AUTOMATIC DISCRETE EMPIRICAL INTERPOLATION FOR NONLINEAR MODEL REDUCTION *

RUSSELL L. CARDEN AND DANNY C. SORENSEN †

Abstract.
The Discrete Empirical Interpolation Method (DEIM) is a technique for model reduction of nonlinear dynamical systems. It is based upon a modification to proper orthogonal decomposition which is designed to reduce the computational complexity for evaluating the reduced order nonlinear term. The DEIM approach is based upon an interpolatory projection and only requires evaluation of a few selected components of the original nonlinear term. Thus, implementation of the reduced order nonlinear term requires a new code to be derived from the original code for evaluating the nonlinearity. This work describes a methodology for automatically deriving a code for the reduced order nonlinearity directly from the original nonlinear code. The methodology is derived from standard techniques of automatic differentiation. This capability removes the possibly tedious and error prone task of producing such a code by hand and hence can facilitate the use of DEIM by non-experts.

1. Introduction. The Discrete Empirical Interpolation Method (DEIM) is a technique for model reduction of nonlinear dynamical systems. The principal goal of model reduction is to replace the given high dimensional dynamical system with a very low dimensional approximation that retains nearly the same response characteristics. One situation that has been successfully exploited through Proper Orthogonal Decomposition [4, 9, 11, 12, 13] occurs when solutions to the dynamical system approximately lie in a low dimensional linear subspace. In many cases, restricting the dynamical system to this subspace in a Galerkin sense can significantly reduce the amount of computation required for simulation. Once the reduced order model (ROM) has been constructed, such savings allow for efficient calculation of many instances of such a dynamical system over varying parameter settings or initial conditions. This can facilitate such things as Monte Carlo simulation, optimization of parametrized systems, neural systems simulation and others that would otherwise be computationally intractable [1, 7].

Unfortunately, while a straightforward implementation of the POD-Galerkin idea often yields a very low order reduced model, the computational complexity of this ROM is often the same as the complexity of the full system. The DEIM addresses this complexity issue by replacing the orthogonal projection of POD-Galerkin with an oblique projection. The DEIM oblique projector is based upon interpolation. The DEIM procedure constructs a *important* small selected subset of indices through a greedy algorithm and the nonlinear function only needs to be evaluated at this greatly reduced set of component functions. Usually, each of these components only depends upon a small subset of the state variables and herein lies the complexity reduction. Implementation of this scheme requires the generation of a code for the ROM that is derived from the original nonlinear function. This can be a time consuming and error prone process when done by hand, especially when the user is unfamiliar with the source code for the original nonlinearity.

To overcome this difficulty in implementation, we propose using techniques from algorithmic (or automatic) differentiation (AD). AD provides a methodology for de-

* This work was supported in part by AFOSR grant FA9550-09-1-0225 and by NSF grant CCF-1017401.

†DEPARTMENT OF COMPUTATIONAL AND APPLIED MATHEMATICS, MS-134, RICE UNIVERSITY, 6100 MAIN STREET, HOUSTON, TEXAS 77005-1892, USA. RUSSELL.L.CARDEN@RICE.EDU, SORENSEN@RICE.EDU

riving a code for computing derivatives, i.e. Jacobians, gradients, etc., directly from the source code for function evaluation. A number of the intermediate steps of AD turn out to provide exactly the information needed to automatically construct a code for evaluating the resulting ROM nonlinearity and also its Jacobian. Here, we describe the basic ideas of this process.

Consider the following dynamical system, $y \in \mathbb{R}^n$, $A \in \mathbb{R}^{n\times n}$, $f : \mathbb{R}^n \to \mathbb{R}^n$:

$$\frac{dy}{dt} = Ay + f(y). \tag{1.1}$$

We can think of A and f as the linear and nonlinear terms in the system. The proper orthogonal decomposition method computes a linear subspace for approximating the solution $y(t)$ by computing a singular value decomposition of snapshots of the trajectory: $V\Sigma W^T = [y(t_1), y(t_2), \ldots, y(t_s)]$, where s is the number of snapshots. The left singular vectors corresponding to the k largest singular values are then used to approximate the solution $y(t) \approx V_k\widehat{y}(t)$. Plugging the approximation for $y(t)$ into (1.1) yields

$$V_k \frac{d\widehat{y}}{dt} \approx AV_k\widehat{y} + f(V_k\widehat{y}). \tag{1.2}$$

