) =  5.50.
```

The model result for this case is also plotted in Figure 2(a), represented by dotted lines. As expected, the trajectories for case 1 and 2 are very close to each other at the start, but the difference increases substantially at the later stage of the integration.

As a contrast, when r is smaller than the critical Rayleigh number, the bifurcation in model solution no longer occurs. In case 3 and case 4, we run a parallel test, with all parameters and numerical specifications the same as in case 1 and case 2, but with a Rayleigh number r of 10, which is smaller than the critical Rayleigh number. The initial model state of case 3 is the same as case 1, and case 4 the same as case 2. The results are shown in Figure 2(b), with solid lines for case 3, and dotted lines for case 4. The trajectories of these two cases are seen to remain very close to each other throughout the integration period.

In the following experiments of this section, we will consider case 1 as a control case, and refer the VDA run for that case as control run, and discuss various numerical results from the constructed VDA model.

4.2 Coding check

We first perform code verification to make sure the VDA system has been coded correctly. Since the AD code here is derived from the tangent linear model, we begin with the validation of the TL, following the method discussed in Subsection 2.6. Table 1 shows the result obtained with the TL check for control run (case 1), at the DMI Convex 3880 computer, with the double precision compiling option (64 bits). The perturbation vector $\Delta\mathbf{w}$ is chosen as 0.1% and 1% relative to the "true" initial model states, respectively. In Table 1, all calculated values given are for the 200th time step, which is chosen in the later variational assimilation experiments as the length of the assimilation window. We see that for a TL perturbation of 0.1% over the initial model state, the difference between $\mathbf{N}(\mathbf{w}+\Delta\mathbf{w})-\mathbf{N}(\mathbf{w})$ and $\mathbf{L}(\Delta\mathbf{w})$ is fairly small, but not zero. At the perturbation of 1%, the differences are more substantial. The differences in these two quantities may be explained by the fact that a TL operation is a limit process by definition and there is no strict equality between the r.h.s and l.h.s of (21).

On the other hand, the correctness of the adjoint model can be verified rather precisely, according to (22). Table 2 shows the output of the adjoint check for the same case. For both perturbation vectors, the two terms are practically identical until the last digit, where the difference is attributed to the computer truncation error.

	$\Delta\mathbf{w}$	$\mathbf{N}(\mathbf{w}+\Delta\mathbf{w})-\mathbf{N}(\mathbf{w})$	$\mathbf{L}(\Delta\mathbf{w})$
w(1)	0.001	1.384879008488671E-002	1.366098997302821E-002
w(2)	0.003	-2.533392899875597E-002	-2.535304379766051E-002
w(3)	0.005	3.898631904370831E-002	3.877479805054089E-002
w(1)	0.01	0.156208419286204	0.136609899730282
w(2)	0.03	-0.251276994362486	-0.253530437976605
w(3)	0.05	0.409706718624186	0.387747980505408

Table 1. Validation of the tangent linear model.

$\Delta\mathbf{w}$	(0.001, 0.003, 0.005)	(0.01, 0.03, 0.05)
$(\mathbf{LX})^T(\mathbf{LX})$	2.332884440709498E-003	0.233288444070948
$\mathbf{X}^T[\mathbf{L}^T(\mathbf{LX})]$	2.332884440709488E-003	0.233288444070949

Table 2. Validation of the adjoint model.

The above AD check is performed for the whole code segment involving the AD operation, that is, a 200 time step forward tangent linear integration and a 200 time step backward adjoint integration. In a more complex system such as an operational weather prediction model, instead of checking the whole adjoint code together, it is often more practical to perform individual verification of the AD code, piece by piece, corresponding to an arbitrary number of steps in the NL model.

Next, we also perform cost function gradient check according to (23). The solid lines in Figure 3 shows the result obtained by performing such a calculation on the Convex 3880. In this plot of gradient Ψ *vs* α value, it shows that when α is in the range between 10^{-17} and 10^{-3}, Ψ remains close to unity, which is a rather good result.

It should also be emphasized that, in order to draw a fair conclusion from various error checks of a program, one may need to use double precision in compiling the code. In our experiments, we have seen examples from parallel runs performed on a SGI R4600 workstation (with no double precision option, 32 bits), where the results from the adjoint

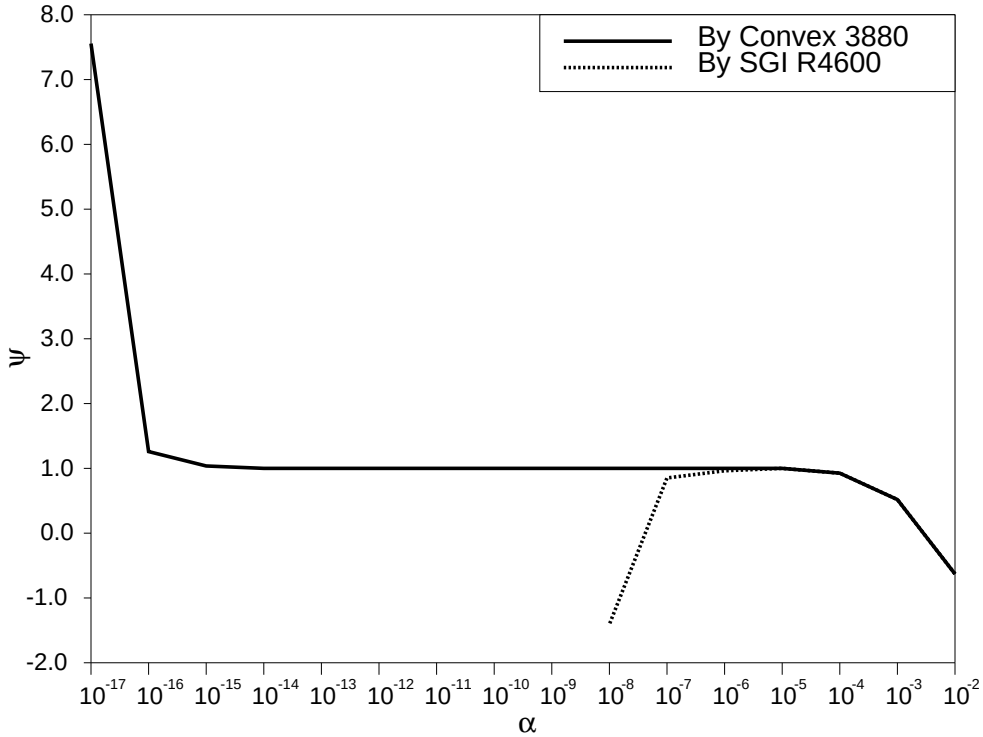

Figure 3: The cost function gradient check for control run, with α value *vs* Ψ.

check seem to be alarmingly poor under some circumstances. However, rerunning those tests with double precision on the Convex proves that the differences are due to the computer truncation error. Calculation with double precision is equally essential in the gradient check as performed above. The dotted line in Figure 3 shows the parallel gradient check result from the SGI R4600, which is rather different from the one obtained on the Convex. In this case, Ψ approaches unity only for a much narrower range of α values, between 10^{-7} and 10^{-3}. Tests with an α which is smaller than 10^{-8} on the SGI are impossible because of the computer truncation.

4.3 Variational assimilation experiments

We now look at the numerical results from the variational data assimilation experiments. As a contrast to the NL integration such as shown in Subsection 4.1, where the initial model state is known, in an assimilation problem, the exact initial model state is unknown, and it is the goal of the assimilation process to retrieve the initial model state as accurate as possible. Typically, there is only a guess for the initial state, plus some observations at certain points in time. Our task now is, through iterations of assimilation

cycles, to find a more suitable initial model state, so that the model can simulate the evolution process as close as possible to the true trajectories.

Assume that we have an initial guess of the model state as

