Calculation of Sensitivity Derivatives in an MDAO Framework

Kenneth T. Moore*
NASA Glenn Research Center, Cleveland, OH

During gradient-based optimization of a system, it is necessary to generate the derivatives of each objective and constraint with respect to each design parameter. If the system is multidisciplinary, it may consist of a set of smaller "components" with some arbitrary data interconnection and process workflow. Analytical derivatives in these components can be used to improve the speed and accuracy of the derivative calculation over a purely numerical calculation; however, a multidisciplinary system may include both components for which derivatives are available and components for which they are not. Three methods to calculate the sensitivity of a mixed multidisciplinary system are presented: the finite difference method, where the derivatives are calculated numerically; the chain rule method, where the derivatives are successively cascaded along the system's network graph; and the analytic method, where the derivatives come from the solution of a linear system of equations. Some improvements to these methods, to accommodate mixed multidisciplinary systems, are also presented; in particular, a new method is introduced to allow existing derivatives to be used inside of finite difference. All three methods are implemented and demonstrated in the open-source MDAO framework OpenMDAO. It was found that there are advantages to each of them depending on the system being solved.

Nomenclature

API	Application Programming Interface
CFD	Computational Fluid Dynamics
CONMIN	Constraint Minimization (an optimizer)
MDA	Multi-Disciplinary Analysis, or an interdependent set of analyses requiring a coupled solution
MDAO	Multi-Disciplinary Analysis and Optimization
SLSQP	Sequential Least SQuares Programming (an optimizer)
XDSM	Extended Design Structure Matrix
u	A local input to a component
v	A local output from a component

I. Introduction

Aerospace design often involves the optimization of large multidisciplinary systems that consist of analyses of varying levels of fidelity with potentially complex interconnections. Some of these analyses may have long run times, particularly when computational fluid dynamics (CFD) or finite element analysis is involved. Design space exploration and optimization of these systems can be challenging because of the high computational cost of each evaluation of the objective function. Optimization methods can generally be divided into two categories: gradient-based methods and gradient-free methods. Gradient-based methods require the evaluation of the gradient of all objectives and constraints with respect to all design variables, while gradient-free methods only require functional evaluation of the objectives and constraints. Optimizers based on genetic algorithms or particle swarm techniques fall into the gradient-free category. CONMIN,[1] SUMT,[2] and SLSQP[3] are all examples of gradient-based optimizers. While genetic optimizers are being

*Senior Consultant, MDAO Branch, Mail Stop 5-11, NASA Glenn Research Center, AIAA Member

used increasingly in conceptual design, the large number of required functional evaluations can still make them prohibitive for systems containing high-fidelity analyses. This leaves gradient-based techniques, which may have their own disadvantages (e.g., no guarantee that the reached minimum is a global one), but the number of function evaluations should be much smaller. Still, for high fidelity analyses, the computational cost can be very high. Much of this cost comes from the calculation of the gradient, which is typically done at each iteration via a forward or backward finite difference initiated by the optimizer for each design variable. In addition to gradient-based optimization, there are a number of other uses for derivatives in multidisciplinary design, including uncertainty quantification, system solution using Newton's methods, and the broader sensitivity analysis methods.

A. Background on Derivatives

Improving the efficiency of the calculation of the gradient is an active area of research, both from the perspective of individual analysis components as well as the full system. In some simple cases, an analysis might be able to provide derivatives (or approximations of the derivatives) of its outputs with respect to its inputs. One example of this occurs in CFD, where the solution of the adjoint equations can be used to calculate the sensitivities of the flow variables.[4] In cases where the source code is available, there are methods that either parse the code line by line to produce a new code that can evaluate the derivatives of its output with respect to its inputs. There are also methods that provide an operator overloading that allows the original code to return its output and the gradient of the output. These methods include the Complex Step Method,[5] Dual Number Automatic Differentiation,[6] and the more general automatic differentiation.[7] In addition, there are analytic methods that solve a linear system of equations.[8] In cases where no analytical or approximate derivatives are available, and where the source code is closed, the fall back solution is once again the finite difference method. Finite difference is disadvantageous; it is an approximation whose accuracy is dependent on the chosen step size, and it is not very efficient, requiring one (or more, for higher order differences) functional evaluation per design variable. However, it is sometimes the only technique available to determine the sensitivity of an analysis.

In multidisciplinary optimization, the gradient for a system of interconnected analysis components must be calculated. Some of the methods listed above can be applied to complete systems. A detailed survey of the techniques that can be used for calculating the derivatives of multidisciplinary systems was made by Martins and Hwang in 2012.[9] In their work, all of the different methods for calculating derivatives are derived from the generalized chain rule equation. Furthermore, they show that the functional approach ($y = F(x)$; $\frac{dy}{dx}$ defined) and the residual approach ($R(y, x) = 0$; $\frac{dR}{dx}$, $\frac{dR}{dy}$ defined) are interchangeable at the component level, regardless of the solution method. This conclusion allows us to express all of our formulations in functional form in this paper.

A frequently occurring feature of multidisciplinary systems is tight coupling, where two or more analysis components are interdependent. For example, component A depends on an output of component B, which in turn depends on an output of component A. This can also be called a feedback loop, a coupled loop, a solver loop, an MDA (Multi-Disciplinary Analysis) loop, an implicit solution, or a set of tightly coupled analyses. The execution of such a system is typically accomplished by breaking the feedback connections and using some solver to iteratively execute the analyses until the residuals on the broken connections are below some tolerance. Fixed-point iterators and Newton solvers are frequently used candidates for this kind of problem. The calculation of the gradient for tightly-coupled components is also a tightly-coupled problem. Sobieszczanski-Sobieski solves the problem by assembling what he calls the *Global Sensitivity Equations (GSE)*,[8] a set of linear equations. These can be solved by linear algebra techniques to produce the total derivatives across that MDA loop. Martins et al. further extends this method by considering aero-structural coupling where the CFD solution provides a set of adjoint equations; a coupled-adjoint solution to the system sensitivity is presented.[10] Efficient gradient solution for coupled multidisciplinary systems is an active area of research. Any method that is used to calculate derivatives of a large coupled system must be flexible to allow the specification of a specific efficient numerical method.

When no individual derivatives are known for any of the analysis components, then the finite difference method can be used to approximate the sensitivities of the objective and the constraints to the design variables. When all of the analysis components provide their derivatives, then a chain rule method or an analytic method can evaluate the complete system sensitivity. However, most real systems are going to be mixed systems – systems where some components can provide derivatives and some cannot. It turns out that any of these methods could be used, but some modifications are needed, and care must be taken to

ensure that the gradient calculation is as efficient as possible. At present, the literature does not seem to cover gradient calculation for mixed systems in detail, so that was a motivation for this work.

B. Frameworks and OpenMDAO

For multidisciplinary systems, the software integration framework seems the logical choice for where to include the gradient calculation capabilities. The framework is already responsible for passing data between analysis components, and it already contains a complete representation of the workflow and the dataflow, which can be used to direct the chain rule evaluation or to assemble the analytic system of equations. The framework can also be used to provide a common documented API through which new optimizers or solvers can query a submodel for its gradient. To investigate the calculation of system sensitivity, the open-source framework OpenMDAO is used.[11] OpenMDAO is a logical choice because it is an active open-source project with publicly hosted code and a community of external developers who contribute code back into the project. Although the methods presented in this paper are implemented in OpenMDAO, they are general enough that they can be implemented in other frameworks.

C. Derivatives of Mixed Systems

As mentioned above, we present three methods – finite difference, chain rule, and analytic methods – to calculate the gradient for mixed systems (systems where some components can provide derivatives and some cannot). While all of the underlying methods have been used in the MDAO field, a new modification to finite difference (called *fake finite difference*) is presented here. This method allows a finite difference calculation to take advantage of any analytic derivatives that are present by replacing the execution with a first-order linear model, which always produces the exact derivative when finite-differenced alone. Both direct (forward) and adjoint (reverse) modes are considered, as is calculation on models with implicit solver loops. Some attention is also given to identifying blocks of components that need to be finite-differenced together because of non-differentiable connections. Finally, three test problems are presented, comparing the relative performance of these three methods.