This is an overdetermined dynamical system for $\widehat{y}$. A Galerkin condition applied to (1.2) requires that the residual of this approximation be orthogonal to the subspace $\mathrm{Ran}(V_k)$ from which we are approximating. This leads to the following reduced order dynamical system:

$$\frac{d\widehat{y}}{dt} = V_k^T AV_k\widehat{y} + V_k^T f(V_k\widehat{y}). \tag{1.3}$$

The reduced linear term $V_k^T AV_k$ is a $k \times k$ matrix that can be precomputed prior to solving the reduced dynamical system, thereby providing, depending on the size of k, a reduction in the amount of computation. However the nonlinear term $V_k^T f(V_k\widehat{y})$ requires promoting $\widehat{y}$ to a length n vector $V_k\widehat{y}$ and then evaluating f, hence computationally the reduced nonlinear term is just as expensive to evaluate as the original. This problem can be overcome by approximating the nonlinear term with interpolation.

2. Discrete Empirical Interpolation Method. The Discrete Empirical Interpolation Method (DEIM) reduces the nonlinear term using interpolation. The idea is that if the nonlinear function $f(y)$ for $y \in \mathrm{Ran}(V_k)$ is known to lie (approximately) in some linear subspace $\mathrm{Ran}(U_k)$ for $U_k \in \mathbb{R}^{n\times k}$, then $f(y)$ can be approximated if only a few of the entries of $f(y)$ are known. Suppose entries $p_1, p_2, \ldots, p_k$ of f are given, then the DEIM approximation of the nonlinear function f is as follows:

1. $P = [e_{p_1}\ e_{p_1} \ldots e_{p_k}]$,
2. To get $f \approx U_k c$,
3. Require Interpolation: $P^T f = P^T U_k c$,
4. Put $c = (P^T U_k)^{-1} P^T f$,
5. Then $U_k c = U_k(P^T U_k)^{-1} P^T f$.

In order for the DEIM approximation to make sense, $P^T U_k$ must be invertible and this is ensured by the DEIM greedy algorithm for selecting the p_i. If $f(y) \in \mathrm{Ran}(U_k)$, then $f(y) = U(P^T U)^{-1} P^T f(y)$ and if the basis U_k is orthogonal,

$$\|f(y) - U_k(P^T U_k)^{-1} P^T f(y)\| \le \|(P^T U_k)^{-1}\| \min_{g\in\mathrm{Ran}(U_k)} \|f(y) - g\|$$

(see [3] for a proof). Hence, if $f(y)$ is nearly in $\mathrm{Ran}(U_k)$ then it will be well approximated by it's interpolant. The DEIM approximation to the reduced nonlinear term in (1.3) is

$$V_k^T U_k (P^T U_k)^{-1} P^T f(V_k \widehat{y}).$$

The matrix $V_k^T U_k (P^T U_k)^{-1}$ is $k \times k$ and it can be precomputed. The vector $P^T f(V_k \widehat{y})$ is a length k vector that requires evaluating only k components of $f(V_k \widehat{y})$. Provided the k desired components of $f(V_k \widehat{y})$ can be computed efficiently, the DEIM approximation to the nonlinear term can provide a significant computational savings.

Determining a DEIM approximation of a vector valued function requires specifying the important components p_i and providing a way of efficiently evaluating the components of $f(y)$. The algorithm for determining the p_i may be found in [3]. Let us assume that along with our given nonlinear function f we are provided functions f_i for $i = 1, \ldots, n$ for evaluating each component of f. The *dependency list* for f_i is the list of components of y needed to compute $f_i(y)$. While there is always a complexity reduction from the DEIM approximation to f, it is clearly greatest when each component f_i depends only on a limited number of components of the state y. This dependency list can be determined from the sparsity pattern of the Jacobian of f. The key idea is that the nonzero entries the i-th row correspond to the subset of state variables required to evaluate f_i. Once the dependency list is known for all selected components, the formation of the full vector $V\widehat{y}$ for $P^T f(V\widehat{y})$ can be avoided, and each entry of $P^T f(V\widehat{y})$ can be computed by passing only the necessary entries of $V\widehat{y}$ to each of the f_i. AD offers a means of computing the dependency lists by analyzing the code for evaluating the f_i. Moreover, we shall show how AD techniques can be used to automatically generate code for evaluating the selected f_i directly from the code provided to evaluate f. Depending on the origin and nature of the code for f, the ability to generate code for evaluating the selected components of f can be quite useful. For example, when the code for f consists of legacy code, for which there exists little documentation and few remaining experts, it would probably be very difficult and time consuming to derive the reduced order code by hand. The code for f might be so complex that writing code for efficiently evaluating the selected f_i would be prohibitively tedious and error prone.