```
w(1) = 1.10 ,  w(3) =   3.30 ,   w(3) =  5.50,
```

which is equivalent to the 10% increased model state amplitudes as specified for case 1. Now assume that we have three model state observations available during the assimilation period; Assume further that those observations have a randomly distributed error of maximum 10%. Running the Lorenz VDA system for 100 iterative cycles, with a 200 time step assimilation window (which is two non-dimensional time units at $dt = 0.01$), and the parameter α in (6) taken as 0.0005, we obtain a series of solutions, each representing an evolution trajectory of the model state. The solutions from the cycle 1, 2, 3, 5, 10 and 30 are plotted in Figure 4 (a) - (f). As can be seen, the model is rather efficient in bringing the trajectories to match the observations and the "true" evolution trajectories. After 5 iteration cycles, the model trajectories are already fairly close to the truth. In (e), for cycle 10, and (f), for cycle 30, the model results are seen to be almost indistinguishable from the true ones.

One may easily observe, from plots in Figure 4, that with the given initial guess of the model state, throughout the iteration cycles, the differences between the simulated trajectories and observations as well as "true trajectories" in the initial integration period are rather small. However, for the first few iteration cycles, [Figure 4 (a), (b) and (c)], the differences are increasingly significant at the later part of the integration period, reflecting the chaotic behavior of the system. To our satisfaction, with seemingly rather small improvements on the initial model state at each cycle, the assimilation iteration is able to bring the model trajectories close to the truth.

Figure 5 plots the cost function *vs* iteration cycle for control run. The dashed line with constant value in the figure is for the observation cost function, which represents the distance of the observation from true trajectories. It is obvious that the first 30 or so iteration cycles are most effective in reducing the cost function. From cycle 50, the curve of the cost function becomes rather flat, which means that additional cycles are no longer meaningful. In fact, from Figure 4 (f), it is obvious that, at cycle 30, the model solution

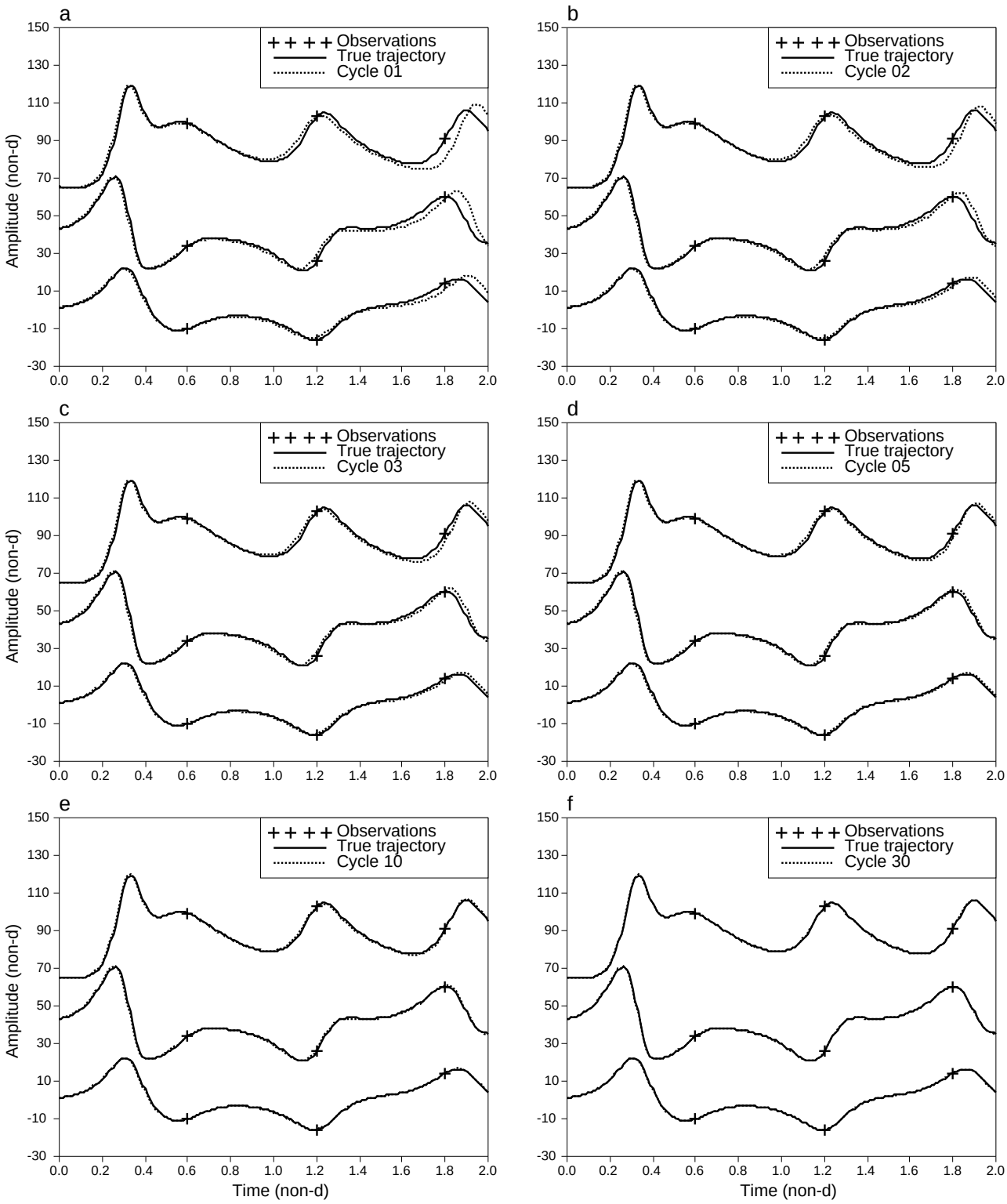

Figure 4: Variational data assimilation result *vs* true trajectories and observations for case 1. The initial guess is 10% off the true model state. Note that the $w(2)$ curves have been offset by +40, and $w(3)$ by +60. (a) Cycle 1; (b) Cycle 2; (c) Cycle 3; (d) Cycle 5; (e) Cycle 10; (f) Cycle 30.

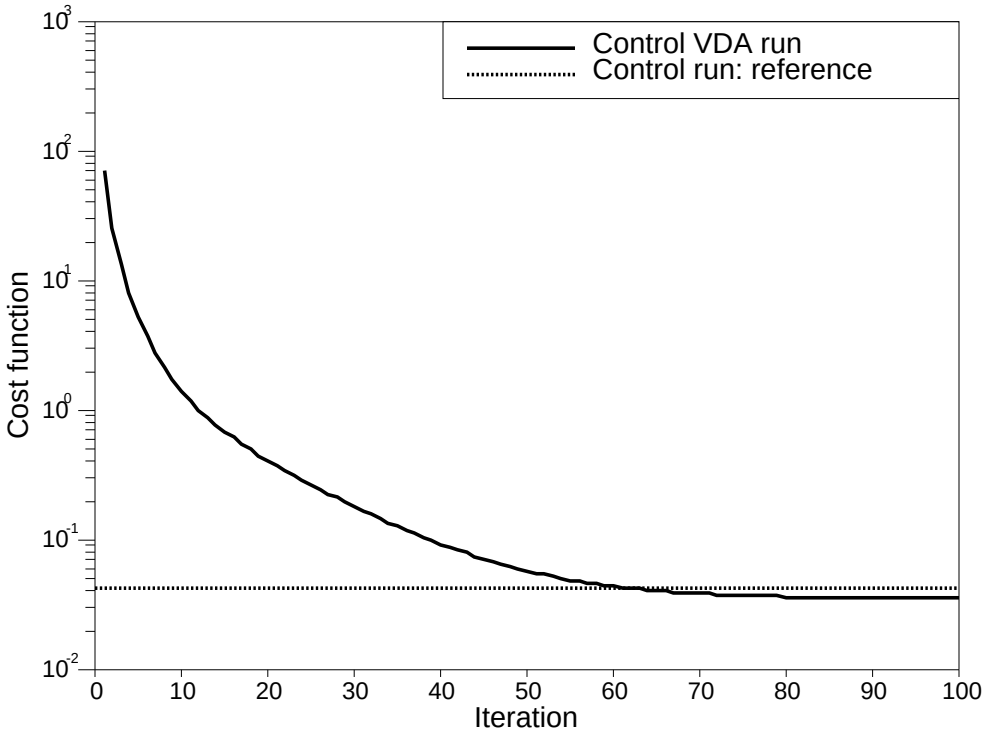

Figure 5: Cost function *vs* iteration cycle for the variational data assimilation run of case 1.

is already indistinguishable from the true trajectories.

Figure 6 and 7 present two sets of VDA runs to assimilate the same evolution trajectories as in case 1. In the case shown on the left column of Figure 6, the available observation is the same as in control run, but the initial guess is rather poor,