II. Requirements and Notation

One of the challenges of this work was to develop and implement a set of methods that were not tied to a specific problem (e.g., aero-structural coupling) but instead were general enough to handle an extremely broad range of problems. To define this space, it was helpful to define a set of requirements that an ideal differentiation scheme would satisfy.

A. Requirements

Must support decomposition of the problem.

Decomposition is the process where a large complicated problem is broken down into smaller pieces that can be more easily visualized, classified, and managed. MDAO frameworks use decomposition to allow the user to assemble a problem in a logical manner while taking advantage of reusability. In the OpenMDAO framework, decomposition is supported through the definition of four base classes which are shown in Figure 1. The fundamental building block is the Component, which is a black box that operates on input data to produce output data. The Component can represent a wide range of complexity: anything from a simple equation to a high-order analysis or discipline code such as CFD. The framework also allows the user to define connections that describe how data is passed between the components. These connections define the dataflow. A Driver is a special component that controls the execution of a set of components. Examples of the kinds of processes that Drivers enable include iteration, optimization, systen solution, sensitivity analysis, and design space exploration. Each Driver has a Workflow that contains the execution order of the Components it controls. Finally, Components and Drivers can be assembled into Assemblies, which are sub-models that encapsulate a portion of the full process.

Must be able to calculate the total derivative for any dataflow or workflow.

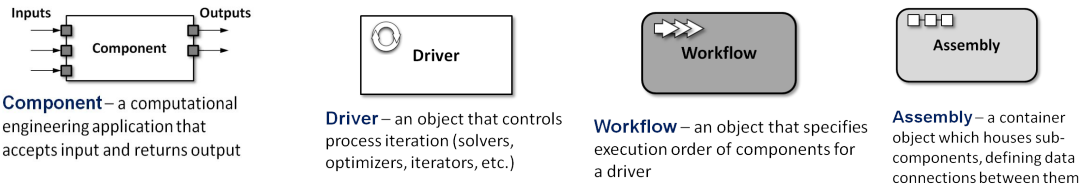

Figure 1: The building blocks of OpenMDAO.

With the inherent ability to combine and nest these basic building blocks, a framework allows the user to define arbitrarily complicated dataflows and workflows. The ideal differentiation scheme should be able to calculate a derivative across any set of input-output pairs without regard to the complexity of either the dataflow or the workflow.

Must have a consistent interface that is unaffected by the system topology.

Quite a few driver processes could make use of derivatives: for example, gradient-based optimization, sensitivity analysis, and Newton iteration. The API for calculating derivatives needs to be consistent, independent of the driver that calls them, and the same structure as the underlying process model.

Must be able to take advantage of any available derivatives.

Sometimes a component is able to calculate its own derivatives. This could be done a number of ways – perhaps the derivatives are computed as a byproduct of the component execution, or maybe they are expressible analytically, or perhaps they are computed from the source code using automatic differentiation or the complex-step method. The ideal differentiation scheme should be able to utilize any of these derivatives in the calculation of the full system sensitivities.

Must be capable of executing in direct (forward) or adjoint (reverse) mode.

For best performance of certain problems, namely those with more design variables than objectives and constraints, adjoint mode is a necessity.

Must be flexible to accommodate unknown future addition or modification to the framework.

Finally, the ideal differentiation scheme should be flexible in the face of future changes to the framework. The three most important non-functional requirements that underlie the development of the OpenMDAO framework are *extensibility*, *flexibility*, and *adaptability*.[12] These terms philosophically describe a framework that is able to extend its capabilities in unanticipated ways and is able to adapt to unknown future requirements. The ideal differentiation scheme is one that is similarly flexible and can handle future features with little or no recoding.

B. Notation

The diagrams that are presented here are essentially network graphs, so some of the notation should be explained. In graph theory, a network graph is a visual realization of the connectivity of a system. A network graph consists of vertices, which represent the objects that are connected, and edges, which are the connections between those objects. For a multidisciplinary system, the vertices are components, and the edges are the data connections. An "n-squared" diagram is more commonly used to show an MDAO dataflow, but we used a more free-form approach here to better illustrate the structure of the graph. XDSM[13] is another format that can show both the workflow and the dataflow in a single graph but was also not used to keep the diagrams simple.

Figure 2 shows an example of the notation that will be used. Solid lines represent data connections between components (the edges in the network graph.) The dashed lines represent the virtual data connections between the model and any driver attributes such as parameters, objectives, and constraints. The distinction is made because a framework's network graph does not generally support circular loops. The boxes represent the decomposition entities: components, drivers, and assemblies. Components are always labeled with a C

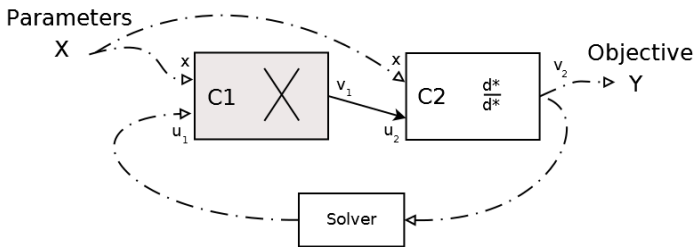

Figure 2: Modified network graphs are used to represent dataflow and the availability of derivatives.

followed by a number. That number is used to subscript its input or output as needed; the subscripts are not shown on the diagrams unless needed for clarity. The right side of a component block specifies the status of its derivatives. If it contains a large X, then the component cannot provide any derivatives. Otherwise, $\frac{d*}{d*}$ indicates that it can provide all of its local derivatives. To make it more visually apparent, components that cannot provide derivatives are also shaded a light gray. The evaluated objective and constraint equations or inequalities are defined as Y, and the input design parameters are defined as X, or sometimes Z.

III. Finite Difference

The first differentiation method that will be examined is the finite difference method. Finite difference is an approximation of the derivative that comes from the truncation and rearrangement of a Taylor's series expansion. For first-order derivatives, the most commonly used types are the forward, backward, and central differences, given in equations 1, 2, and 3 respectively.

$$\frac{df}{dx} = \frac{f(x+\Delta x) - f(x)}{\Delta x} + \mathcal{O}(\Delta x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{1}$$

$$\frac{df}{dx} = \frac{f(x) - f(x-\Delta x)}{\Delta x} + \mathcal{O}(\Delta x) \approx \frac{f(x) - f(x-\Delta x)}{\Delta x} \tag{2}$$

$$\frac{df}{dx} = \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} + \mathcal{O}(\Delta x^2) \approx \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} \tag{3}$$

Here Δx is a step size that must be carefully chosen. If it is too large, then the estimate of the derivative will be inaccurate. If it is too small, numerical underflow will become a problem as it approaches the computational hardware precision. Also, precision may be limited on some of the tools. For example, the finite-element solver Nastran only outputs stresses and displacements to six significant digits, which puts an even tighter lower limit on the step size that can be used. All of the difference formulas give an approximation for the derivative at an initial point. The forward and backward differences require a single function evaluation. The central difference formula is a more accurate approximation, but it requires two functional evaluations instead of one.

The finite difference method is the easiest of the differentiation methods to implement in a framework because it does not require any interaction with the internals of a model. If a model can execute in an MDAO framework, it can be finite-differenced. This also means that the finite difference method is quite flexible and will be able to accommodate future additions to the framework without modification.

A. Adjoint Mode

One of the limitations of finite difference is that it does not support adjoint mode calculation. A reverse-direction finite difference is theoretically possible if the components are executable in reverse (i.e, operate on their local outputs to produce their local inputs), but there are few cases where that can happen.

