Adjoint Differentiation of Hydrodynamic Codes

Maria L. J. Rightley[(1)], Rudolph J. Henninger[(2)] and Kenneth M. Hanson[(3)]

[(1)] XNH, MS–F664
[(2)] XHM, MS–D413
[(3)] DX-3, MS–P940

Los Alamos National Laboratory, Los Alamos, NM 87545

April 1998

Abstract

Many problems in physics and modern computing are inverse problems – problems where the desired output is known, and the task is to find the set of input parameters that will best reproduce that output in a hydrodynamics code (hydrocode). Optimization methods tackle this type of problem, and a central task in applying optimization methods is to be able to determine the gradient of the output with respect to the input parameters that are being adjusted. Presented here is the authors' progress (through the use of adjoint differentiation) in obtaining those gradients in the case of some relatively simple hydrocodes.

1 Introduction

When a program simulates a physical system, it does so through the use of a set of equations and mathematical relationships known as a physical model. This model will generally contain a number of parameters that influence the system. Depending on the problem of interest, there may be several situations that arise. One such situation is the inverse problem, where the output (or at least the desired output) is known and it is the input parameters that need to be determined.

An example of an inverse problem is the flyer-plate experiment. The flyer-plate experiment is conducted to help determine material properties. It can be used to help determine many material properties, but let us consider one specific regime or behavior: spall. When a material is subjected to certain forces, it will begin to accrue defects or breakage – this is the

phenomenon known as spall. A schematic of the process is shown in Figure 1. There are two plates involved, the flyer-plate and the target plate. The target is initially stationary and generally thicker than the flyer-plate (but other dimensions are the same), and the flyer-plate is sent toward the target plate at high velocity, usually propelled by either a gas gun or high explosives – this is shown in the first "line" of the figure. The flyer-plate impacts the target plate and begins to push the target plate (second line). When they impact, a shock wave is sent out from the impact plane into both materials (denoted by the thicker lines, with the small arrows indicating direction of movement, not relative velocity). These shock waves eventually reach the other (non-impact) boundaries of these two plates, at which point the shock wave reflects back into the two plates as rarefaction waves (the third line is supposed to represent a time just after this has occurred, and one can see that the direction of wave movement has now reversed in both materials). These rarefaction waves will continue until they meet in the target plate (due to the relative widths of the plates that was discussed above). The fourth line of the figure is a time just before this has occurred. When they meet, they act to pull the material apart at the intersection plane. Depending on the material strength and the original conditions present, this will happen to varying degrees. When the interest is not in spall behavior, lower pressures and velocities are used, and spall does not occur.

Experimentally, the velocity of the leading edge of the target plate (the rightmost edge in Figure 1) is measured, for example, by a method of interferometry such as VISAR. A window material is usually added at the leading edge of the target plate, and a laser beam is shined through the window material and reflects from that leading edge. How that beam has been altered when it reaches the receiving optics allows the velocity to be determined. When conditions are changing within the material (i.e., the initial impact, the spallation – especially if the material breaks), it is understandable that that leading edge velocity will change. So we end up with a velocity trace at the leading edge, and we want to determine material parameters. We also have a program (CHARADE [1], for example) that will simulate the experiment, given certain material input parameters, and produce a calculated velocity trace. The task is then to adjust the material input parameters to get the best match of the calculated velocity trace to the experimental velocity trace.

That is a problem of the sort described above as an inverse problem. It is for problems just like this (among others) that optimization methods are found to be very useful, because they perform the process of optimizing the parameters for the user. Given an initial guess for the input parameters, the optimization algorithm with iterate, adjusting the parameters until some criterion is satisfied, usually some cost function that is minimized until the cost function falls below some specified value. There are many optimization algorithms available, although a popular and useful method of optimization is the Bayesian method [2–6], which has had considerable success in reconstructing radiographic data [7, 8]. The most efficient optimization algorithms (including the Bayesian method) make use of the gradient(s) of the output with respect to the input parameters. In most cases, the output of interest is a cost function as just described – for the example cited above, it would be probably be a χ^2-type cost function comparing the experimental and calculated velocity traces. The input param-