```
w(1) = 1.35 ,   w(2) = 4.05 ,   w(3) = 6.75,
```

which is a 35% increase in model state amplitudes compared to the true values specified for case 1. In the run for the right column of Figure 6, the initial guess field is even worse,

```
w(1) = 1.40 ,   w(2) = 4.20 ,   w(3) = 7.00,
```

which is a 40% error in initial model state amplitudes. Interestingly, for the case with 35% initial error, although the assimilated trajectories at the first few cycles differ significantly from the true trajectories, the following iteration cycles have been rather efficient. By cycle 10, the model results already converge nicely to the true trajectories. However, for

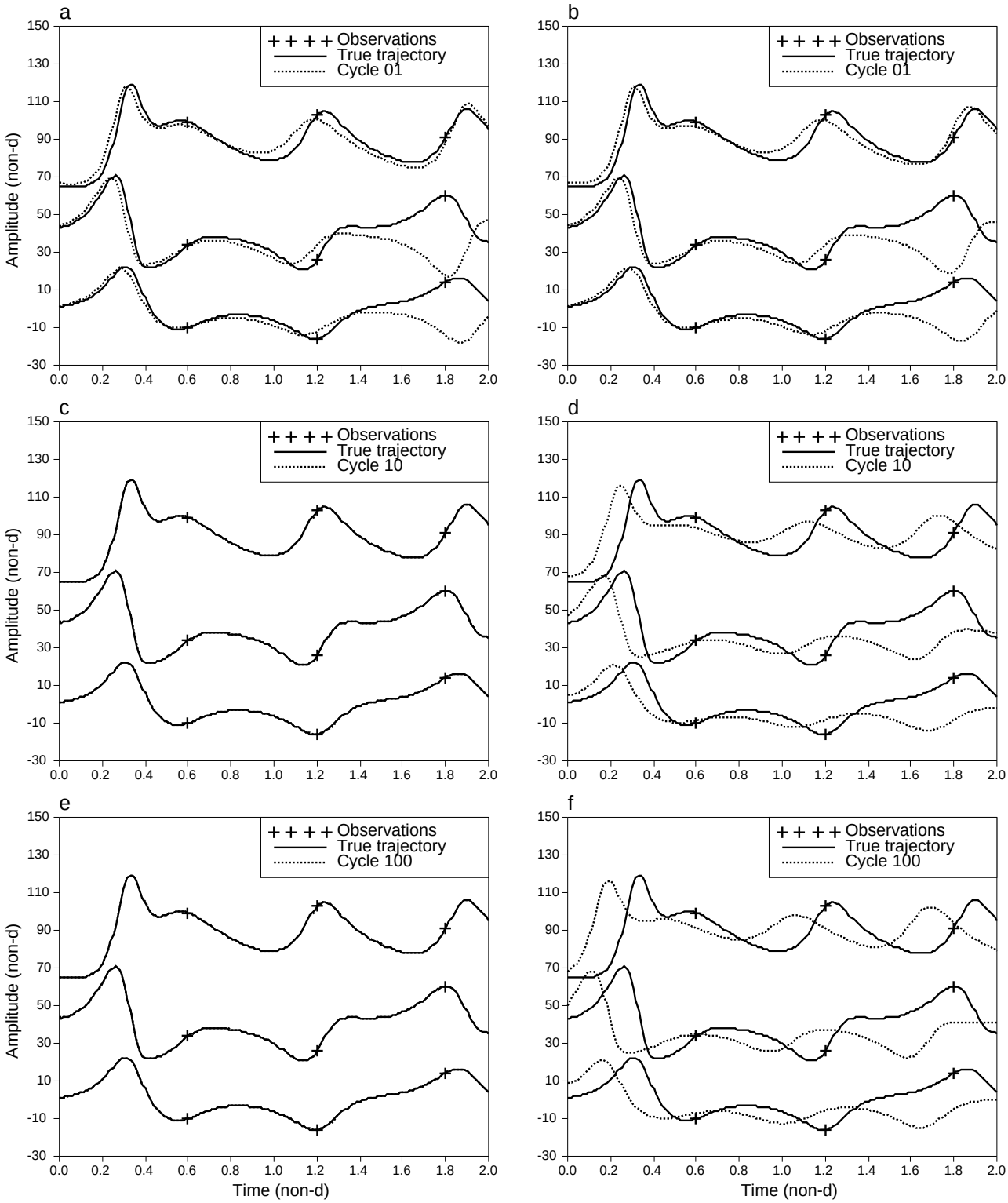

Figure 6: Same as in Figure 4, but in (a), (c) and (e), the initial guess is 35% off the true model state, and in (b), (d) and (f), the initial guess is 40% off the true values. (a) and (b) Cycle 1; (c) and (d) Cycle 10; (e) and (f) Cycle 100.

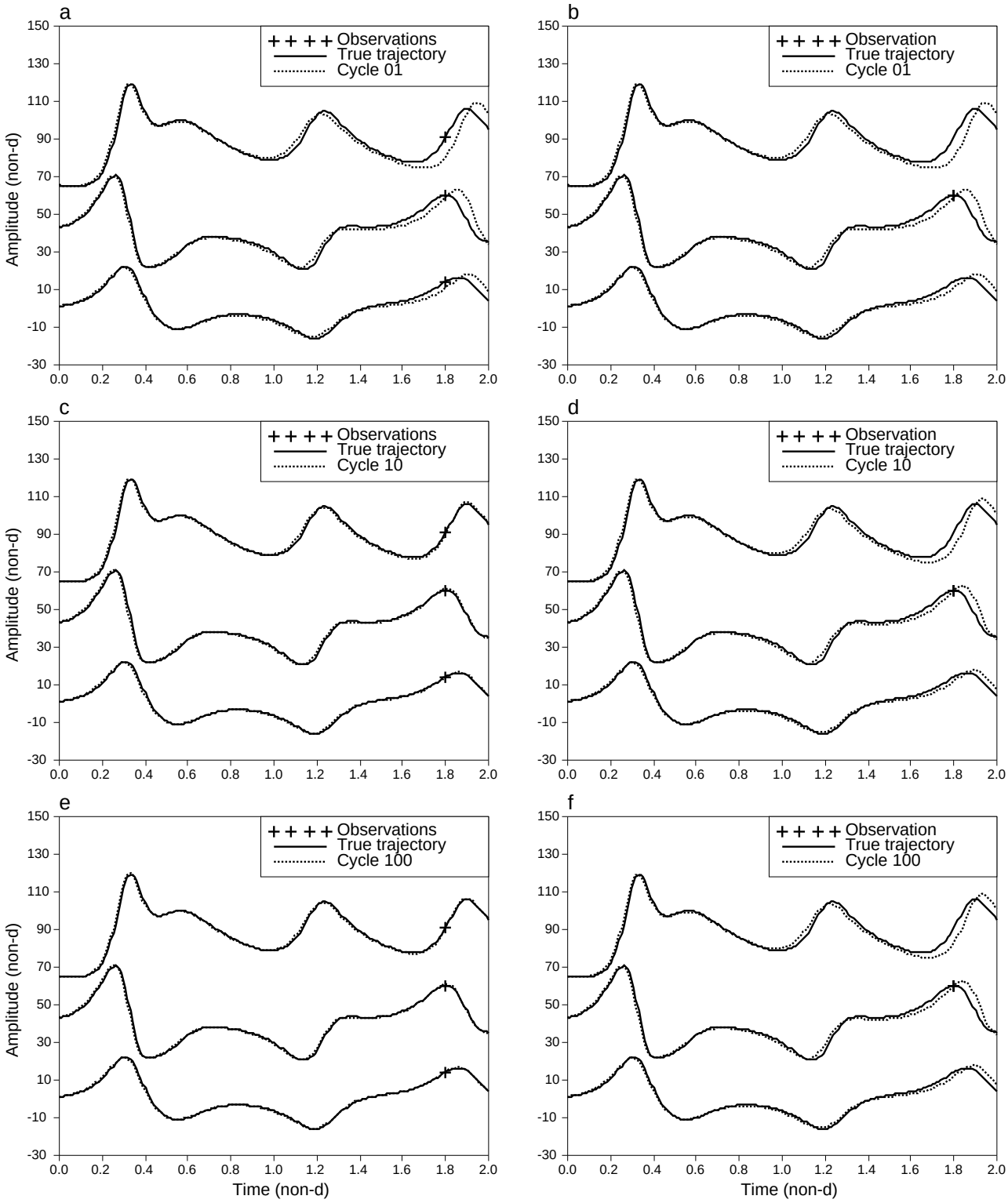

Figure 7: Same as in Figure 4, but in (a), (c) and (e), there is one observation at time step 180 for $\mathbf{w}$, and in (b), (d) and (f), there is only one observation at time step 180 for $w(2)$, and no observation for $w(1)$ or $w(3)$. (a) and (b) Cycle 1; (c) and (d) Cycle 10; (e) and (f) Cycle 100.

the case with 40% error, although the initial guess field is only 'slightly' worse than the previous case, the iteration does not converge.

The results shown in Figure 6 are not surprising. In fact, due to the chaotic nature of the Lorenz model solutions at the high Rayleigh number, the convergence of the minimization process with the poor initial guess as bad as 35% error would be impossible, had it not been for a relatively large number of observations. In each of the above runs, we specified 3 sets of observations for all components of $\mathbf{w}$. Considering the fact that the assimilation model has only three degrees of freedom, *i.e.*, there are only 3 components in the model state vector, $w(1)$, $w(2)$ and $w(3)$, the amount of available observations in the above experiments is rather "luxurious". Indeed, in a realistic assimilation application, it is seldom that an assimilation problem is over-determined by available observations. As a demonstration of this problem, Figure 7 shows two more runs in which, throughout the integration period, each has only one observation available at 180th time step, with the remaining setup the same as in control run. For the run shown by the left column of Figure 7, the observation covers all the components of the model vector $\mathbf{w}$. For the right column run of Figure 7, there is only one observation for $w(2)$. Obviously, in the latter case, the observation has less degrees of freedom compared to the assimilation model itself. For the former case, the assimilation converges successfully after a few iteration cycles. However, for the latter case, at 100 cycles, the VDA assimilation is still unable to obtain the desired trajectories.

4.4 Minimization methods

In the results obtained so far, we have used the steepest descent method for the minimization of the cost function. With this method, one only needs only to specify a constant parameter α, and the remaining algorithm of the minimization process is simple, as (6) shows. However, the efficiency of this method in bringing the iterative process to convergence is often questionable. Figure 8 shows schematically the algorithm used by the steepest descent approach. To obtain the desirable solution of model state $\mathbf{w}_0^{f,\infty}$, at each step, the steepest descent method advects the model state vector $\mathbf{w}_0^{f,k-1}$ toward the next state $\mathbf{w}_0^{f,k}$ along the direction of the cost function gradient, which is perpendicular to the tangent line of the cost function isolines, as shown in Figure 8 by dashed and dotted