B. Using Analytical Derivatives in a Finite Difference Calculation

Another limitation of finite difference is that, out of the box, it has no support for analytical derivatives. To remedy this, we propose a modification to the finite difference method that we are calling *fake finite*

difference.

Consider the simple multidisciplinary system pictured in Fig. 3, containing three discipline components, a single parameter x, an objective or constraint equation y, and two connecting variables represented by the line between v_1 and u_2, and the line between v_2 and u_3. Each of the component interconnections may include a simple equation and a unit conversion factor, though in the simplest of cases, $u_{n+1} = v_n$, and the derivative is just 1.0. We would like to determine the total derivative of the objective y to the parameter x.

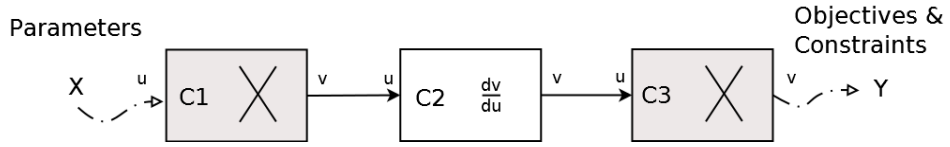

Figure 3: Three component sequence – only Component 2 has analytical derivatives.

If none of the components can provide analytical derivatives, then clearly the derivative must be calculated numerically via a finite difference approximation, either individually or monolithically. If all of the components can provide analytical derivatives, then methods are available, for example the chain rule:

$$\frac{dy}{dx} = \frac{\partial v_1}{\partial u_1}\frac{\partial u_2}{\partial v_1}\frac{\partial v_2}{\partial u_2}\frac{\partial u_3}{\partial v_2}\frac{\partial v_3}{\partial u_3}\frac{\partial y}{\partial v_3} \tag{4}$$

The chain rule can accommodate a mixed system because a local finite difference can be performed on any component where it is needed. But is there an analogous technique for a global finite difference? Instead of using local numerical solutions to augment a global analytical solution, we would like to use a local analytical solution to improve a global numerical solution. It turns out that this can be accomplished by taking a component that has derivatives and replacing its output with a function that always returns the exact derivative when differenced. So, during a global finite difference step, if a component cannot provide derivatives, it runs as normal. If it can, then instead of executing, it outputs the value of this function evaluated at its inputs. This function is constructed from the first-order linear model using its known analytical derivative. In equation form:

$$v = v^0 + \frac{dv}{du}(u - u^0) \tag{5}$$

We have tentatively given this method the name fake finite difference, because a component with derivatives can "fake" its output. The output of this linear model may only be correct at the starting point of the finite difference, but its derivative is always exact. To prove this, consider a single component:

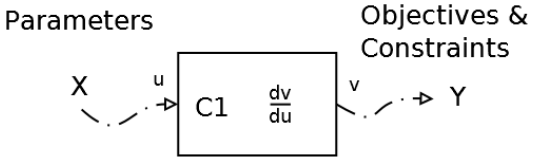

Figure 4: A workflow with a single component.

By inspection, we know that the local derivative $\frac{dv}{du}$ that the component provides is also the total derivative that we seek. Let's perform fake finite difference by first replacing our component execution with the linear model:

$$v = v^0 + \frac{dv}{du}(u - u^0) \tag{6}$$

At the top level, we formulate the finite difference approximation for the global derivative. Here we use a central difference, though other forms are also valid:

$$\frac{dy}{dx} = \frac{dv}{du} \approx \frac{v^+ - v^-}{u^+ - u^-} \tag{7}$$

Now, substitute Eq. 6 into Eq. 7 and simplify:

$$\frac{dv}{du} \approx \frac{v^0 + \frac{dv}{du}(u^+ - u^0) - v^0 - \frac{dv}{du}(u^- - u^0)}{u^+ - u^-} \tag{8}$$

$$\frac{dv}{du} \approx \frac{v^0 - v^0 + \frac{dv}{du}(u^+ - u^0 + u^0 - u^-)}{u^+ - u^-} \tag{9}$$

$$\frac{dv}{du} \approx \frac{dv}{du} \tag{10}$$

The final simplification returns the component's local derivative, as expected. Fake finite difference has allowed us to improve the finite difference solution using the analytical derivative, and has returned the exact solution instead of an approximation.

Note that the result is independent of the step size Δx used in the finite difference. This is a very nice feature of the method. However, if this is part of a larger model, and derivatives are not available for everything, then the stepsize will still need to be chosen with care.

IV. Chain Rule

The next differentiation method that will be examined is the chain rule method. One of the basic tenets of calculus, the chain rule states that the derivative of the composition of functions is just the multiplicative chain of the derivatives of the functions. This can be applied to an MDAO system where each component is considered to be a functional composition. Consider the following example:

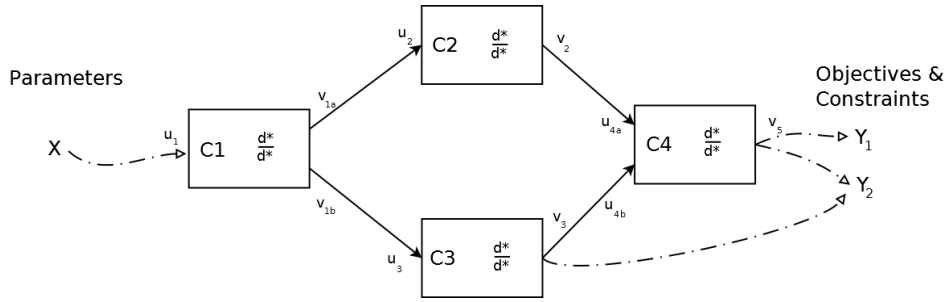

Figure 5: An MDAO dataflow. Derivatives are cascaded by following the network graph.

While this is a simple example, it exhibits some features that can be found in an MDAO dataflow, such as the splitting and merging of the data paths. Each component is numbered (C1 through C4) and has its own input and output. Subscripted letters will be used to indicate the branching for components with multiple outputs. To find the derivatives of the objective (Y_1) and constraint (Y_2) with respect to the design parameters (X),we can start at the design parameter and successively apply the chain rule from left to right. The first component gives us one equation for each output.

$$\frac{dv_{1a}}{dx} = \frac{\partial v_{1a}}{\partial u_1} \tag{11}$$

$$\frac{dv_{1b}}{dx} = \frac{\partial v_{1b}}{\partial u_1} \tag{12}$$

This gives us the derivative of the output of component C1 with respect to the design variable. We now take those derivatives and chain them through the next components in the graph:

$$\frac{dv_2}{dx} = \frac{\partial v_2}{\partial u_2}\frac{dv_{1a}}{dx} \tag{13}$$

$$\frac{dv_3}{dx} = \frac{\partial v_3}{\partial u_3}\frac{dv_{1b}}{dx} \tag{14}$$

Component C4 has two inputs. The chain rule extends to multivariate functions as:

$$\frac{dF(y,z)}{dx} = \frac{\partial F}{\partial y}\frac{dy}{dx} + \frac{\partial F}{\partial z}\frac{dz}{dx} \tag{15}$$

So, the total derivative at the output of C4 is:

$$\frac{dv_4}{dx} = \frac{\partial v_4}{\partial u_{4a}}\frac{dv_2}{dx} + \frac{\partial v_4}{\partial u_{4b}}\frac{dv_3}{dx} \tag{16}$$

Finally, the objective and constraints depend on some of the model outputs, so we complete the chain using the partial derivatives found in their equations.

$$\frac{dy_1}{dx} = \frac{\partial y}{\partial v_4}\frac{dv_4}{dx} \tag{17}$$

$$\frac{dy_2}{dx} = \frac{\partial y}{\partial v_4}\frac{dv_4}{dx} + \frac{\partial y}{\partial v_3}\frac{dv_3}{dx} \tag{18}$$

The process we performed is also known as automatic (or algorithmic) differentiation applied in forward mode.