3. Automatic or Algorithmic Differentiation.

Algorithmic differentiation is a set of techniques for generating a procedure for evaluating the derivative of a function specified by an algorithm. AD is not symbolic differentiation; the end result is a code for computing derivatives of the original function rather than a symbolic formula for the derivative. Proponents of AD state that unlike difference methods for approximating derivatives, for AD there is no truncation error, and hence the computed derivatives are accurate to the order of working precision [6].

To differentiate an algorithm using AD, the algorithm must be specified in terms of basic operations such as

1. Assign independent/dependent variables: $\leftarrow, \rightarrow$.
2. unary operations: `sin,cos,tan,asin,atan,exp,log,sinh,cosh,` `tanh,ceil,floor,++,--,sqrt,abs`
3. Binary operation: `+,-,*,/,pow,min,erf,=,>,<,>=,<=,==`
4. Simple Conditional assignement: `result = a>b?x:y`

Applying AD then consists of repeated application of the chain rule. Note to be consistent with earlier DEIM papers, counter to AD notation, we have y rather than

x as the input variable, and f rather than y as the output variable.

3.1. Examples. As a simple first example, consider the following line of code

```
f= y_1*y_2+ y_1-exp(y_1*y_2)
```

where $y_1, y_2, f \in \mathbb{R}$. The following sequence of operations, with temporary variables v, reflects how a computer might compute f in terms of basic operations. *(Might want to cut down on reuse of intermediate variables.)*

```
v_1=y_1;
v_2=y_2;     Load input variables
-----------------
v_2=v_1*v_2;
v_3=exp(v_2);
v_3=v_1-v_3;
v_2=v_2+v_3;
----------------
f=v_2;       Store output variables
```

Example 1: Operations for a simple example.

Each line corresponds to one basic operation. The temporary variables v_2 and v_3 are used for storing intermediate results. First the inputs are loaded into the temporary variables. The output is computed and then loaded into the desired output variable.

There are two ways in which AD hast been implemented: source transformation and operator overloading. Source transformation makes use of compiler techniques to generate code for evaluating derivatives of the function directly from its source code. For the example above, source transformation would produce the following sequence of operations.

```
v_1=y_1;
vdot_1=ydot_1;
v_2=y_2;
vdot_2=ydot_2; Load input variables
-----------------
v_2=v_1*v_2;
vdot_2 = vdot_1*v_2+v_1*vdot_2;
v_3=exp(v_2);
vdot_3=exp(v_2)*vdot_2;
v_3=v_1-v_3;
vdot_3=vdot_1-vdot_3;
v_2=v_2+v_3;
vdot_2=vdot_2+vdot_3;
----------------
f=v_2;
fdot=vdot_2;    Store output variables
```

Example 2: Forward Mode via Source Transformation.

The dot variables correspond to the change in that particular variable at that point in the code due to the change in the input variables y_i.

Operator overloading incorporates the above operations for the dotted variables automatically by changing the type of the variables and then overloading all the basic operations for this new type. For example a `C double` variable would become an `adouble` which would have a field `val` for the value of the variable and a field `dot` for the change in the variable due to changes in the inputs. An overloaded version of `+` is shown below.

```
adouble operator*(adouble a,adouble b)
{
c = new adouble;
c.val =a.val*b.val;
c.dot=a.dot*b.val+a.val*b.dot;
return c;
}
```

Example 3: Operator overloaded version of `+`.

An advantage of operator overloading is that the only part of the code that the user has to change is the type of the variables; the function and all its operations remain the same.

In both operator overloading and source transformation, independent and dependent, as well as *active* and *passive* variables must be specified. Active variables are variables that change as a result of changes in the independent variables, all non-active variables are passive. In operator overloading, to facilitate the process of designating active variable, the assignment of an active variable to a passive variable is prevented by type checking. Generating efficient code for computing derivatives with standard automatic differentiation tools requires a good understanding of which variables should be active and which should be passive. To use AD to generate code for the f_i given code for f we are concerned with active variables as well as active lines of code, i.e., which operations are need for computing the desired output. The analysis that must be performed by the user or the AD tool to determine active variables in the case of operator overloading and active lines of code in the case of source transformation is known as *activity analysis* see [6, 10, 8].