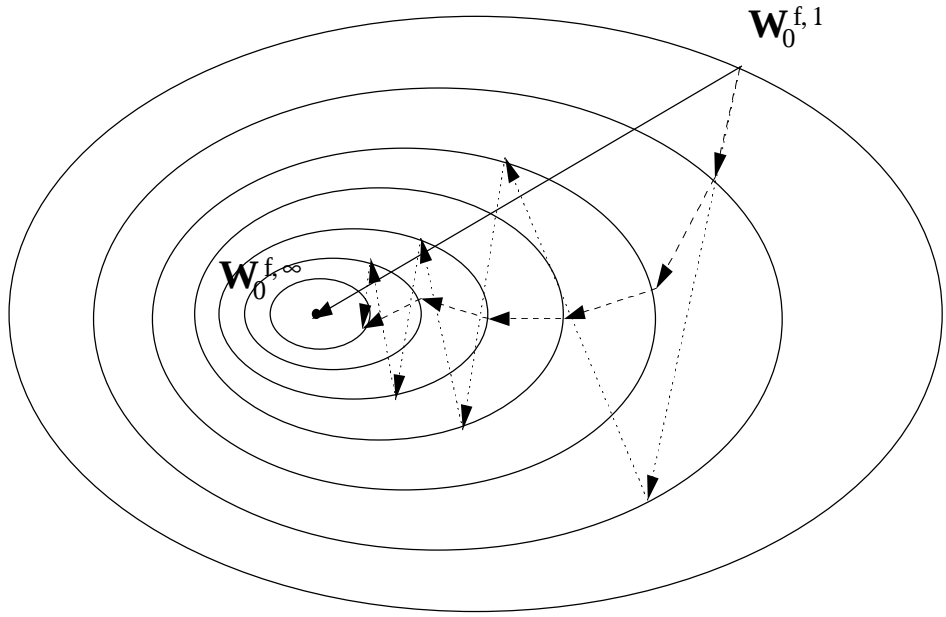

Figure 8: The steepest descent principle as a minimization algorithm in the variational data assimilation. The isolines are for the cost function, the solid line with arrow shows the optimal descent direction, the dashed lines shows typical track with the steepest descent method at a relatively small α value, dotted line at a larger α value.

lines. The efficiency of the algorithm depends on the magnitude of the constant α as well as the form of the cost function. A larger α value results in bigger individual descent steps. However, it does not necessarily bring about a more efficient minimization, nor does it guarantee ultimate convergence. As a demonstration of this, Figure 9 shows the cost functions *vs* iteration cycle for the VDA control run. For an α value of 0.001, the cost function reduction is fastest. A rather good convergence is achieved at iteration cycle 30. However, if the α value is taken as 10^{-5} or 10^{-6}, the iteration process become extremely slow. On the other hand, when α is as large as 0.01, convergence does not occur.

Obviously, with the steepest decent algorithm, it is important to choose suitable α values. Normally an optimal α value may only be obtained after many repeated test runs. Furthermore, this approach, even with a relatively "optimal" α value, is not necessarily the most efficient to achieve convergence. This can clearly be seen from Figure 8, where the solid line points to the most efficient minimization, and such a descent direction is not necessarily represented by the cost function gradient, as the 'steepest descent' algorithm

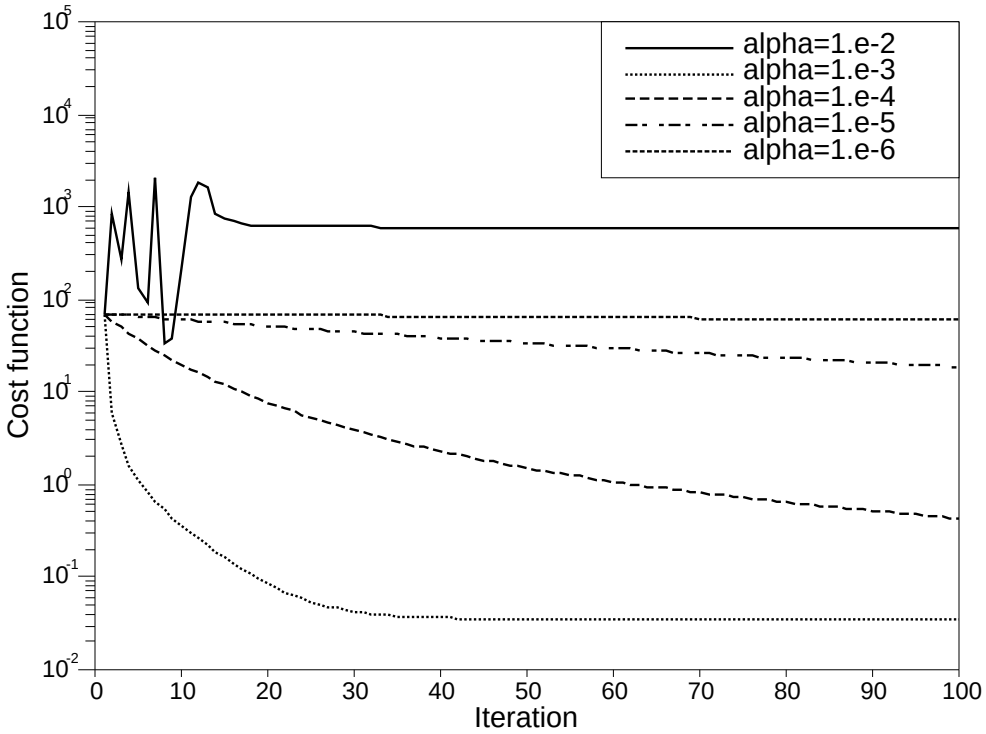

Figure 9: Same as in Figure 5, but with different α values. The experiment runs are performed on a SGI R4600 workstation

implies. Fortunately, more advanced minimization techniques are available (see, *e.g.*, Navon and Legler, 1987). As a demonstration of the importance of choosing a suitable minimization algorithm, we make another VDA run for case 3 as discussed in Subsection 4.1, which is basically the same as in control VDA run, but with a Rayleigh number of 10. In this case, both the initial guess and the three observations are of extremely bad quality. The integration length is set to be 100 time steps. Figure 10 (a) shows the VDA result for this case, by applying a sophisticated minimization software M1QN3, which uses a variable-storage quasi-Newton method (Gilbert *et al*, 1989). Despite the unusually poor initial guess and bad quality observations, a rather good assimilation result is obtained after 20 assimilation cycles. On the other hand, when a corresponding VDA run is performed with the steepest descent method, using an "optimal" α value of 0.01, which is chosen based on many test runs, the solution at 100 cycles is still a very poor fit to the true trajectories [Figure 10 (b)]. Furthermore, the iteration process has been very inefficient. After the 50'th cycle, there was scarcely any noticeable improvement in the cost function reduction (Not shown here). The minimization technique is another

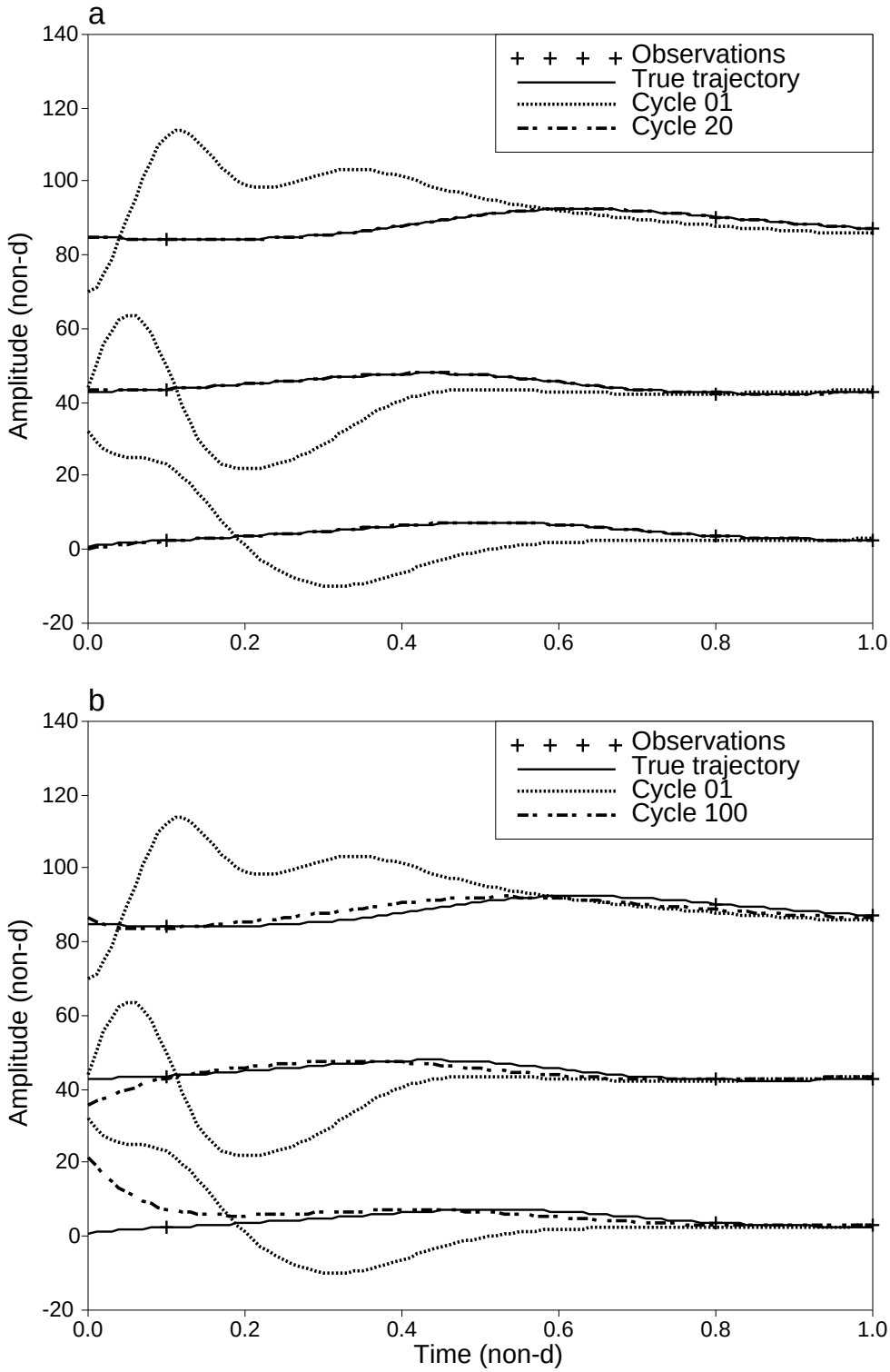

Figure 10: (a) Assimilation result with M1QN3 minimization method at Cycle 20 *vs* true trajectories and observations for case 3, with bad initial guess, and three observations. Note that the $w(2)$ curves have been offset by +40, and $w(3)$ by +80. (b) Assimilation result with steepest descent method at Cycle 100.

complicated issue in VDA applications, but we will not discuss the matter further here.

5. SUMMARY