The chain rule method can be implemented in an MDAO framework, although it is a much more difficult task than implementing the finite difference method. One thing that helps the implementation is that a framework typically uses a network graph to determine execution order of the components; this is particularly important to support complex dataflows. This same network graph can be queried to find the correct order for processing the chain and querying the component derivatives. It can also be used to first find the set of interior edges so that no extra derivative (e.g., an unconnected output) is processed. OpenMDAO uses the NetworkX[14] Python package, which is developed and hosted by Los Alamos National Laboratory.

However, even with taking advantage of the network graph, the chain rule is a challenge to implement in a general integration framework such as OpenMDAO. Unlike finite difference, the chain rule method must address and properly handle every feature of the framework. If the framework supports assemblies, then the algorithm must be able to recurse. If the data connections contain unit conversion or allow the specification of an expression, then their derivatives must be chained. A particular challenge is OpenMDAO's iteration hierarchy, which allows an arbitrary nesting of drivers including optimizers, solvers, and iterators. There may also be new types of drivers that have not been envisioned, and how they function under the chain rule will need to be addressed when they are implemented. This is not to say that it cannot be done; it is only an attempt to qualify the relative difficulty. The chain rule has been implemented in OpenMDAO, but some features (e.g., nested optimizers) are not supported yet.

A. Adjoint Mode

The chain rule method also supports adjoint mode evaluation, which can be performed by starting at the outputs and applying the chain rule in reverse (i.e., chaining the denominator instead of the numerator.) To illustrate, let's take the same dataflow from Figure 5 and apply the chain rule in reverse. For the objective:

$$\frac{dy_1}{du_{4a}} = \frac{\partial v_4}{\partial u_{4a}}\frac{\partial y_1}{\partial v_4} \tag{19}$$

$$\frac{dy_1}{du_{4b}} = \frac{\partial v_4}{\partial u_{4b}}\frac{\partial y_1}{\partial v_4} \tag{20}$$

Next, proceed along the split paths.

$$\frac{dy_1}{du_2} = \frac{\partial v_2}{\partial u_2}\frac{dy_1}{du_{4a}} \tag{21}$$

$$\frac{dy_1}{du_3} = \frac{\partial v_3}{\partial u_3}\frac{dy_1}{du_{4b}} \tag{22}$$

And finally:

$$\frac{dy_1}{dx} = \frac{\partial v_{1a}}{\partial u_1}\frac{dy_1}{du_2} + \frac{\partial v_{1b}}{\partial u_1}\frac{dy_1}{du_3} \tag{23}$$

In a framework, we can reverse the direction of the model's network graph to obtain the correctly sorted execution order for adjoint derivative calculation. This is also known as reverse mode automatic differentiation. Here, we only solved for the objective, but we could also solve for the constraint by repeating

the process for that equation. For this example, the forward mode is more efficient than the adjoint mode because it only requires one sweep as opposed to two. Adjoint mode is more efficient when the number of constraints plus objectives is greater than the number of design parameters.

V. Analytic Solution

The final differentiation method to be examined is the analytic solution method. Instead of solving for the derivatives at the output of each component sequentially, we can assemble the equations and solve for them simultaneously. Consider the same simple example from Section IV, shown in Figure 6.

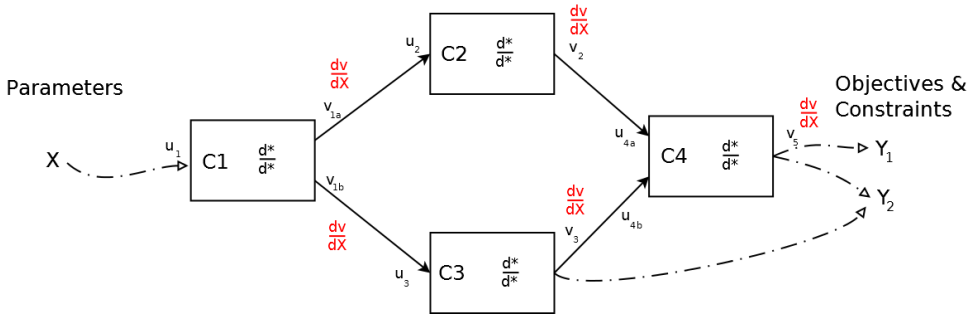

Figure 6: With analytic solution, all unknown derivatives are solved for simultaneously.

We can assemble the following linear system to solve for the derivatives at each component output.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ -\frac{\partial v_2}{\partial u_2} & 0 & 1 & 0 & 0 \\ 0 & -\frac{\partial v_3}{\partial u_3} & 0 & 1 & 0 \\ 0 & 0 & -\frac{\partial v_4}{\partial u_{4a}} & -\frac{\partial v_4}{\partial u_{4b}} & 1 \end{bmatrix} \begin{bmatrix} \frac{dv_{1a}}{dx} \\ \frac{dv_{1b}}{dx} \\ \frac{dv_2}{dx} \\ \frac{dv_3}{dx} \\ \frac{dv_4}{dx} \end{bmatrix} = \begin{bmatrix} \frac{\partial v_{1a}}{\partial u_1} \\ \frac{\partial v_{1b}}{\partial u_1} \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{24}$$

This system can be solved using available linear algebra methods. Notice that the system matrix is lower triangular. Any system that does not contain reverse connections (i.e, no solver loops or MDAs) will always be triangular. In fact, the equations that we solved for in this problem using the chain rule method can be generated by taking this system of equations and solving it by hand one line at a time. Thus the analytic solution is a more generalized representation than the chain rule method. It is possible to solve it in an equivalent manner (back substitution), but it is also possible to use other solutions (e.g., Cholesky decomposition, Gauss–Seidel, various sparse techniques), which will be important for systems with a large number of coupling variables. So it may appear that the chain rule method is functionally superfluous; however, we will see from the examples in Section IX that there are some cases for which the chain rule is a better choice than the analytic solution.

A. Adjoint Mode

Analytic methods can also be used to solve the adjoint linear system. As with the chain rule, we want to solve for the derivative of the objective and constraints with respect to all the component outputs. Consider the same system in Figure 7.

We can assemble the following adjoint linear system to solve for the needed derivatives.

$$\begin{bmatrix} 1 & 0 & -\frac{\partial v_2}{\partial u_2} & 0 & 0 \\ 0 & 1 & 0 & -\frac{\partial v_3}{\partial u_3} & 0 \\ 0 & 0 & 1 & 0 & -\frac{\partial v_4}{\partial u_{4a}} \\ 0 & 0 & 0 & 1 & -\frac{\partial v_4}{\partial u_{4b}} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{dy_1}{dv_{1a}} & \frac{dy_2}{dv_{1a}} \\ \frac{dy_1}{dv_{1b}} & \frac{dy_2}{dv_{1b}} \\ \frac{dy_1}{dv_2} & \frac{dy_2}{dv_2} \\ \frac{dy_1}{dv_3} & \frac{dy_2}{dv_3} \\ \frac{dy_1}{dv_4} & \frac{dy_2}{dv_4} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & \frac{\partial y_2}{\partial v_3} \\ \frac{\partial y_1}{\partial v_4} & \frac{\partial y_2}{\partial v_4} \end{bmatrix} \tag{25}$$

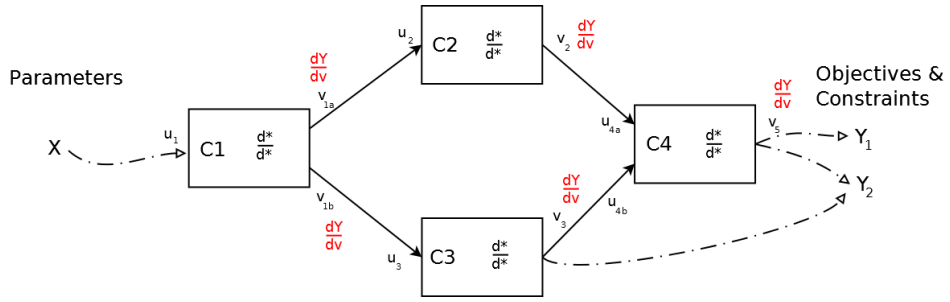

Figure 7: Simultaneous solution of the adjoint system.