In general, in AD an algorithm for computing some vector f as a function of the input vector y is represented as a sequence of functions $\phi_j,\ j = 1, \ldots, k$ applied to a sequence of vectors

$$v_j = \phi_j(v_{j-1}) \tag{3.1}$$

with $v_0 = y$ where y is the input vector and $f = v_k$ is the output vector. Each function ϕ_j must be representable in terms of basic operations or operations with known derivatives. Also, each operation ϕ_j must be differentiable in the neighborhood of its input v_j.

If we let $\dot{v}_i$ denote the change in v_i then applying the chain rule to (3.1) yields

$$\dot{v}_j = \phi'_j(v_{j-1})[\dot{v}_{j-1}], \tag{3.2}$$

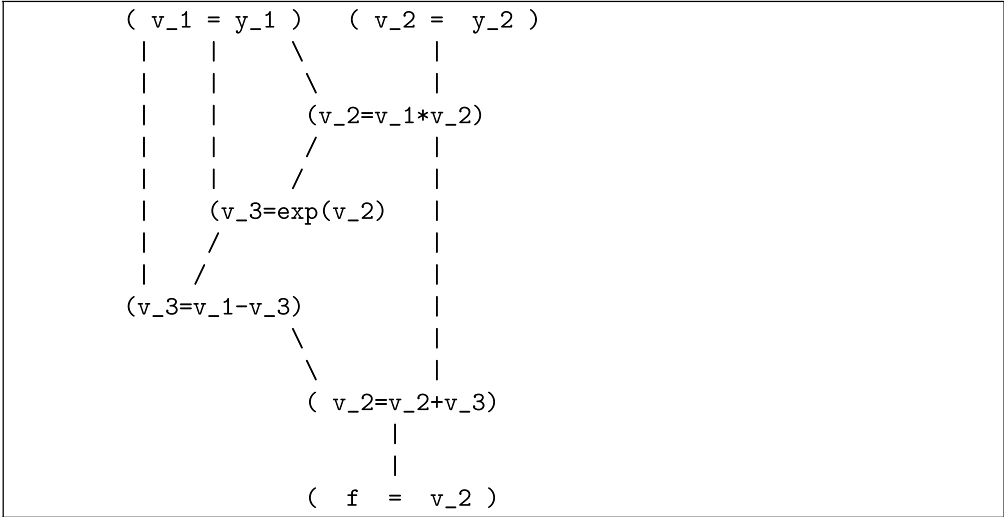
relating the changes in each of the v_i, where $\phi'_j(v_{j-1})[\dot{v}_{j-1}]$ denotes the derivative of $\phi_j(v_{j-1})$ in the direction $\dot{v}_{j-1}$. This is the *forward mode* of algorithmic differentiation;

from changes in the inputs compute the corresponding changes in the outputs. The forward mode allows one to compute the Jacobian of a function times a vector without explicitly forming the Jacobian. The operations needed for forward mode can be done at the same time the operations for evaluating the function are performed, in exactly the same manner as in Example 2. Forward mode corresponds to computing tangent vectors. By choosing $\dot{v}_0 = e_i$ one can determine the components of f that depend on the ith component of y. For the dependency list we need the reverse, i.e. which components of y does f_i depend upon.

The solution of optimization problems is often concerned with maximizing or minimizing expressions of the form $\lambda^T f$. The vector λ may correspond to some direction of interest for the output vector f or λ may be a vector of Lagrange multipliers. Let $\overline{v}_i$ denote the gradient of $\lambda^T f$ with respect to v_i then $\overline{v}_k = \lambda$. The jth entry of $\overline{v}_i$ is the partial derivative of $\lambda^T f$ with respect to the jth entry of v_i. The $\overline{v}_j$ are related as follows

$$\overline{v}_{j-1} = (\phi_j'(v_{j-1}))^T \overline{v}_j, \tag{3.3}$$

which relates the gradient $\overline{v}_{j-1}$ to the gradient $\overline{v}_j$. The quantity $\overline{v}_0$ gives the gradient of $\lambda^T f$ with respect to the input vector x. This is the *reverse mode* of automatic differentiation. Given a direction λ for measuring change in the output vector y, reverse mode gives the gradient, i.e. change in the input vector that will maximize the change in $\lambda^T y$. Thus, reverse mode allows one to compute the Jacobian of a function left multiplied by a vector. This is useful for computing gradients and adjoints. For DEIM, this is particularly useful for determining the dependency list. By choosing $\lambda = e_i$ one can determine the components of y upon which the ith component of f depends. We will use the reverse mode to do this and to determine the operations need for evaluating the f_i.