In this note, we introduce basic aspects of variational data assimilation using the adjoint technique. A VDA system for the Lorenz equations is constructed. The Lorenz model is chosen because of its simplicity in structure, yet richness in solution patterns and many dynamical similarities with atmospheric systems. Detailed derivations and discussions are presented to introduce the basic concepts of the variational method, such as the cost function gradient, tangent linear and adjoint operators, *etc.* We then demonstrate, step by step, the procedure to code the Lorenz VDA system, and introduce the line-by-line approach for adjoint coding. With the constructed VDA system for the Lorenz model, many numerical experiments are performed. Following the procedures discussed in this note, we also perform various coding error checks to assure that the VDA system has been coded correctly.

We feel that by designing, coding, and running the Lorenz VDA system, many useful insights and experiences can be obtained which will be valuable for the future development work of a more complicated variational data assimilation system, such as the 4DVAR for HIRLAM. Of course, an actual weather forecast model like HIRLAM is a complex model in which processes much more sophisticated than the Lorenz model are involved, hence there is a much wider variety of problems in adjoint coding, many of which are untouched in this note.

Acknowledgments

We thank Nils Gustafsson, Erland Källén and Leif Laursen for carefully reading the manuscript and offering useful comments. Hans Hvas, Karsten Bolding Kristensen and Lars Peter Riishøjgaard are acknowledged for discussions on variational assimilation problems. The M1QN3 minimization package for the Lorenz VDA system, as used in subsection 4.4, was originally implemented by Lars Peter Riishøjgaard. We also thank Leif Laursen for encouraging the writing of this note.

Appendix A: An alternative derivation of the cost function gradient

Consider a scalar product $\langle x, y \rangle$. The first order variation in the cost function with respect to a perturbation $\delta \mathbf{w}^f$, δJ, may be expressed by the gradient of the cost function with respect to initial conditions $\mathbf{w}_0^f$, or with respect to the forecast $\mathbf{w}_k^f$ at the observation time $t = t_k$, respectively:

$$\delta J = \left\langle \nabla J(\mathbf{w}_0^f), \delta \mathbf{w}_0^f \right\rangle$$

or

$$\delta J = \left\langle \nabla J(\mathbf{w}_k^f), \delta \mathbf{w}_k^f \right\rangle = \left\langle \nabla J(\mathbf{w}_k^f), \mathbf{L}_k \delta \mathbf{w}_0^f \right\rangle = \left\langle \mathbf{L}_k^T \nabla J(\mathbf{w}_k^f), \delta \mathbf{w}_0^f \right\rangle$$

In the second expression, we have used the tangent linear model (11), $\delta \mathbf{w}_0^f = \mathbf{L}_k \delta \mathbf{w}_0^f$, and a property of inner product , $\langle x, \mathbf{L}_k y \rangle = \left\langle \mathbf{L}_k^T x, y \right\rangle$.

Identification between the two expressions leads to the equation for calculation of the gradient of the cost function with respect to the initial conditions:

$$\nabla J(\mathbf{w}_0^f) = \mathbf{L}_k^T \nabla J(\mathbf{w}_k^f)$$

Using the definition of J in (5) and including more observation times, we obtain the same equation as (14):

$$\nabla J(\mathbf{w}_0^f) = \sum_k \mathbf{L}_k^T (\mathbf{w}_k^f - \mathbf{w}_k^o).$$

Appendix B: FORTRAN code for the Lorenz VDA system

It is our belief that, by working with a relatively simple variational data assimilation system such as the one for the Lorenz model, researchers may gain many useful insights and experiences in practical VDA work. For that consideration, we provide here the FORTRAN source code of the Lorenz VDA system that we have developed, to help interested readers to familiarize with the program details. In order to limit the length of this note, the code given below has been reduced to a "bare-bone" version. Should a reader needs to use the code to make practice runs on his own, the authors would be happy to provide a more extensive and "user-friendly" code version.