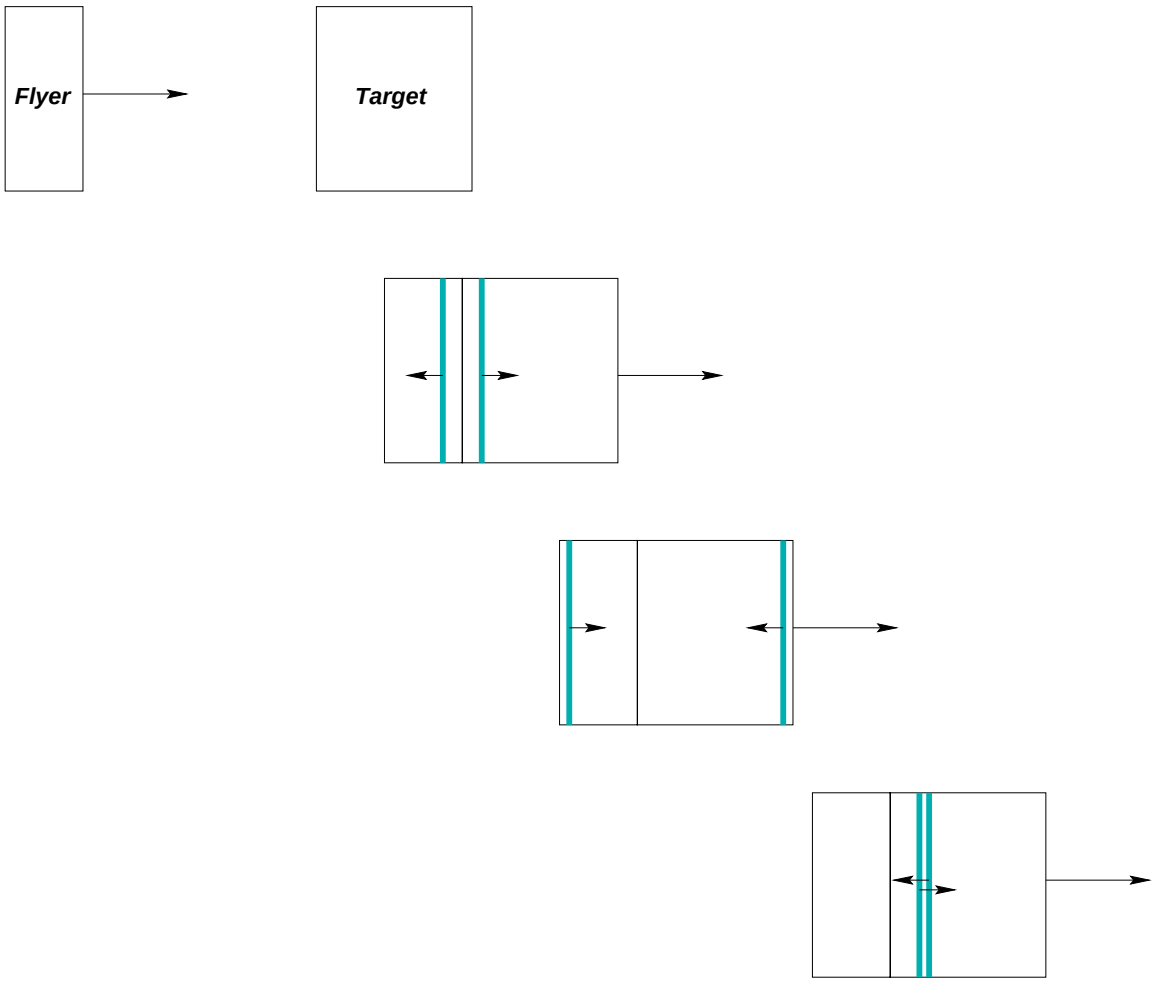

Figure 1: Flyer Plate Experiment Setup

eters of interest for the example given would be the material parameters relating to spall). Therefore, a primary component in optimization of an inverse problem is determination of those gradients; various common methods of obtaining those gradients are discussed in the next section.

2 Gradient Methods

2.1 Perturbation/Direct Method

One straightforward method is to simply run the program twice, with different values of the parameter (i.e., perturbing the value of the parameter). Dividing the difference in the output by the difference in the parameter gives an approximate derivative, or sensitivity. Problems with this include the fact that it is difficult to determine *a priori* what the ap-

propriate difference in the parameter should be; these sensitivity values may thus vary with the parameter difference in some regimes of parameter values, necessitating multiple runs to determine convergence of the sensitivities. In addition, at least two program runs are necessary to calculate each sensitivity; for n parameters, this leads to at least $n + 1$ runs of the program (more if one cannot magically divine the proper perturbation to use) – this is, therefore, a computationally expensive method, especially for long-running, complicated programs.