Notice that the vector of unknowns has two columns: one for the objective and one for the constraint equation. As mentioned earlier, the direct analytic solution is computationally more efficient for this problem because the number of objectives and constraints is greater than the number of design parameters. The adjoint system is upper triangular when there is no feedback coupling in the system, and in fact, the direct and adjoint systems are related. The direct system is of the following form:

$$Ax = b \tag{26}$$

$$y = d + cx \tag{27}$$

while the adjoint system is an equivalent rearrangement:

$$A^T z = c^T \tag{28}$$

$$y = d + z^T b \tag{29}$$

The desired quantity from each of these equation sets is y, which is the matrix of derivatives of the objective and constraints with respect to the design variables, and is the same regardless of which method is chosen. Thus, the same system matrices can be assembled for both direct and adjoint modes, and the appropriate problem can be chosen afterwards.

VI. Behavior for a Multidisciplinary Analysis (MDA) Loop

So far we have covered differentiation of systems whose state can be solved explicitly, but we also need to see how they work for systems that require an implicit solution for some of the variables. Consider the case pictured in Figure 8. A framework such as OpenMDAO does not allow a circular data dependency to be declared in a model, so one (or more) of the connections in the loop must be broken. This allows an iterative solver to drive the convergence of those variables using such techniques as fixed-point iteration, quasi-Newton methods with Broyden update, or full Newton-Raphson solution. This is a typical feature of a multidisciplinary analysis, so it is important for all differentiators to be able to calculate derivatives of a model containing an MDA loop.

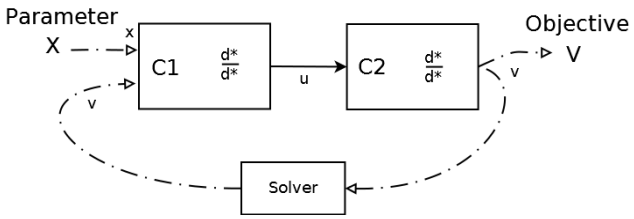

Figure 8: A simple set of coupled components.

A. MDAs under Analytic Solution

The analytic differentiator is well-suited for solving the derivatives of a model that contains a solver loop because it solves for all intermediate derivatives at once. Consider the model pictured in Figure 8. If we

remove the solver and reconnect output v of component C2 to input v of component C1, and assemble the analytic equations, we get:

$$\begin{bmatrix} 1 & -\frac{\partial u}{\partial v} \\ -\frac{\partial v}{\partial u} & 1 \end{bmatrix} \begin{bmatrix} \frac{du}{dx} \\ \frac{dv}{dx} \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial x} \\ 0 \end{bmatrix} \tag{30}$$

Now, instead of a triangular matrix, we have a full matrix where the upper triangular portion (or the lower triangular portion in adjoint mode) contains the derivatives of all the connections that were broken to insert the solver. This system of equations can be solved for the system containing an MDA; successful examples are in the literature.[10]

B. MDAs under Chain Rule

The chain rule algorithm as presented above cannot handle an MDA because the sequential operation in either the direct or adjoint directions eventually reaches a point where it needs a derivative that has not been calculated yet. However, there is an improvement to the method that can enable a solution. Consider how the core problem – we require information from further ahead in the chain – mirrors the same problem in the dataflow which was solved by inserting an iterative solver. This problem has also been solved. Gilbert[15] investigated automatic differentiation applied to a problem with an implicit solver (in his case, fixed-point iteration) and showed that if the implicit solution of the process converges, then the implicit solution of the derivative of that process will also converge. This means that we could modify the chain rule method to iteratively converge the derivatives for the same connections that the solver iterates.

This implicit chain rule solution has not been implemented in OpenMDAO, primarily because at the time of implementation, there was not a clear advantage to using chain rule over the analytic solution method. For the test problems in Section IX, the chain rule is omitted in the problems that have solver connections.

C. MDAs under Fake Finite Difference

Finally, we need to investigate how fake finite difference handles a solver loop. There is no question that plain finite difference can handle an MDA as well as it handles anything else, provided the stepsize is chosen appropriately. Once the algorithm increments a design parameter and the model is executed, the MDA converges to a new value that is consistent with the state of the incremented system. However, the fake finite difference modification replaces some or all of the components in that solver loop with a linear model. So there is a question of how this affects the convergence of the MDA during solution of the incremented points.

To investigate this, lets take the simple coupled system in Figure 8 and form the first-order linear models that we need for fake finite difference.

$$u = u^0 + \frac{\partial u}{\partial v}(v - v^0) + \frac{\partial u}{\partial x}(x - x^0) \tag{31}$$

$$v = v^0 + \frac{\partial v}{\partial u}(u - u^0) \tag{32}$$

This is a linear system of equations that depends on the coupling variables offset from their initial state, which is the baseline point around which we intend to finite-difference. If a solver were used to converge this system, it should arrive at the solution that satisfies this set of equations, which can be found by solving:

$$\begin{bmatrix} 1 & -\frac{\partial u}{\partial v} \\ -\frac{\partial v}{\partial u} & 1 \end{bmatrix} \begin{bmatrix} u - u^0 \\ v - v^0 \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial x}(x - x^0) \\ 0 \end{bmatrix} \tag{33}$$

This set of equations should look familiar: it's the analytical solution to the coupled derivatives for this problem shown in Equation 30. The only difference is the $(x - x^0)$, though recall that automatic differentiation is seeded by a 1.0 at the initial input. The same set of equations that describes the coupled derivatives also describes the coupled system when the components are replaced by linear models in fake finite difference. The difference here is that an MDA will solve these implicitly rather than using a matrix solution.

Since Equation 33 is a 2x2 system, the solution is known. We are interested in the value of v:

$$v = v^0 + \frac{\frac{\partial v}{\partial u}\frac{\partial u}{\partial x}(x - x^0)}{1 + \frac{\partial v}{\partial u}\frac{\partial u}{\partial v}} \tag{34}$$

We can also solve for the derivative $\frac{dv}{dx}$ analytically using Equation 30.

$$\frac{dv}{dx} = \frac{\frac{\partial v}{\partial u}\frac{\partial u}{\partial x}}{1 + \frac{\partial v}{\partial u}\frac{\partial u}{\partial v}} \tag{35}$$

This is the exact derivative for the system valid at the point at which the MDA was converged. We can see how the linear models perform by taking the output equation (Equation 34) and substituting it into the forward difference equation (Equation 1) to simulate a finite difference.

$$\frac{dv}{dx} \approx \frac{v^0 + \frac{\frac{\partial v}{\partial u}\frac{\partial u}{\partial x}(x - x^0)}{1 + \frac{\partial v}{\partial u}\frac{\partial u}{\partial v}} - v^0}{(x - x^0)} \tag{36}$$

$$\frac{dv}{dx} \approx \frac{\frac{\partial v}{\partial u}\frac{\partial u}{\partial x}}{1 + \frac{\partial v}{\partial u}\frac{\partial u}{\partial v}} \tag{37}$$

Comparing to Equation 35, finite difference has given us the exact analytical value for this set of coupled components, so we have shown that fake finite difference can also handle calculating the derivative across a system with a solver. Note again how the stepsize does not appear in the final equation. It is interesting to compare this method with the implicit chain rule solution. The MDA on the linearized models solves the same equations as are solved in the implicit chain rule, so it seems reasonable that its convergence properties would hold true for fake finite difference as well.