```
C     +-------------------------------------------------------------------------------------------+
C     +                          VARIATIONAL DATA ASSIMILATION WITH THE LORENZ MODEL                +
C     +                                                                                             +
C     +                  DEVELOPED BY: XIANG-YU HUANG, DMI, XYH@DMI.MIN.DK, SPRING 1995             +
C     +                  REVISED BY:   XIAOHUA YANG,   DMI, XHY@DMI.MIN.DK, SPRING 1996             +
C     +                                                                                             +
C     +-------------------------------------------------------------------------------------------+
C     +             TASK LIST :  1. GENERATE A TRUE EVOLUTION TRAJECTORY                            +
C     +                          2. TEST THE TANGENT LINEAR MODEL AND ITS ADJOINT                   +
C     +                          3. SIMULATE OBSERVATIONS                                           +
C     +                          4. TEST THE GRADIENT OF THE COST FUNCTION                          +
C     +                          5. VDA WITH STEEPEST DESCENT MINIMIZATION                          +
C     +                          6. ARCHIVE DATA                                                    +
C     +                          7. QUIT                                                            +
C     +-------------------------------------------------------------------------------------------+

      program lorenzvda
      real      fw(3),tl(3),ad(3),tmp(3)
      logical   ltest
      data p,r,b,dt,nstop/10.,28.,2.66666667,0.01,300/