Example 4: Computational graph of first example.

Note from Example 2 and (3.2) that the forward mode of algorithmic differentiation can easily be implemented alongside the same code for evaluating the function.

```
fbar = lambda
-------------------------
pop v_2;
vbar_2 = fbar;
-------------------------
pop v_2, v_3;
vbar_2=vbar_2;
vbar_3=vbar_2;
-------------------------
pop  v_1,v_3;
vbar_1=vbar_3;
vbar_3=-vbar_3;
-------------------------
pop v_2
vbar_2=vbar_2+v_3*vbar_2;  <--Notice we accumulate here
-------------------------
pop v_1,v_2;
vbar_1 = v_2*vbar_2;
vbar_2=v1*vbar_2
-------------------------
ybar_1=vbar_1;
ybar_2=vbar_2;
```

Example 5: Reverse mode of AD for first example.

As the function is evaluated, the values and derivatives of any intermediate variables become available as they are needed. This is not the case for reverse mode. Reverse mode, as the name suggests, requires traversing the code and variable values in reverse. However, in order to traverse the variable values in reverse the function must first be evaluated and during the evaluation the values of all intermediate variables must be recorded. Typically, the values of the intermediate variables, the v_i in (3.3), are stored in a last in first out stack. A record of the operations performed is a representation of the *computational graph*. The computational graph is a directed acyclic graph that shows how the operations in the code depend upon one another. Each node in the graph represents one basic operation performed in the code. The nodes are generated in the same order in which they would be evaluated in the code. An edge in the graph indicates that the output of the starting node is needed as input for the ending node. For a computational graph made up of basic operations, a node may have many outgoing edges but at most three incoming edges. As the computational graph is traversed in reverse, the values of the intermediate variables are restored from the stack. As reverse mode requires at least one forward sweep through the code, computing a single gradient or adjoint using algorithmic differentiation is at least as expensive as evaluating the function twice [6].

The computational graph for the example is shown in Example 4.
Evaluating the gradient of $\lambda^T f$, $\bar{f}$ is shown in Example 5.

We will use the record of the computational graph needed for reverse mode to determine the dependency list and to generate code for computing the f_i. For operator overloading the differentiation is done at run-time by observing each of the basic

operations. As a consequence for operator overloading the computational graph will have a record of all of the operations needed to evaluate the function. All loops are unrolled and any aliasing due to pointers and references is undone. Operator overloading is *context sensitive*, in that if there is a branch in the code, then only the branch that is evaluated is recorded in the computational graph. The branch recorded depends upon where the function is evaluated. Source transformation, as it is done at compile time, takes a more modular approach to the code. For every branch or loop block, a separate computational graph is formed and code for forward and reverse mode is then inserted into the original code based upon this computational graph. The modular approach of source transformation generally leads to a much smaller computational graph. Also, source transformation is *context insensitive*, all branches of the function are available. Hence the function derivative can be evaluated wherever the function is differentiable. However, source transformation is much more difficult to implement than operator overloading. For this reason, since our purpose here is only to demonstrate feasibility, we shall use operator overloading to determine the computational graph. Ultimately however, generating efficient code for evaluating the components of a vector valued function f will likely require a combined use of techniques from both source transformation and operator overloading.

4. Generating Reduced Code. Reverse mode gives a tool for determining the dependency list for components of the output variable f. For reverse mode, operator overloading generates a context sensitive computational graph G that has a record of every operation needed to evaluate the function. With the computational graph, reduced code for evaluating components of f can be determined as follows.

1. Input: $V_r \subset G$ the desired leaves of the computational graph. They are the nodes that assign values to desired components of the output vector f.
2. Mark the nodes in V_r nodes as nodes that will be in the reduced computational graph.
3. Determine all nodes $V_i \in G \setminus V_r$ that have outgoing edges to nodes in V_r.
4. Mark the nodes in V_i nodes as nodes that will be in the reduced computational graph.
5. Let $V_r = V_r \cup V_i$.
6. Repeat steps 3-5 until $V_i = \emptyset$.
7. The nodes in V_r are the operations needed for computing the desired components of the output.
8. The input variable nodes in V_r are the independent variables for the reduced computational graph.