VII. Identification of Finite-Difference Blocks

For both the chain rule and analytic methods, it turns out that it's not enough to singly identify and finite difference the components that don't have derivatives. So far, we have treated every connection as a differentiable one, but a model can also have non differentiable connections. For example, a component may pass a data file containing some Fortran-formatted input to another component. Other examples of non differentiable components include integers, enumerated variables, booleans, and strings. The chain rule and analytic differentiators cannot operate on variables of these types because it is not possible to define a derivative for them. In order to process them, components that pass non differentiable data must be finite-differenced together. Thus, an additional step that identifies these blocks is necessary.

These blocks can be identified by finding all edges in the network graph that contain non differentiable variables and merging their connected nodes (components) until all of those edges are removed. However, this can lead to the situation shown in Figure 9, where a differentiable block is surrounded by a group of components that must be finite differenced together. Here, component C2 has derivatives, but the block that contains components (C1, C3, C4) surrounds it, or feeds it with an input and depends on its output.

We have created a coupled system and it could be resolved by breaking one of the connections and inserting a solver while finite-differencing the block. However, convergence with a solver would require more executions of components (C1, C3, C4) and would likely be slower than just finite differencing the whole model. The only alternative is to group component C2 in with Components C1, C3, and C4 to form a larger finite difference block. Thus, the analytical derivatives that component C2 provides will never get used. But if we use the fake finite difference trick introduced in Section III for all of our finite-difference blocks, then the analytic derivatives for any differentiable island can be utilized when this block is finite-differenced.

One further consideration is to limit the total number of finite difference blocks to simplify the problem solution. If two components are in series, and neither has derivatives, then grouping them into a finite-difference block reduces the number of steps in the chain rule calculation, or the number of variables in the analytic solution, so it is reasonable to do this. This provides the additional benefit of having fewer inputs for which a reasonable stepsize must be determined.

To summarize, we have the following rules for finding groups of components that need to be finite-differenced together:

1. **Components with non-differentiable connections must be grouped**

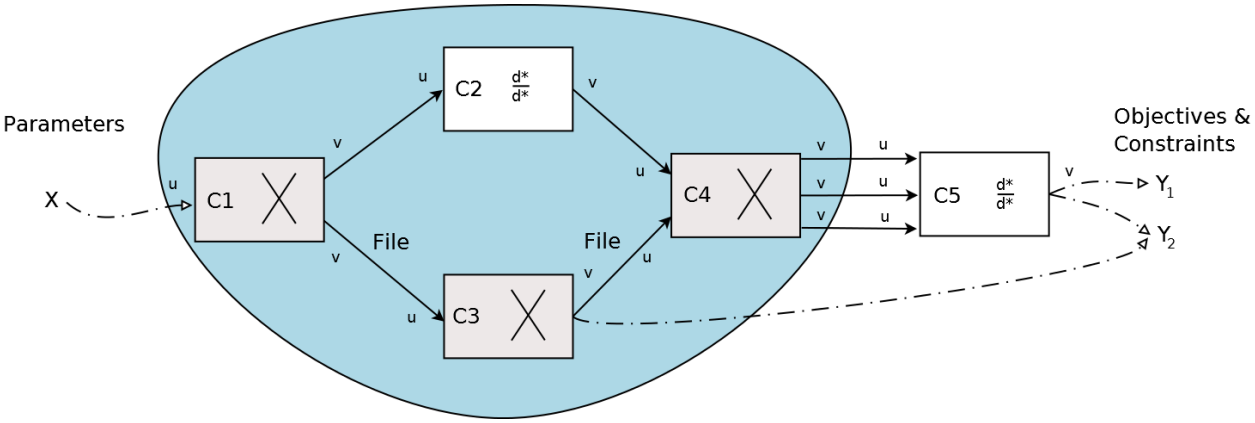

Figure 9: A differentiable component surrounded by non-differentiable components must be finite differenced with them.

2. **Differentiable islands are not permitted**

3. **Otherwise group to reduce the total number of finite differences**

These rules seem reasonable, but they are currently untested. Presently, OpenMDAO does not perform any automatic grouping of finite-difference blocks for either the chain rule or the analytic differentiators, though there is support for specifying the groupings manually. Otherwise, each component that needs it is finite differenced individually (and of course, the finite difference differentiator treats the whole system as one block). This is an area that will be investigated more completely in the future.

VIII. Comparison of the Methods

Table 1 summarizes the characteristics of the three differentiation methods.

Table 1: Summary of differentiation methods

	Finite Difference	Chain Rule	Analytic
Analytic Derivatives	Yes (with fake FD)	Yes	Yes
Solves MDAs	Yes	Yes (with Mod)	Yes
Forward/Adjoint	Yes/No	Yes/Yes	Yes/Yes
Implementation	Easy	Difficult	Difficult
Flexibility	Should Always Work	Some Additional Work	Some Additional Work

All of them are capable of utilizing analytic derivatives, and all of them can solve an MDA, although we have not investigated and implemented this for the chain rule differentiator. There is no adjoint mode for finite difference, but the other differentiation methods can handle it fine. Implementing the finite difference method (including the fake finite difference modification) in an MDAO framework is relatively easy, but implementing the chain rule and analytic differentiators is difficult and requires an understanding of how the framework operates internally. Every feature (e.g., driver type, unit conversion, etc.) has to be explicitly addressed, so it is expected that any new features added to the framework will require retooling of these two differentiators, depending on the nature of the addition. For example, a recent update to OpenMDAO added support for user-specification of a simple equation when connecting an output to an input. This provides a shortcut – the user can specify the equation instead of creating and inserting a new component between the two connected components. However, this expression is handled during the information passing, so it does not show up in the framework's network graph. Thus it was necessary to modify the chain rule and the analytic differentiators to handle these appropriately. The finite difference differentiator required no modification.

IX. OpenMDAO Examples

All three of these methods have been implemented in OpenMDAO as Differentiators. A Differentiator is an object that can be plugged into any OpenMDAO Driver that requires derivatives. A simple API was specified so that all of the Differentiators are interchangeable.

To compare their performance, we will show and discuss three simple test problems. The components in the problems are defined by equations which can be differentiated and for which the correct analytical solution of the system and the system's derivative can be computed by hand. In order to evaluate how mixed systems perform, for each of the problems we designate some components as ones that can supply analytical derivatives and some as ones that cannot. Since the components are all simple equations, all of the examples run very quickly. To simulate the performance of components with longer run times, a delay of 1.0 seconds was added to each function execution, and a delay of 0.3 seconds was added to each component's local derivative calculation. In OpenMDAO parlance, the *execute* method gets a 1.0 second delay, and the *calculate_first_derivatives* method gets a 0.3 second delay. This will also serve to swamp out any performance differences that are due to framework overhead, since at this time, OpenMDAO's performance has not been completely optimized.

A. Multi-feature Example

The first test case (Figure 10) involves differentiation of a dataflow that includes several of OpenMDAO's features. The model contains five components, three of which are in a nested assembly. Variables are passed into and out of an assembly through intermediate connections on the assembly boundary, as indicated by the small circles in the figure. This particular dataflow was chosen because it has some interesting features – splitting, joining, multiple variable connections – to make sure that the differentiators were properly traversing the network graph. The equations for the components are listed in Table 2. In addition, an expression connection $u = 2v$ was defined between the output of the assembly and the first input of component C5. Note that only components C2 and C5 have analytical derivatives defined in this problem.