1     read (5,*) itask

      if (itask.eq.1) then                                                  ! Generate a true solution
         ncyc=0
         call in_out('in','i',01,ncyc,0,fw)
         call fom(ncyc,fw)
      endif

      if (itask.eq.2) then                                                  ! Test the tangent linear
         ncyc=1                                                             ! model TLM
         call in_out('in','i',01,ncyc-1,0,tmp)
         call in_out('in','p',01,ncyc-1,0,tl)
         call hplus (tmp,tl,fw)
         call fom(ncyc-1,tmp)
         call fom(ncyc,fw)
         call hminus(fw,tmp,fw)
         write(6,*) 'N (w+ dw) - N (w), from NL ',(fw(i),i=1,3)
         call tlm(ncyc,tl)
         write(6,*) '      L(dw), from TL       ',(tl(i),i=1,3)
         ltest=.true.                                                       ! Test the adjoint model ADM
         call hinnerp(tl,tl,tt)
         call hcopy(tl,ad)
         call adm(ncyc,ad,ltest)
         call in_out('in','p',01,ncyc-1,0,tl)
         call hinnerp(tl,ad,pa)
         write(6,*) '    LX_t LX    = ',tt
         write(6,*) 'X_t (L_t (LX)) = ',pa
      endif

      if (itask.eq.3) call obsloz(01,nstop)                                 ! Construct 'observations'

      if (itask.eq.4) then                                                  ! Test the gradient of the
         ltest = .false.                                                    ! cost fuction
         call obsloz(01,nstop)
         ncyc = 1
         call in_out('in','i',01,ncyc,0,fw)
         call fom(ncyc,fw)
         cost0 = ocost(01,ncyc,nstop)
         ncyc = 2
         call hsetcon(ad,0.)
         call adm(ncyc,ad,ltest)
```

```
         call hinnerp(ad,ad,aa)
         read (5,*) na1,na2                                          ! read in alpha range in form
         do nalpha = na1,na2                                         ! of 10**alpha
            alpha = 10.**(-nalpha)
            call in_out('in','i',01,ncyc-1,0,fw)
            call hmini(fw,ad,alpha)
            call fom(ncyc,fw)
            cost1 = ocost(01,ncyc,nstop)
            chkgra = -(cost1-cost0)/(alpha*aa)
            write(6,*) 'chkgra = ',chkgra
            call in_out('out','g',01,0,nalpha,chkgra,1)
         enddo
      endif

      if (itask.eq.5) then                                           ! Run the VDA with the
         ltest = .false.                                             ! steepest descent minimization
         call obsloz(01,nstop)
2        write(6,*) 'for iterations from istart to and istop:'
         read (5,*) istart,istop,alpha                               ! read the start and end of
         if (istart.eq.0) goto 1                                     ! the iteration cycle and
         do ncyc = istart,istop                                      ! alpha value
            if (ncyc.eq.1) then
               call in_out('in','i',01,ncyc,0,fw)
            else
               call hsetcon(ad,0.)
               call adm(ncyc,ad,ltest)
               call in_out('in','m',01,ncyc-1,0,fw)
               call hmini(fw,ad,alpha)
            endif
            call fom(ncyc,fw)
            cost = ocost(01,ncyc,nstop)
         enddo
         goto 2
      endif

      if (itask.eq.6) call arcdat(01,ncyc,nstop)                   ! Archive experiment output

      if (itask.ge.7) stop                                         ! Stop the program
      goto 1                                                       ! Restart
      end
C-----------------------------------------------------------------------------------------------------------
C                                subroutines related to forward model (fom)
C-----------------------------------------------------------------------------------------------------------
      subroutine florenz(x,y,p,r,b)                                ! subroutine :
      dimension x(3),y(3)                                          ! the descretized Lorenz
      y(1) = -p*x(1) + p*x(2)                                      ! equations
      y(2) = x(1)*(r-x(3)) -x(2)                                   !     ***********
      y(3) = x(1)*x(2) - b*x(3)                                    !     * FLORENZ *
      return                                                       !     ***********
      end
C-----------------------------------------------------------------------------------------------------------
      subroutine fom(ncyc,x)                                       ! subroutine :
      dimension x(3),y(3)                                          ! the forward model
      data p,r,b,dt,nstop/10.,10.,2.66666667,0.01,300/
      call in_out('out','m',nexp,ncyc,0,x)
      do ntim = 1,nstop
         call florenz(x,y,p,r,b)                                   ! compute tendencies
         call fstep(x,y,dt)                                        ! time stepping
         call in_out('out','m',nexp,ncyc,ntim,x)                   ! store data
      enddo
      return
      end
C-----------------------------------------------------------------------------------------------------------
      subroutine fstep(x,y,dt)                                     ! subroutine :
      dimension x(3),y(3)                                          ! forward time stepping scheme
      do i = 1,3
         x(i) = x(i) + dt*y(i)
      enddo
      return
      end
C-----------------------------------------------------------------------------------------------------------
C                                subroutines related to tangent linear model (tlm)
C-----------------------------------------------------------------------------------------------------------
      subroutine tlm(ncyc,tl)                                      ! subroutine :
      dimension tl(3),xx(6),yy(6),tb(3),y(3)                       ! the tangent linear model
      data p,r,b,dt,nstop/10.,10.,2.66666667,0.01,300/
      do ntim = 1,nstop
         call in_out('in','m',nexp,ncyc-1,ntim-1,tb)               ! get basic state
         call tlorenz(tl,y,tb,p,r,b)                               ! compute tendencies
         call fstep(tl,y,dt)                                       ! time stepping
      enddo
      return
      end
C-----------------------------------------------------------------------------------------------------------
      subroutine tlorenz(x,y,xb,p,r,b)                             ! subroutine :
      dimension x(3),y(3),xb(3)                                    ! the tangent linear Lorenz
      y(1) =        -p*x(1)     + p*x(2)                           ! equations
      y(2) = (r-xb(3))*x(1)       - x(2) - xb(1)*x(3)              !     ***********
      y(3) =     xb(2)*x(1) + xb(1)*x(2)     - b*x(3)              !     * TLORENZ *
      return                                                       !     ***********
```