This algorithm involves performing a reverse sweep on the computational graph as in the reverse mode of AD. In general, the selection of portions of code of a program using some criteria is known as program slicing, see [14]. Program slicing was originally introduced for program debugging. In AD, activity analysis of differentiated code based on independent and dependent variables is used to determine the variables and lines of code needed to compute the derivatives of the dependent variables. The unnecessary lines and variables are then sliced from the code. For the algorithm above only the desired dependent variables are specified. The necessary independent variables and the necessary operations are then determined from the computational graph.

The algorithm above only reduces the computational graph. To yield a savings in memory usage, unnecessary variables must be removed as well. This can be ac-

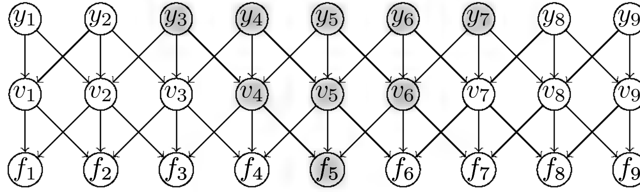


FIG. 4.1. *Computational graph for $f = L^2 y$. The highlighted nodes are the nodes needed for f_5, the fifth component of the output vector f.*

complished simply by noting the variables used in the reduced computational graph and then slicing out, via relabeling, the unused variables. Note, this may not yield optimal memory usage, as the reduced computational graph may allow for the reuse of more intermediate variables than the full computational graph. However, if this simple approach does not significantly reduce the amount of memory required then it is likely, that DEIM will not yield a savings in complexity.

To further illustrate the use of the computational graph, consider the example shown in Figure 4.1 with

$$f = L^2 y,$$

where $y, f \in \mathbb{R}^9$ and $L \in \mathbb{R}^{9\times 9}$ is a tridiagonal matrix having ones along the super and sub diagonals and negative two on the main diagonal. The shaded nodes in Figure 4.1 indicate the nodes that are needed for computing the fifth component of the vector f. These are the nodes that would remain in a reduced computational graph for computing f_5.

5. Implementation. At the suggestion of Paul Hovland of Argonne National Lab, to implement the approach outlined above we decided to use the package ADOL-C which stands for Automatic Differentiation by Overloading in C++. ADOL-C overloads all basic arithmetic operations for variables of type `adouble` as discussed above. ADOL-C stores the computational graph as needed for reverse mode in the form of a tape. A tape is essentially a pseudo-assembly language code as shown in Example (5.1). Each entry in the tape indicates the type of operations, and the memory locations involved.

Generating the computational graph from a tape requires mapping each of the memory locations to a particular operation. This can be done by performing a forward sweep through the tape. If operation n writes to location a then we would record that the parent operation of a is currently n. If operation n uses locations b and c as inputs then we would record that the operation depends on the parent operations of b and c. With all the locations mapped to their parent operations, we have a computational graph from which we can determine a reduced computational graph. Generating a reduced tape then requires removing unnecessary operations and then relabeling memory locations. As a tape is essentially pseudo-assembly code, translating a tape to a language such as C is trivial.

Using ADOL-C's `tapedoc` function as a starting point, we created the function `reducetape`. The function `reducetape` takes as an input a tape, created with the important outputs designated as dependent variables. The function creates a reduced

		code	op	loc	loc	loc
0	10	33	start of tape			
1	9	39	take stock op		8	0
2	8	1	assign ind			1
3	7	1	assign ind			2
4	6	15	mult a a	1	2	2
5	5	19	exp op		2	3
6	4	13	min a a	1	3	3
7	3	11	plus a a	2	3	2
8	2	2	assign dep			2
9	1	0	death not		1	7
10	0	32	end of tape			

TABLE 5.1
Example of a ADOL-C Tape

tape and `C` reduced function. The function returns a list of the indices of the input vector that are needed to evaluate the reduced function. The steps required are as follows,

1. Inputs: C function `void fun( int m, int n, double * y, double * f)`, where y is the input vector with length n and f is the output vector with length m.
   A vector `p` indicating the desired components of the output vector f.
2. Change active variables in `f` from `double` to `adouble`.
3. Designate that all entries of y are independent variables.
4. Designate using the entries of vector p the desired components of f as dependent variables.
5. Record a tape of the evaluation of f at a desired input vector y. Store the tape id in `tnum`.
6. Call reducetape on the tape:
   `reducetape(tnum,m,n,y,f, "reduced_func", Pin, len_Pin)`
   where `reduced_func` is the desired name of the reduced function. The variable `Pin` is a vector indicating which entries of the input vector y are needed to compute the desired entries of f. The variable `len_Pin` indicates the length of `Pin`
7. The code for the reduced function is in `reduced_func.c`. The syntax is
   `reduced_func(double * yred, double * fred)`,
   where the components of `yred` are determined from y by the vector of indices `Pind` and the entries of `fred` are determined from f by the entries of the vector `p`.