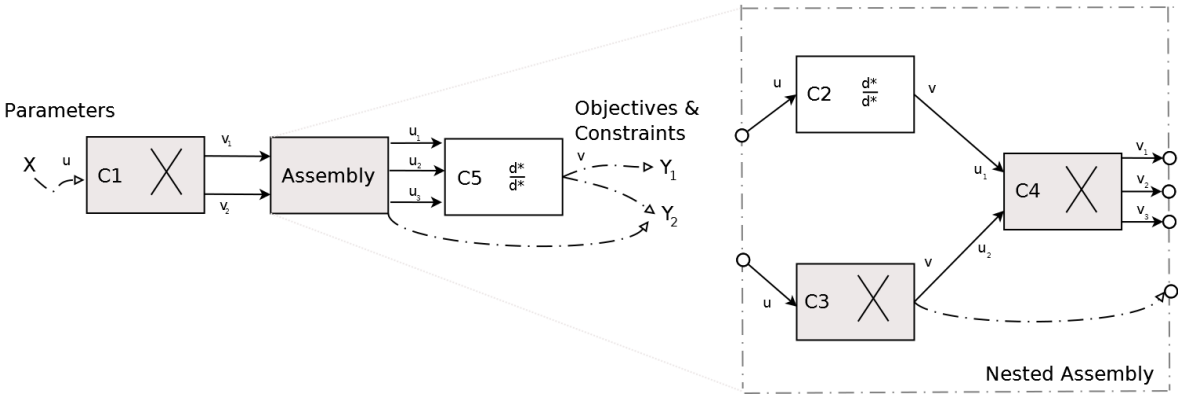

Figure 10: Case A dataflow contains a nested assembly.

Table 2: Component Definitions for Case A

C1	C2	C3	C4	C5
$v_1 = 2u^2$ $v_2 = 3u$	$v = \frac{1}{2}u$	$v = 3.5u$	$v_1 = u_1 + 2u_2$ $v_2 = 3u_1$ $v_3 = u_1 + u_2$	$v = u_1 + 3u_2 + 2u_3$

This problem was executed using all three differentiators under all available operating modes, and the performance of each of them is outlined in Table 3. All of the differentiators returned the correct derivative values with eight decimal places of accuracy.

The first column is for the plain finite difference differentiator, so the analytic derivative functions on components C2 and C5 are never called. Each component executes three times: once for the initial run and two additional times for a full-model central finite difference. As expected, this is the slowest differentiator at

10 seconds. The next column is finite difference with the fake finite difference improvement, so the analytic derivatives on components C2 and C5 are used to replace the function execution during the finite difference. Thus, components C2 and C5 only execute once when the model is first run. The function containing the analytic derivative expression runs once during the finite difference to create the linear model that is used in both the positive and negative deltas of the design parameter. Note the execution time of 6.62 seconds is a nice improvement over pure finite difference.

Table 3: Performance of the differentiators in OpenMDAO for Case A

	Finite Difference	Finite Difference (with Fake FD)	Chain Rule	Analytic (Direct)	Analytic (Adjoint)
Funct Evals	3,3,3,3,3	3,1,3,3,1	3,1,3,5,1	3,1,3,5,1	3,1,3,5,1
Deriv Evals	0,0,0,0,0	0,1,0,0,1	0,1,0,0,1	0,1,0,0,1	0,1,0,0,1
Time (sec)	10.02	6.62	8.90	8.70	8.70

The next columns show the chain rule differentiator as well as the analytic differentiator running in direct and adjoint mode. The analytic solution appears to be slightly faster; however, both of them are slower than fake finite difference on this problem. The component execution reveals the reason: component C5 is executed five times because it needs two local finite differences to calculate its derivatives with respect to its two inputs. This is unfortunate, because these derivatives are necessary. It is easy to envision a case where a component has so many local inputs that finite differencing all of them is slower than performing finite difference on the whole system. However, there is a workaround. Consider that in this case, we only have one design variable, so we should not care about the full richness of the input space of component C5. If we could map the inputs of that component onto a single input that is *most strongly influenced by the design parameter x*, then we would only need one finite difference of C5 per design variable. The linear combination of local inputs that accomplishes this is the set of sensitivities of those inputs to the design variable. If we are using the chain rule differentiator in the direct mode, then we already have those quantities and can perform the projection. Unfortunately, when using the analytic differentiator, these quantities are not available until after the solution. This may be the one advantage of the chain rule differentiator, at least for workflows without a solver loop. This input projection method has not been implemented in OpenMDAO, and addressing this problem for both of these differentiators is an area for future research.

B. Simple MDA

The second test problem (Figure 11) is an MDA loop containing two components where the goal is to solve for the derivative of the coupled system. The equations for both components are listed in Table 4.[16] Note that component C2 is implemented with an analytic expression for the derivatives. The Broyden solver, which is a Newton-Raphson solver with Broyden approximation to the Jacobian, is used to converge the MDA loop.

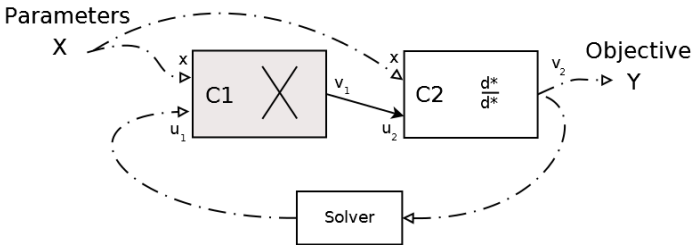

Figure 11: Case B is a simple coupled problem.

Table 4: Component Definitions for Case B

C1	C2
$v_1 = \frac{1}{2}sin(x) - \frac{1}{2}xu_1$	$v_2 = x^2u_2$

This problem was solved using only the finite difference and analytic differentiators because the chain rule differentiator in OpenMDAO currently cannot iteratively solve the coupled system. The performance of the differentiators is outlined in Table 5. All of the differentiators achieved the correct value for the derivative between x and y to 7 decimal places of accuracy.

Table 5: Performance of the differentiators in OpenMDAO for Case B

	Finite Difference	Finite Difference (with Fake FD)	Analytic (Direct)	Analytic (Adjoint)
Funct Evals	14, 15	14, 3	7, 3	7, 3
Deriv Evals	0, 0	0, 1	0, 1	0, 1
Time (sec)	23.04	11.33	4.48	4.37

Once again, pure finite difference is the slowest method, but the difference is more notable here because the solver must converge the MDA loop two times (for central difference) on the slowly executing components. Fake finite difference again fares much better because only the component without derivatives is executing under the extra MDA loops. However, this time the analytic solution is the clear winner. The explicit solution requires less execution of the components than the implicit solver. Note that the first 3 function evaluations are due to the initial solution of the MDA, so the analytic differentiator only adds 4 executions of C1 to evaluate the local derivatives of its two input variables. Also, the difference between the direct and adjoint methods is small and may not have any significance since neither should have an advantage on a model with these dimensions.

C. MDF Architecture with the Sellar Problem

The final test problem (Figure 12) is an optimization problem using the MDF (Multidisciplinary Design Feasible) architecture. The model to be optimized is the two-discipline Sellar problem for which a derivative function is available on the second discipline. The dataflow of this model is somewhat similar to the second test problem, although it introduces two additional global design variables to the local design variable in the first component, and adds a couple of constraints. Also, instead of singly mimicking the step where an optimizer asks for computation of the system's derivative, this test performs the entire optimization to convergence. This should give us some insight into how the differentiators will perform on a real case. The equations for the Sellar problem are listed in Table 6

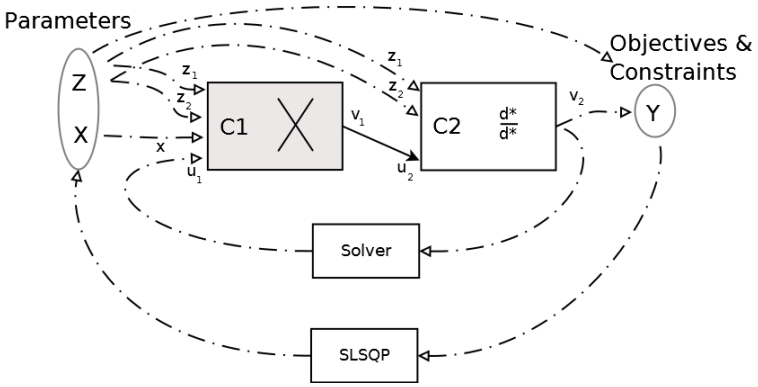

Figure 12: Case C is the optimization of the Sellar problem using MDF.