2.2 Equation-based Methods

Another method is to differentiate the equations (differential, for example) that make up the physical model, then combine them to define the needed sensitivities. A concern with this model is that the sensitivities found do not necessarily represent the gradients of the quantities calculated in the program; if finite-difference equations are used in the program, this method doesn't produce sensitivities of finite-difference equations, it produces the sensitivities of the differential equations. Why this would be of great concern can be seen if one considers that this is part of a larger task, which is to optimize the parameters of the system. If the adjustments to the parameters are based on something other than what is in the code, it is possible that convergence to the optimum parameter set (i.e. those that produce the best calculated fit to the experimental output) would never be achieved, because the adjustments don't fit the computational system. Whether this misfit has a significant impact on the sensitivities or convergence to them is difficult to determine, but at any rate, one can not necessarily assume that the values obtained are the sensitivities of the model/code. The DST method [9–14] is an example of this type of method, where the differential equations associated with a problem are differentiated. A closely related method, DSA [12–14], differentiates the difference equations instead of the differential equations and is more closely linked to the code-based methods discussed below. Work with these methods by the second author and others on the MESA2D code has had mixed success [15–17], and that is one of the reasons that the straightforwardly code-based methods were considered and attempted, as a check and counterpoint to the DSA/DST methods.

2.3 Code-Based Methods

The method of interest in this study was to differentiate the code itself in order to determine the sensitivities that are of interest, thereby avoiding the difficulties that the equation-based methods can have. There are two ways to do this – manually, by examining the code and producing appropriately programmed derivatives; or automatically, using a tool that will look at each line of code and produce derivative values automatically. The first of these two ways is bound to be more efficient, because the concern of the automatic tools is not efficiency, it is overall applicability. Both automatic and manual methods were implemented in this study.

3 Forward vs. Adjoint Modes of Differentiation

Both the code- and equation-based methods can ideally operate in both the forward and adjoint modes. By forward and adjoint, we mean the direction through the code in which the derivative values are obtained. A forward mode of differentiation would involve determining the necessary derivatives by following the code's logic in the forward direction (top to bottom, front to back), while for the adjoint mode, the derivatives are determined by following the code's logic in the reverse direction (bottom to top, back to front). Which of these is more useful depends on the relative numbers of input parameters of interest and output variables of interest. The forward mode is more efficient for determining the sensitivity of many outputs to one or a few input parameters, while the adjoint mode is better suited for sensitivities of one or a few outputs with respect to many input parameters.

As was mentioned earlier, it is common to provide a cost function that computes the relative difference between a calculated and experimental data set for the problem of optimization. If that is done, we have provided ourselves with just one output of interest, although there are obviously a lot of other variables that contribute. As was also mentioned above, there may be several parameters of interest (in the example cited, these would the material parameters related to spall). This points us in the direction of the adjoint mode, which was most efficient for one or a few outputs and many input parameters. More is discussed about the adjoint mode below, but it should be pointed out that with many of the automatic differentiation tools, both modes are available and were easily implemented, so that these modes were also used. Given that both modes produced the same results, results will not be listed independently for each mode.

4 Mechanics of Adjoint Differentiation

Consider that a program could be represented in terms of a flow diagram such as that shown in Figure 2 below. Granted, this is a simplification, but the principle should still stand for more complex situations. $\mathbf{x}$ is the input, ϕ is the output, presumably the scalar cost function, and **A, B** and **C** are the processes or transforms to which the input is subjected, and the output of process **A** is $\mathbf{y}$, and that of **B** is $\mathbf{z}$, and thus obviously ϕ is the output of **C**. That sequence of processes is what is considered the forward calculation.

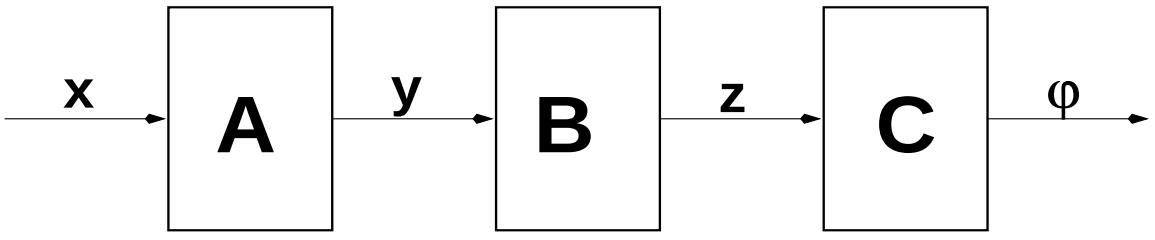

Figure 2: Data Flow Diagram of the Forward Calculation

x, y and **z** are general data structures; they can consist of mixed types of data structures. Some of the data may even be parameters that affect the transformations or processes themselves. There is no loss in generality if they are thought of as being carried along in the sequence of data structures up to the module at which they are used. In fact, it is necessary that the input to a process must be all that is required in order to determine the output of that process, so one could almost consider that the data structures are all of the data, and the only changes are in variables from input to output of the process is in variables affected by that particular process. With that in mind, these structures can have high dimensionality. We also do not place any restrictions on the processes, other than that they be differentiable (and functions not obviously differentiable can often be handled, also). By requiring that the input to a process be all that is necessary to produce the output, we have thus required that each transformation or process is self-contained.