6. Miscible Flow Example. In this section we apply the procedure above to the example considered in [2]. The example involves nonlinear miscible viscous fingering in porous media. The code for the example was originally written for Matlab. To use ADOL-C we had to convert the example to `C`. We wish to note the accuracy and the speed of the resulting code. For this particular example, the reduction of the nonlinear terms can be done by hand. We compare the runtimes of the reduced model with AD generated code and with manually generated code in Table 6.1. For this example we had a total of 250 snapshots. We also recorded the time required

Without compiler optimization

Size		Number of ops		Time				
Full	Reduced	Full	Reduced	Full	Prune	AD	Manual	Error
15000	20	1651827	2803	51.1	0.14	0.69	0.06	0.039
15000	40	1651927	5483	51.1	0.41	0.91	0.23	0.030
60000	20	6603805	2875	247.9	0.14	1.61	0.06	0.017
60000	40	6603905	5621	247.9	0.42	1.87	0.23	0.002

With compiler optimization `-O2`

Size		Number of ops		Time				
Full	Reduced	Full	Reduced	Full	Prune	AD	Manual	Error
15000	20	1651827	2803	51.7	1.30	0.10	0.06	0.039
15000	40	1651927	5483	51.7	2.55	0.34	0.23	0.030
60000	20	6603805	2875	247.6	2.33	0.10	0.06	0.017
60000	40	6603905	5621	247.6	4.73	0.36	0.24	0.002

TABLE 6.1

Results for Model reduction on Miscible Flow example from [2]. Note that the Prune time also includes the time required to compile the reduced functions.

to generate the reduced functions. The time for AD To generate the reduced code grows linearly with size of the reduced system, which for our example is also equal to the number of DEIM points. Without any compiler optimization, the AD system took no more than twice the time of the manual system. With compiler optimization the AD system provided a reduction in run-time that is comparable with that of the manual system. However, it does take longer to compile the reduced code with compiler optimization.

7. Discussion. DEIM is an effective way of reducing the complexity of nonlinear functions in reduced order models. The techniques of algorithmic differentiation are useful for realizing the computational savings of DEIM. Operator overloading is a straightforward and fool proof AD technique for generating reduced code, i.e. code for evaluating important components of a nonlinear function. However, the reliability of the operator overloading approach comes at a cost. Operator overloading is context sensitive, hence the reduced function captures only the evaluated branch of the code. Operator overloading unrolls all loops and hence can lead to code that is much larger than the original code. For a large system this can lead to a prohibitively large tape for the unreduced system.

The problems of context-sensitivity and tape size can be overcome by incorporating other AD techniques. Source transformation by taking a modular approach is capable of generating context-insensitive code. Whereas operator overloading focuses on each evaluated operation, source transformation is focused on loop, branch and function bodies. Such modularity lends itself to producing smaller computational graphs. To take advantage of modularity using ADOL-C would require augmenting the set of operations recognized by ADOL-C. This can be done using ADOL-C externally defined functions. Source transformation can be used to modularize the code, transforming loops, branches, and functions, into ADOL-C externally defined functions. Operator overloading can then be performed on the modularized version of the code. Proper handling of branches will require a more advanced form of conditional assignment then that already supported by ADOL-C. Modularizing the original code

using source transformation and then performing operator overloading can further reduce the size of the computational graph. However, even after incorporating more modularity, the resulting computational graph for the original function may still be too large.

To handle functions whose corresponding tape would be too large, there is the method of checkpointing [5]. Checkpointing is an automatic differentiation technique for performing reverse mode on large computations, i.e. computations for which it is not feasible to form the entire tape. Issues with tape size arise for example when computing gradients associated with solutions of initial value problems. The numerical solution of an initial value problem requires either a explicit or implicit scheme for propagating the initial conditions forward through time. Maximizing some function of the solution at the final time with respect to the initial conditions requires the computation of adjoint or gradient which can be done via the reverse mode of AD. Computing such a gradient via AD would require forming a computational graph that includes all the steps of explicit/implicit scheme needed to get to the final time.