Table 6: Problem Definitions for Case C

C1	C2	Objective	Constraint	Constraint
$v_1 = z_1^2 + z_2 + x - 0.2u_1$	$v_2 = \sqrt{u_2} + z_1 + z_2$	Minimize $x^2 + z_2 + v_1 + e^{-v_2}$	$v_1 > 3.16$	$v_2 < 24.0$

Once again, this problem was solved using only the finite difference and analytic differentiators because the chain rule differentiator in OpenMDAO currently cannot iteratively solve the coupled system. The

performance of the differentiators is outlined in Table 7. For each of the differentiators, the optimizer was able to reach essentially the same minimum for the objective (3.18339) with differences only arising in the 6th decimal place. The SLSQP optimizer required no changes from the default settings for any of the cases.

Table 7: Performance of the differentiators in OpenMDAO for Case C

	Finite Difference	Finite Difference (with Fake FD)	Analytic (Direct)	Analytic (Adjoint)
Funct Evals	139, 139	139, 25	73, 25	73, 25
Deriv Evals	0, 0	0, 6	0, 6	0, 6
Time (sec)	278.5	166.1	100.2	100.0

As we expected, finite difference is the slowest of the methods, requiring 139 function evaluations of each of the disciplines over the entire optimization. Turning on fake finite difference offers a nice improvement because it reduces the number of executions of component C2. The analytic solution is again the best performer here because it does not require a iterative solution of the MDA to find the coupled derivatives. The numbers reveal that the optimizer evaluated 25 different design points and queried the gradient 6 times, resulting in 48 functional evaluations, since there are four inputs to C1. Also, during a system finite difference, the Broyden solver took an average of just over two iterations to converge. Since the number of design variables matched the number of objectives and constraints, neither of the direct or adjoint methods holds an advantage over the other.

X. Conclusions and Further Work

In this paper, we have described and implemented three methods for evaluating the derivatives of a model in an MDAO framework, with emphasis on systems that simultaneously include components that can provide their derivatives and components that cannot. The implementation in OpenMDAO provided some new insights about each differentiation method. We originally thought that one of the three methods would be the clear winner and that we could whittle OpenMDAO's collection of differentiators down to just a single one, but it turned out that there are uses for each of them.

The analytic method seems to be the best performing of the three methods. It is the fastest, and is one of the two methods that can handle adjoints. It is one of the more challenging algorithms to implement in a general framework, and since it is not very robust to future improvements to a framework, it will require some new code. However, it probably deserves to be your first choice if you want to implement just one of these.

The chain rule method performed nearly as well as the analytic one. The current OpenMDAO implementation does not handle the implicit solution for solver loops, but this is an area for future investigation. Implementing the chain rule method in a framework may be slightly more difficult than the analytic method once you include treatment of the solver loops.

Both the chain rule and analytic method have a performance problem with finite differencing a component with more inputs than the total number of parameters. The algorithm performs more finite differences than the problem needs. We proposed a solution where the local inputs are transformed and incremented along a single direction that is the linear combination of inputs that is most strongly influenced by the design parameter being differenced. To find the right projection, you need to compute the sensitivities of the component's inputs with respect to the design parameter. This is only possible with direct mode chain rule, where those variables are available right when they are needed. There may be a way to find this projection for the analytic methods as well, but this will need to be investigated more thoroughly. This idea has not been implemented in OpenMDAO.

The finite difference method is the slowest and is further disadvantaged by its lack of adjoint mode support. Fake finite difference was proposed as an improvement that allows the method to take advantage of available analytical derivatives. We have shown that this method works for the general case including explicit solver loops. Further, this method is very easy to implement in a general MDAO framework and is robust to future changes.

Finally, we have scratched the surface on the subject of forming groups of components for finite difference. One interesting result is that, in the presence of non-differentiable connections, there are cases where fake

finite difference is the best and possibly only way to take advantage of some provided derivatives.

Acknowledgments

This work was supported by the Subsonic Fixed Wing project under NASA's Fundamental Aeronautics Program, and we gratefully acknowledge them for their support. The author would also like to thank Joaquim Martins and John Hwang for many discussions about derivatives, and the OpenMDAO user community for all of their contributions.

References

[1]Venderplaats, G. N., "CONMIN User's Manual," Tech. Rep. X-62282, NASA, 1978.

[2]Miura, H. and d Wojtkiewicz Jr, L. A., "NEWSUMT - A Fortran Program for Inequality Constrained Function Minimization - Users Guide," Tech. Rep. NASA-CR-159070, University of California, Los Angeles, 1979.

[3]Kraft, D., "A software package for sequential quadratic programming," Tech. Rep. DFVLR-FB 88-28, DLR German Aerospace Center.

[4]Giles, M. A. and Pierce, N. A., "An introduction to the adjoint approach to design," *Flow, Turbulence and Combustion*, Vol. 65, No. 3-4, 2000, pp. 393–415.

[5]Martins, J. R. R. A., Kroo, I., and Alonso, J., "An automated method for sensitivity analysis using complex variables." *Proceedings of the 38th Aerospace Sciences Meeting*, Reno, NV, January 2000, AIAA 2000-0689.

[6]Yu, W. and Blair, M., "DNAD, a Simple Tool for Automatic Differentiation of Fortran Codes Using Dual Numbers," *Computer Physics Communications*, Pending publication.

[7]Bischof, C., Corliss, G., Green, L., Griewank, A., Haigler, K., and Newman, P., "Automatic differentiation of advanced CFD codes for multidisciplinary design," *Computing Systems in Engineering*, Vol. 3, No. 6, December 1992, pp. 625–637.

[8]Sobieszczanski-Sobieski, J., "Sensitivity of Complex, Internally Coupled Systems," *AIAA Journal*, Vol. 28, No. 1, 1990, pp. 153–160.

[9]Martins, J. R. R. A. and Hwang, J. T., "Review and Unification of Methods for Computing Derivatives of Multidisciplinary Systems," *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Honolulu, HI, April 2012, AIAA 2000-0689.

[10]Martins, J. R. R. A., Alonso, J. J., and Reuther, J. J., "A Coupled-Adjoint Sensitivity Analysis Method for High-Fidelity Aero-Structural Design," *Optimization and Engineering*, Vol. 6, No. 1, March 2005, pp. 33–62.

[11]Gray, J. and Moore, K. T., "OpenMDAO: An Open-Source Framework for Multidisciplinary Analysis and Optimization," *13th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, AIAA, Fort Worth, Texas, August 2010, AIAA 2010-9101.

[12]Moore, K., Naylor, B., and Gray, J., "The Development of an Open-Source Framework for Multidisciplinary Analysis and Optimization," *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, AIAA, Victoria, Canada, August 2008, AIAA 2008-6069.

[13]Lambe, A. B. and Martins, J. R. R. A., "Extensions to the Design Structure Matrix for the Description of Multidisciplinary Design, Analysis, and Optimization Processes," *Structural and Multidisciplinary Optimization*, 2012.

[14]"NetworkX," http://networkx.lanl.gov/.

[15]Gilbert, J., "Automatic Differentiation and Iterative Processes," *Optimization Methods and Software*, Vol. 1, No. 1, 1992, pp. 13–21.

[16]Martins, J. R. R. A. and Hwang, J. T., "A Unification of Discrete Methods for Computing Derivatives of Single- and Multi-disciplinary Computational Models," *AIAA Journal*, Pending publication.