Considering the possibly high dimensionality of the data structures, storing the sensitivity matrices of the transformations, such as $\frac{\partial y_j}{\partial x_i}$ for all i and j, is likely to be extremely costly, because one is multiplying the dimensionalities of $\mathbf{x}$ and $\mathbf{y}$, which may already be large. The chain rule, however, allows the calculation of the output ϕ with respect to the ith component of $\mathbf{x}$ –

$$\frac{\partial \phi}{\partial x_i} = \sum_{jk} \frac{\partial \phi}{\partial z_k} \frac{\partial z_k}{\partial y_j} \frac{\partial y_j}{\partial x_i}.$$

Even if a process is nonlinear, the expression above amounts to a product of matrices, each element of which specifies the differential response of an output variable with respect to a differential change of an input variable. The order of summations can be done two ways, either over j first or over k first. If the summation is done over j first, one is going in the same direction as the forward calculation and is therefore in the forward mode discussed above. The data-flow diagram in Figure 3 illustrates the forward mode process (where the derivative notation is that the subscript is what the derivative is taken with respect to, so that $\mathbf{y}_x$ is the derivative of $\mathbf{y}$ with respect to $\mathbf{x}$, and by studying the data flow diagram this, one can see how the forward mode is not optimum for a situation with many input parameters and one output of interest. If $\mathbf{x}$ is large, then the results of the first summation ($\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$) can be very large (dimensionality = dimensionality of $\mathbf{x}$ times the dimensionality of $\mathbf{y}$), and so on through the process until the last step, which reduces just to $\frac{\partial \phi}{\partial \mathbf{x}}$. ϕ is a scalar, so

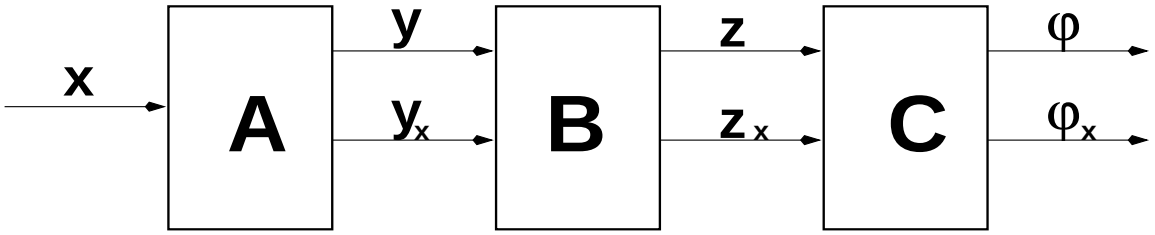

Figure 3: Data Flow Diagram of the Forward Derivative Calculation

that our final result has just the dimensionality of $\mathbf{x}$, while the intermediate results had the dimensionality of either $\mathbf{x} \times \mathbf{y}$ or $\mathbf{x} \times \mathbf{z}$. As these data structures can be very large, this can result in extremely large intermediate results that need to be stored.

Summing over k first, on the other hand, yields the adjoint mode, and the sequence of events goes backwards from ϕ. Figure 4 illustrates this sequence in a data-flow diagram. The notation for the derivatives is given in the same way that it was for the forward mode. The adjoint mode of differentiation can be seen to be useful; since ϕ (a scalar) is what is always being differentiated, the dimensionality of the possibly large data structures are never multiplied together as they are in the forward mode. Instead of storing the matrix of the adjoint of each process ($\frac{\partial \phi}{\partial \mathbf{z}}$, $\frac{\partial \mathbf{z}}{\partial \mathbf{y}}$, and $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$), only the intermediate data structures given in the previous sentence are formed and stored. Thus the requirement for storing these data structures is only about double that required to store the structures for the forward calculation (the forward calculation structures might also be required for the sensitivity calculation if the processes are non-linear).

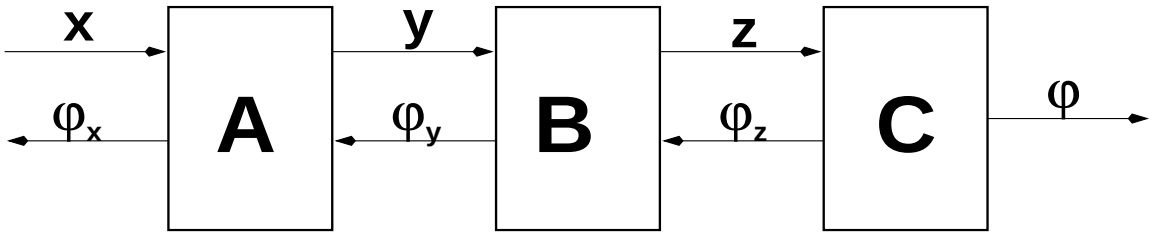

Figure 4: Data Flow Diagram of the Adjoint Derivative Calculation

This is the basic adjoint differentiation technique, and it is the method that is followed in the manual implementations, but it might not necessarily be exactly the way that the automatic differentiation programs do things. The automatic differentiation tools are discussed below.