Checkpointing gets around the problem of large tapes by taking advantage of loops in the code. Since the same steps are performed in a loop body, a tape need only be recorded once for one evaluation of the loop body. The same tape can then be reused for propagating gradients in reverse mode. The *checkpoints* would be at the start of each loop body. At each checkpoint the value of the inputs needed for the loop body are recorded. Recall that reverse mode requires creating a record of the values of all the intermediate variables. By storing the value of the solution at the checkpoints, the value of the intermediate variables between any two checkpoints can be recomputed, thereby also saving on the number of values of intermediate variables that need to be stored. Propagating a gradient using checkpointing at each step requires performing a forward sweep from the previous checkpoint to the current checkpoint to generate a record of the intermediate variables followed by a backward sweep from the current checkpoint to the previous checkpoint to propagate the gradient.

The same idea of checkpointing can be applied to generating reduced code for DEIM. Instead of making one long reverse sweep through a large tape for the original function which would require storing the complete tape, one can make many small forward and reverse sweeps through smaller tapes. The only issue with applying checkpointing to reducing a general nonlinear function is the placement of the checkpoints. The numerical solution of differential equations with time steps provides natural checkpoints. For a general nonlinear function there may not be any natural points in the program for checkpoints.

To sumarize, this paper has presented a description of how to use AD techniques to automate the generation of a code for a reduced order nonlinear function derived from a DEIM approximation to a given function. A prototype system was developed to implement this approach and its feasibility was demonstrated by reproducing computational results of DEIM model reduction for a quite complicated miscible flow simulation. Although not implemented here, it is clear that the Jacobian for the reduced model could be automatically generated as well. The generation of code for evaluating reduced functions is a time consuming and error prone tasks that would have to be done by hand without the assistance of the AD techniques. There is considerable potential in this approach for disseminating the DEIM for nonlinear model reduction to a far wider audience beyond experts in model reduction.

8. Acknowledgments. The authors are indebted to Paul Hovland and Jean Utke for valuable discussions and advice on AD techniques and how to adapt them

to the implementation of DEIM model reduction.

REFERENCES

[1] T. Bui-Thanh, K. Willcox, and O. Ghattas, *Model reduction for large-scale systems with high-dimensional parametric input space*, SIAM Journal on Scientific Computing, 30 (2008), pp. 3270–3288.

[2] S. Chaturantabut and D. Sorensen, *Nonlinear model reduction via discrete empirical interpolation*, SIAM Journal on Scientific Computing, 32 (2010), pp. 2737–2764.

[3] ———, *Application of POD and DEIM on dimension reduction of non-linear miscible viscous fingering in porous media*, Mathematical Modeling of Dynamical Systems, 17 (2011), pp. 337–353.

[4] K. Fukunaga, *Introduction to statistical pattern recognition (2nd ed.)*, Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[5] A. Griewank and A. Walther, *Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation*, ACM Trans. Math. Softw., 26 (2000), pp. 19–45.

[6] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Society for Industrial and Applied Mathematics, 2008.

[7] A. Kellems, S. Chaturantabut, D. Sorensen, and S. Cox, *Morphologically accurate reduced order modeling of spiking neurons*, Journal of Computational Neuroscience, 28 (2010), pp. 477–494. 10.1007/s10827-010-0229-4.

[8] B. Kreaseck, L. Ramos, S. Easterday, M. Strout, and P. Hovland, *Hybrid static/dynamic activity analysis*, in Computational Science ICCS 2006, V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, eds., vol. 3994 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 582–590.

[9] K. Kunisch and S. Volkwein, *Galerkin proper orthogonal decomposition methods for a general equation in fluid dynamics*, SIAM Journal on Numerical Analysis, 40 (2003), pp. pp. 492–515.

[10] J. Shin and P. D. Hovland, *Comparison of two activity analyses for automatic differentiation: context-sensitive flow-insensitive vs. context-insensitive flow-sensitive*, in Proceedings of the 2007 ACM symposium on Applied computing, SAC '07, New York, NY, USA, 2007, ACM, pp. 1323–1329.

[11] L. Sirovich, *Turbulence and the dynamics of coherent structures. I. Coherent structures.*, Quarterly of Applied Mathematics, 45 (1987), pp. 561–571.

[12] ———, *Turbulence and the dynamics of coherent structures. II. Symmetries and transformations.*, Quarterly of Applied Mathematics, 45 (1987), pp. 573–582.

[13] ———, *Turbulence and the dynamics of coherent structures. III. Dynamics and scaling*, Quarterly of Applied Mathematics, 45 (1987), pp. 583–590.

[14] F. Tip, *A survey of program slicing techniques*, JOURNAL OF PROGRAMMING LANGUAGES, 3 (1995), pp. 121–189.