```
      end
C-----------------------------------------------------------------------------------------------------------
C                                subroutines related to adjoint model (adm)
C-----------------------------------------------------------------------------------------------------------
      subroutine adm(ncyc,a,ltest)                                                  ! subroutine :
      dimension a(3),tl(3),ab(3),d(3),y(3),tb(3)                                    ! the adjoint model
      logical ltest
      data p,r,b,dt,nstop/10.,10.,2.66666667,0.01,300/
      call hsetcon(y,0.)
      do ntim = nstop,1,-1
         call in_out('in','m',nexp,ncyc-1,ntim-1,ab)                                ! get basic state
         if (.not.ltest) then
            call obminuso(d,ntim,nexp,ncyc-1)                                       ! compute the xb-xo
            call hplus(a,d,a)                                                       ! and add to the ad variables
         endif
         call astep(a,y,dt)                                                         ! time steeping adjoint
         call alorenz(a,y,ab,p,r,b)                                                 ! tendency adjoints
      enddo
      return
      end
C-----------------------------------------------------------------------------------------------------------
      subroutine alorenz(a,y,xb,p,r,b)                                              ! subroutine :
      dimension a(3),xb(3),y(3)                                                     ! the adjoint Lorenz equations
      a(1) = a(1) -p*y(1) + (r-xb(3))*y(2) + xb(2)*y(3)                             !     ***********
      a(2) = a(2) +p*y(1)           - y(2) + xb(1)*y(3)                             !     * ALORENZ *
      a(3) = a(3)             - xb(1)*y(2)     - b*y(3)                             !     ***********
      y(1) = 0.
      y(2) = 0.
      y(3) = 0.
      return
      end
C-----------------------------------------------------------------------------------------------------------
      subroutine astep(x,y,dt)                                                      ! subroutine :
      dimension x(3),y(3)                                                           ! forward time stepping adjoint
      do i = 1,3
         y(i) = y(i)+dt*x(i)
      enddo
      return
      end
C-----------------------------------------------------------------------------------------------------------
C                                subroutines related to observations
C-----------------------------------------------------------------------------------------------------------
      subroutine obminuso(xj,ntim,nexp,ncyc)                                        ! subroutine :
      dimension xj(3),x(3),o(3)                                                     ! compute difference between
      call in_out('in','m',nexp,ncyc,ntim,x)                                        ! model state and observations
      call in_out('in','o',nexp,0,ntim,o)
      do i=1,3
         if (x(i).eq.999.999.or.o(i).eq.999.999) then
            xj(i) = 0.
         else
            xj(i) = x(i) - o(i)
         endif
      enddo
      return
      end
C-----------------------------------------------------------------------------------------------------------
      subroutine obsloz(nexp,nstop)                                                 ! subroutine :
      dimension o(3),e(3),zwrk(3)                                                   ! simulate observations
      call in_out('in','e',nexp,0,0,e)
      call in_out('in','i',nexp,0,0,zwrk)
      call fom(0,zwrk)
      do ntim = 0,nstop
         call in_out('in','m',nexp,0,ntim,o,3)
         if (mod(ntim,10).eq.0) then
            do i = 1,3
               o(i) = o(i) + e(i)*2.*(ran(0)-0.5)
            enddo
         else
            call hsetcon(o,3,999.999)
         endif
         call in_out('out','o',nexp,0,ntim,o)
      enddo
      cost = ocost(nexp,0,nstop)
      return
      end
C-----------------------------------------------------------------------------------------------------------
      function ocost(nexp,ncyc,nstop)                                               ! function :
      dimension xj(3)                                                               ! compute cost function
      ocost = 0.
      do n = 0,nstop
         call obminuso(xj,n,nexp,ncyc)
         do i = 1,3
            ocost = ocost + xj(i)**2
         enddo
      enddo
      ocost = ocost/2.
      call in_out('out','j',nexp,ncyc,n0,ocost)
      return
      end
```

```
C-----------------------------------------------------------------------------------------------------------
C                                subroutines for general uses
C-----------------------------------------------------------------------------------------------------------
      subroutine hcopy(x,y)                                                          ! subroutine :
      dimension x(3),y(3)                                                            ! copy a matrix
      do i=1,3
         y(i) = x(i)
      enddo
      return
      end
C-----------------------------------------------------------------------------------------------------------
      subroutine hinnerp(a,b,c)                                                      ! subroutine :
      dimension a(3),b(3),c                                                          ! inner product
      c = 0.
      do k=1,3
          c = c + a(k)*b(k)
      enddo
      return
      end
C-----------------------------------------------------------------------------------------------------------
      subroutine hmini(x,a,alpha)                                                    ! subroutine :
      dimension x(3),a(3)                                                            ! minimization with the
      do i=1,3                                                                       ! steepest descent method
         x(i) = x(i) - alpha*a(i)
      enddo
      return
      end
C-----------------------------------------------------------------------------------------------------------
      subroutine hminus(x,y,z)                                                       ! subroutine :
      dimension x(3),y(3),z(3)                                                       ! matrix subtraction operation
      do i=1,3
         z(i) = x(i) - y(i)
      enddo
      return
      end
C-----------------------------------------------------------------------------------------------------------
      subroutine hplus(x,y,z)                                                        ! subroutine :
      dimension x(3),y(3),z(3)                                                       ! matrix addition operation
      do i=1,3
         z(i) = x(i) + y(i)
      enddo
      return
      end
C-----------------------------------------------------------------------------------------------------------
      subroutine hsetcon(a,c)                                                        ! subroutine :
      dimension a(3)                                                                 ! assign a value to a matrix
      do i=1,3
         a(i) = c
      enddo
      return
      end
C-----------------------------------------------------------------------------------------------------------
C                                subroutines related to data i/o
C-----------------------------------------------------------------------------------------------------------
      subroutine arcdat(nexp,ncyc,ntim)
C     archive the calculation result. (ommited)
C-----------------------------------------------------------------------------------------------------------
      subroutine in_out(cgp,ctyp,nexp,ncyc,ntim,x)
C     fetch/store data in the active memory space. (ommited)
C===========================================================================================================  END
```

REFERENCES

Courtier, C., J. Derber, R. Errico, J. F. Louis and T. Vukicevic, 1993: Important literature on the use of adjoint, variational methods and the Kalman filter in meteorology. *Tellus*, **45A**, 342-357.

Daley, R. 1991: *Atmospheric data analysis.* Cambridge University Press, pp 457.

Gilbert, J. Ch., C. Lemarethal, 1989: Some numerical experiments with variable storage quasi-Newton algorithms. *Mathematical Programming*, **45**, 407-435.

Gauthier, P. 1992: Chaos and quadri-dimensional data assimilation: a study based on the Lorenz model. *Tellus*, **44A**, 2-17.

Ghil, M. and P. Malanotte-Rizzoli, 1991: Data assimilation in meteorology and oceanography. *Advances in Geophysics*, **33**, Academic press, 141-266.

Li, Y., I. M. Navon, S. Moorthi and R. W. Higgins, 1991: The 2-D semi- implicit semi-Lagrangian global model: direct solver, vectorization and adjoint model development. *Technical Report FSU-SCRI-91-158*, Super Computations Research Institute, Florida State University, pp 57.

Lorenz, E.N., 1963: Deterministic nonperiodic flow. *J. Atmos. Sci.*, **20**, 130-141.

Navon, I. and D. M. Legler, 1987: Conjugate-gradient methods for large-scale minimization in meteorology. *Mon. Wea. Rev.* , **115**, 1479 - 1502.

Navon, I., X. Zou, J. Derber and J. Sela, 1992: Variational data assimilation with an adiabatic version of the NMC spectral model. *Mon. Wea. Rev.* , **120**, 1433 - 1446.

Rostaing, N., S. Dalmas and A. Galligo, 1993: Automatic differentiation in Odyssee. *Tellus*, **45A**, 558-568.

Pires, C., R. Vautard and O. Talagrand, 1996: On extending the limits of variational assimilation in nonlinear chaotic systems. *Tellus*, **48A**, 96-121.

Stensrud, D.J. and J.-W. Bao, 1992: Behaviors of variational and nudging assimilation techniques with a chaotic low-order model. *Mon. Wea. Rev.*, **120**, 3016-3028.