5 Automatic Differentiation Tools

Although all of these tools vary, there are some things that they all have in common. All have two stages involved to get from the original code to an executable code with derivative coding included. The first step is to submit the original code to a precompiler. This precompiler analyzes the code and modifies it to include code that calculates the derivatives of interest. The output of this step is enhanced code, with some calls to external subroutines. The second step in the process is to compile this enhanced code, including run-time libraries that satisfy the external subroutine calls (for storage and memory manipulation, usually). In addition, all the codes considered in this exercise have been written in FORTRAN, and automatic differentiation tools that I have used work on FORTRAN code. ADIFOR has a version for C, known as ADIC, but TAMC and GRESS, to the authors' knowledge, are only available for FORTRAN programs.

5.1 ADIFOR

The Automatic DIfferentiation of FORtran (ADIFOR) program was developed jointly by the Mathematics and Computer Science Division of Argonne National Laboratory (Christian Bischof, Peyvand Khademi, and Andrew Mauer) and Alan Carle of the Center for Research on Parallel Computation at Rice University [18]. ADIFOR has limited platform availability for the precompiler stage, and only provides executables. For the libraries, you get source code that is given for several platforms, and it may be possible to extend use beyond those platforms by modifying some of the files. Additionally, ADIFOR currently runs in **forward** mode; there is no adjoint mode available.

As for specifics of use, ADIFOR has a very specific set-up that must be followed. There are companion files (to the code to be enhanced) that have to have a specific format and content. ADIFOR requires that the bulk of the problem be in a subroutine. All of the parameters and outputs of interest must be passed from the main program to the subroutine, and the activity relating the parameters to the responses must be in the subroutines. Other variables can be passed to the subroutine, but the parameters and responses **must** be. This means that the values of the parameters must be set in the main program. The main program and major subroutine should also be in separate files, because of the way that the precompiling operation works. There are changes to the main program that must be done manually by the user, in order to seed the gradient values (essentially setting up the identity matrix mentioned in the mechanics section). The enhanced code is retrieved from a newly-created subdirectory and compiled with the proper subdirectories. ADIFOR does not automatically output the gradients – the user needs to do that manually.

5.2 GRESS

The GRadient Enhanced Software System (GRESS) was developed at Oak Ridge National Laboratory, and is one of the many codes in the Radiation Shielding Information Center's RSIC Peripheral Shielding Routine Collection [19]. Jim Hordewel was the primary author, but is no longer directly involved with GRESS. GRESS provides source for both the precompiler and the libraries; although it is apparently platform-specific code, it could conceivably be expanded beyond the specific platforms provided. Both forward and adjoint modes are available.

GRESS, unlike ADIFOR, does not require external files to give its precompiler command information. It also does not require splitting the program a certain way. With GRESS you simply put two kinds of additional programming into the code; precompiler directives and subroutine calls. Precompiler directives tell the precompiler certain general things, such as whether to pass comments to the enhanced codes or not (*comments on/off) or whether this implementation is to be a CHAIN implementation (*chain). Subroutine calls are used to tell the precompiler which variables are to be considered parameters and outputs, or to define this implementation as an adjoint (ADGEN) one (which works differently than when it is a chain implementation, and doesn't really explain why). In the interests of space,

specific commands beyond these will not be discussed. Suffice to say that many of the subroutine calls vary depending on whether adjoint for forward mode is being used, which implies that switching from one subroutine to another merely involves changing subroutine calls. GRESS does require the user to move files around because of its oddities in input and output file names, but a short script file can simplify the process greatly. As with ADIFOR, it is necessary to compile the enhanced code with libraries. Unlike ADIFOR, in all versions of GRESS implementation, the gradients and sensitivities are automatically output.

5.3 TAMC

TAMC's acronym comes from Tangent-linear and Adjoint Model Compiler [20, 21]. It was developed by Ralf Giering while a graduate student at the Max Planck Institute for Meteorology, where the primary interest was in applying it to oceanographic simulation codes. He is currently a postdoc at MIT, and has been successful in applying The precompiler stage is actually run by sending the forward code by email to Ralf's computer in Germany, so platform availability for that stage is unlimited. The source for the libraries is available for several platforms. The Tangent-linear part of the acronym denotes the forward mode, so that both modes are available.

In terms of use peculiarities, TAMC is like ADIFOR in that it does require you split the main program from the subroutine. However, instead of giving the program information through external files, TAMC uses Makefiles, and information is put in the Makefile. TAMC is also different in that it does a dependency analysis to determine on which variables the gradients of interest are dependent, and it only determines adjoint code for those variables. It is also different in its implementation of the enhanced code. Rather than just adding code to the forward code, TAMC develops its own forward code and adjoint code separately, and puts them in two subroutines (presumably of the main code).

As mentioned in Section 4, if any of the processes are non-linear, information from the forward calculation is needed in the adjoint calculation. TAMC will also inform the user when a variable from the forward calculation is needed for the adjoint calculation in its output with a RECOMPUTATION WARNING; these warnings can be Level 1 or Level 2, depending on the amount of recomputation necessary. The need for this information can be satisfied one of two ways: by recomputation of forward results in the adjoint calculation, or by independent storage. TAMC defaults to recomputation, but the user can also choose to store the variables independently (via dynamic memory, static memory, common blocks, or disk files). The command to initialize these storage methods and the actual storage commands are about the only modifications that the user ever makes to his code. The main code, then, to calculate the adjoints, should call first the TAMC-generated forward code, then the TAMC-generated adjoint code, and like ADIFOR, the output of the gradients is the responsibility of the user.

6 Implementation and Results

For the first two codes listed below, the input problem that was considered was a 1-D flyer-like problem. An instantaneous decrease in the velocity at one boundary (simulating a reduced flow) causes a shock, resulting in a 1-D shock problem. The flyer is a metal (copper-like) that uses simplified EOS descriptions and parameters. The premise was to consider a reasonably physical problem, but to simplify where possible for ease of evaluating the results.

The output of interest for this problem is an average pressure value (integrated over all space and time). The parameters of interest are as follows:

gamma	Gruneisen coefficient
c0	nominal sound speed
rho0	nominal density
s	slope in linear u_s-u_p plot
rhobc	boundary condition on density
cq	quadratic artificial viscosity constant
clq	linear artificial viscosity constant
vic1	initial condition of velocity
vtbc	boundary condition of velocity (unchanged boundary)
vbbc	boundary condition of velocity (decreased velocity boundary)
rhoic	initial condition of density

6.1 Simple, 1-D Code

It was decided that this work should begin on a code with simple mechanics and basic materials considerations. The code chosen was written to mimic the actions of the popular code MESA2D in a one-dimensional, much simplified framework. It was approximately 300 lines long. It was submitted to each of the automatic differentiators and to the manual method, and the results for a specific set of initial conditions is given below. Much of this work was done for conditions both with and without energy evolution included in the code. Because the manual implementation was done only with the energy evolution **not** included, those are the results shown here.

Table 1: Results for simple 1-D program

Variable	GRESS-A	GRESS-F	ADIFOR	TAMC	Manual
gamma	-2.54358E+06	-2.543585E+06	-2543585.	-2.54358E+06	-2543584.
c0	1.53903E+05	1.539026E+05	153902.6	153903.	153902.5
rho0	-1.60778E+07	-1.607773E+07	-1.6077732E+07	-1.60778E+07	-1.6077755E+07
s	1.02751E+07	1.027515E+07	1.0275147E+07	1.02751E+07	1.0275145E+07
rhobc	1.99000E+05	1.990000E+05	199000.0	199000.	199000.1
cq	4.22026E+06	4.220258E+06	4220258.	4.22026E+06	4220262.
clq	7.98268E+04	7.982673E+04	79826.73	79826.8	79826.79
vic1	1.48481E+05	1.484787E+05	148478.7	148481.	148482.0
vtbc	-4.01215E+06	-4.012142E+06	-4012143.	-4.01215E+06	-4012149.
vbbc	3.82401E+06	3.824008E+06	3824008.	3.82401E+06	3824007.
rhoic	1.59265E+07	1.592650E+07	1.5926500E+07	1.59265E+07	1.5926487E+07

As one can see from looking at the numbers in Table 1, all of the methods agree exceptionally well, the only difference really being the number of significant digits output by the various methods.

6.2 MESA1D

Once full implementation of the methods was accomplished on the code mentioned above, work shifted to a one-dimensional version of MESA2D, referred to as MESA1D. Initial efforts (including the results posted here) included the further simplification of the strength being turned off within the code, but current work by one of the co-authors is very close to proper operation with the strength model activated. MESA1D is approximately 5000 lines long. The work with MESA1D was somewhat of a side project, and so there was also a decision made to not implement the manual method on this particular code, because on a complicated code (as opposed to the simple one-dimensional code that was 300 lines long and took about a week to adjoint code) this can be a very time-consuming endeavor, and not to be undertaken without considerable forethought. That is one of the reasons that we have been so interested in the automatic differentiation programs and have been searching for one that will efficiently handle the kinds of codes that one runs into on a daily basis at a national laboratory. The problem described in Section 6 was again considered for this work with MESA1D, and the values of input parameters used with the simple, 1-D code were used again here. However, it should be noted that all MESA1D results include energy evolution and will not therefore agree with the results in Table 1, but provide an opportunity to see what a small change in physics can make to the gradient/sensitivity values. The products of this effort are as follows, where the GRESS results are not listed independently for forward and adjoint mode:

Table 2: Results for MESA1D

Variable	GRESS	TAMC	ADIFOR
gamma	1.18086E+06	1.19352E+06	1.19272E+06
c0	1.57567E+05	156086.	156015.
rho0	-7.8636E+07	-1.59544E+07	-1.59544E+07
s	1.03009E+07	1.01880E+07	1.01828E+07
rhobc	3.0268E+05	0.	0.286909
cq	4.31418E+06	4.30264E+06	4.29965E+06
clq	8.19983E+04	81621.7	81586.5
vic1	2.72885E+05	268964.	272518.
vtbc	-4.18944E+06	-4.14965E+06	-4.15325E+06
vbbc	3.87530E+06	3.84001E+06	3.84002E+06
rhoic	1.27794E+07	1.60032E+07	1.60031E+07
e0	1.59549E+04	15876.6	15876.5
ebc	2.01001E+02	0.0	0.

The results for the automatic differentiators do not agree as well with each other for MESA1D as they did for the simple, 1D code. The GRESS results are much older (and obtained from a slightly modified version of the code) than the TAMC and ADIFOR results. The TAMC and ADIFOR results are, in fact, very recent, and the authors have not had time to certify that all three are trying to solve the same problem. For example, for ebc, GRESS disagrees with the other two; however, the value is low, and it is possible that numerical roundoff is coming in to play with GRESS, but not in TAMC and ADIFOR. Similarly, the rho values (rhoic, rho0, rhobc) differ between GRESS and the other two. The most likely reason for this is that the setting up of the problem differs between GRESS and the other two, but there has not been time to test this. For a more definite indication of method differences for the same version of MESA1D, compare the TAMC and ADIFOR results. There is much better agreement within those two sets of results, although one can observe that the results are still not as close together as they were for the simple, 1D code. In rhobc, for example, there is a discrepancy between the values, but although the ADIFOR result is not zero, it is definitely a small number. Additional work needs to be done to determine if GRESS is working on the correct problem, and it also remains to be seen whether these sort of differences would cause significant problems for optimization routines.

6.3 CHARADE

Progress with CHARADE has not been as fast as with the codes listed above; there are several reasons for this. First, while CHARADE is only about 3000 lines to MESA1D's 5000, its logic is significantly less straightforward than MESA1D's. Part of this may be

involved with the fact that MESA1D is a finite-difference code, while CHARADE is a method of characteristics code – CHARADE avoids introducing numerical viscosity at the cost of some contortions in programming. Second, CHARADE has three large, two-dimensional arrays that are very necessary to the adjoint calculation and thus are saved (at least by the automatic differentiators) *en masse* in more places than one would like. And, finally, the first author was a postdoc for the duration of most of the work, and is now a staff member in another division; this limits the available time left to work on this project. As it is, both GRESS and ADIFOR have been tried on CHARADE and have failed due to the memory requirements placed on a system by the enhanced code. Work with TAMC is still in progress and shows some hope of being able to handle everything with appropriate help; TAMC has more flexibility of use than either GRESS or ADIFOR. As for manual implementation, it was and is definitely a possibility, but the change in job by the first author means that there is basically no one presently available to complete that task; the manual coding would require a significant time investment and can not really be accomplished on a part-time basis.

7 Conclusions

This paper has described the efforts by the first author and others to apply code-based adjoint differentiation techniques to a few simple hydrocodes. With efficiently written code, the automatic differentiators are able to perform well and quickly produce accurate results. When the code is less efficiently written, or when the size of the program becomes very large, the automatic differentiators can have problems that are not always solvable. Of the three, TAMC seems the most sophisticated and flexible, and the authors believe that is has a promising future in producing adjoint derivatives in realistic codes. It has yet to be determined if the kinds of differences observed between ADIFOR and TAMC for MESA1D are large enough to have a significant impact on the optimization process.

As for manual implementation, with proper care, it can be implemented on any code, whether written well or not. The variable of import is the time that is required to set up and code the adjoint part of the problem. If the code is written systematically, and possibly with adjointing in mind, the procedure can be made much less painful. In addition, while the automatic differentiators might ultimately fail because of memory considerations, they are often useful in producing derivative code that can then be used to simplify the process of manually coding the derivatives (that is, the AD-produced code can be used, at least in some part, in the manual coding).

References

[1] J. N. Johnson and D. L. Tonks, "CHARADE: A Characteristic Code for Calculating Rate-Dependent Shock-Wave Response," Los Alamos National Laboratory Report, LA-11893-MS, UC-701, January 1991.

[2] K. M. Hanson, "Bayesian Reconstruction Based on Flexible Prior Models," *J. Opt. Soc. Amer. A*, **10**, p. 997, 1993

[3] S. J. Press, *Bayesian Statistics: Principles, Models, and Applications*, Wiley, New York, 1989.

[4] G. S. Cunningham, K. M. Hanson, G. R. Jennings, Jr., and D. R. Wolf, "An Interactive Tool for Bayesian Inference," in Review of Progress in Quantitative Nondestructive Evaluation, D. O. Thompson and D. E. Chimenti, Eds., **14A**, p. 747, Plenum, New York, 1995.

[5] K. M. Hanson and G. S. Cunningham, "The Hard Truth," in Maximum Entropy and Bayesian Methods, J. Skilling, Ed., Kluwer Academic, Dordrecht, 1994.

[6] K. M. Hanson and G. S. Cunningham, "Exploring the Reliability of Bayesian Reconstructions," in *Image Processing*, M. H. Loew, Ed., *Proc. SPIE*, **2434**, p. 416, 1995.

[7] K. M. Hanson, "Method to Evaluate Image-Recovery Algorithms Based on Task Performance," *J. Opr. Soc. Amer. A*, **7**, p. 45, 1990.

[8] K. M. Hanson, "Introduction to Bayesian Image Analysis," in *Image Processing*, M. H. Loew, Ed., *Proc. SPIE*, **1898**, p. 716, 1993.

[9] E. M. Oblow, "Sensitivity Theory for Reactor Thermal-Hydraulics Problems," *Nucl. Sci. Eng.*, **68**, p. 322 (1978).

[10] D. G. Cacuci, C. F. Weber, E. M. Oblow and J. H. Marable, "Sensitivity Theory for General Systems of Nonlinear Equations," *Nucl. Sci. Eng.*, **75**, p. 88 (1980).

[11] D. G. Cacuci, P. J. Maudlin and C. V. Parks, "Adjoint Sensitivity Analysis of Extremum-Type Responses in Reactor Safety," *Nucl. Sci. Eng.*, **83**, p. 112 (1983).

[12] P. J. Maudlin, C. V. Parks and C. F. Weber, "Thermal-Hydraulic Differential Sensitivity Theory," ASME paper No. 80-WA/HT-56, presented at the ASME Annual Winter Conference (1980).

[13] C. V. Parks and P. J. Maudlin, "Application of Differential Sensitivity Theory to a Neutronic/Thermal Hydraulic Reactor Safety Code," *Nucl. Technol.*, **54**, p. 38 (1981).

[14] C. V. Parks, "Adjoint-Based Sensitivity Analysis for Reactor Applications," ORNL/CSD/TM-231, Oak Ridge National Laboratory, 1986.

[15] R. J. Henninger, P. J. Maudlin, and E. N. Harstad, "Differential Sensitivity Theory Applied to the MESA Code, " Proceedings of the Joint AIRAPT/APS Meeting on High Pressure Science and Technology, p. 1781, Colorado Springs, CO (June 28-July 2, 1993).

[16] P. J. Maudlin, R. J. Henninger, and E. N. Harstad, "Application of Differential Sensitivity Theory to Continuum Mechanics," Proc. ASME Winter Annual Meeting 1993, p. 93, New Orleans, Louisiana (November 28-December 3, 1993).

[17] R. J. Henninger, P. J. Maudlin, and E. N. Harstad, "Differential Sensitivity Theory Applied to the MESA2D Code for Multi-Material Problems, " Proceedings of the APS Meeting on Shock Compression of Condensed Matter, p. 283, Seattle, WA, (August, 1995).

[18] C. Bischof, A. Carle, P. Khademi, and A. Mauer, "The ADIFOR 2.0 System for the Automatic Differentiation of Fortran 77 Programs," Argonne National Laboratory Report ANL-MCS-P481-1194 (1995).

[19] J. E. Horwedel, E. M. Oblow, B. A. Worley, and F. G. Pin, "GRESS 3.0 Gradient Enhanced Software System," Oak Ridge National Laboratory RSIC Peripheral Shielding Routine Collection Report PSR-231 (1994).

[20] R. Giering, "Tangent Linear and Adjoint Model Compiler Users Manual," Manual Version 1.1, TAMC Version 4.76, 1997.

[21] R. Giering and T. Kaminski, "Recipes for Adjoint Code Construction," Technical Report 212, Max-Planck-Institut for Meterologie.