AIAA - 98-4872

Optimization in the Adaptive Modeling Language

Duane E. Veley

Air Force Research Laboratory, Structures Division, AFRL/VASD
2130 Eighth St., Ste. 1, Wright-Patterson AFB OH 45433-7542

Abstract

The Adaptive Modeling Language® (AML) is an object-oriented, knowledge-based programming language with some unique capabilities which make it a very versatile design environment. Some of these capabilities include dependency tracking, demand driven calculations, run-time model modification and collaborative engineering. Because the model can be modified at run-time, the user of the code is permitted more flexibility in the choosing of the design variables, objectives and constraints than the code developer initially conceived. However, one of the current deficiencies of AML for the design process is optimization. This paper presents an integration of the Adaptive Modeling Language with the Design Optimization Tools (DOT™), a commercial off-the-shelf optimization algorithm. An example showing the use of DOT in AML is given. User supplied gradient information is obtained from automatic differentiation in AML. An overview of automatic differentiation in AML is given.

Introduction

The Air Vehicles Directorate of the Air Force Research Laboratory (AFRL/VA) is looking to develop a technology assessment system in order to determine the impact of the technologies they develop on aerospace vehicles (Veley *et al*, 1998). The specifications for the Aerospace Technology Assessment System (ATAS) provide not only a description of what ATAS is to do, but specifications on the architecture of the environment in which the system is developed. Several environments are potentially suitable to accomplish the ATAS objectives and the Adaptive Modeling Language (AML) is one of these environments. The purpose of this paper is to assess how optimization, one of the technology requirements for ATAS, operates in the AML environment.

Overview of AML

The Adaptive Modeling Language (AML) is an object-oriented, knowledge-based, programming language developed by TechnoSoft, Inc. AML has several unique capabilities which make it an excellent language for designing engineering systems. These capabilities include dependency tracking, demand-driven calculations (lazy evaluations), adaptive class structure, and the flexibility of properties to be of any class. These capabilities provide a powerful tool by which design studies can be made rapidly.

Dependency tracking is the ability of any object (including properties) to know on which objects/properties it depends and which objects/properties it effects. For example, the width of a file cabinet may depend on the width of the drawers, the number of drawers and the thickness of the shell. At the same time the width of the drawers may depend on the size of paper (U.S. letter, U.S. legal, A4, *etc*.) that is intended to be stored in the file cabinet. The drawer would know that if a different paper size were selected, its width would need to change also. Furthermore, the cabinet would know that its width would need to be changed since the width of the drawer is being changed.

In AML a value is computed only when that calculation is demanded, hence the term, demand-driven calculations. Calculations can be demanded either directly or indirectly. If the paper size is changed, as in the example above, the width of the drawer and the width of the file cabinet are smashed (*i.e.*, a value is no longer associated with the width of the drawer or file cabinet) since these properties depend on the paper size. If the user asks to see the width of the file cabinet at this point, the width of the file cabinet is said to be demanded directly. Since the file cabinet width depends on the width of the drawer, the width of the drawer is also demanded, but indirectly. New values for the cabinet width and drawer width are now available.

A look at our file cabinet would reveal that so far there is an outer cabinet and there are drawers, but the hardware is missing. The drawers need handles so that can be opened and closed, they need rails to slide along and one might want to add a lock to the file cabinet. These things may not be part of the original class definition of the file cabinet, but that does not present a problem to AML. After the cabinet is sized for the paper, or even before, these other features may be added to an instance of the file cabinet model. This ability to add properties and subobjects to an instantiated object is known as an adaptive class structure. That is to say that the class structure of the file cabinet can be adapted to include the hardware after the file cabinet object has been instantiated. Now that the cabinet has rails for the drawers, the front panel of the drawer and the entire cabinet may be made wider to allow for the rails. The rules defining the cabinet width may change because the rails were added. This ability to modify the formulas of properties after they have been instantiated is another feature of the adaptive class structure.

Since the cross-section of the rail helps describe the nature of the rails as opposed to being an object that is a discrete portion of the object, it would be desirable to store the cross-section as a property rather than a subobject. Even though the cross-section does not have the typical value-function form of a property, it can be placed as a property of the object, since all properties are objects.

Now suppose that the designer of file cabinets is not the designer of rails for file cabinets. The requirements that you have developed for the rails require a new rail design. These rail requirements are stored in the model. A rail designer can access the model that is on the file cabinet designer's computer from another computer, get the requirements for the rails, design the rails on the rail designer's computer, and send the designed rails back to the file cabinet designer's computer. Similarly, the analyst can access the model of the file cabinet and rails and perform a stress analysis on the system. Each of these engineers can work on their own machine, accessing the same model and report results back to a central model. This feature of AML truly allows concurrent engineering to occur.

Suppose that the objective of this file cabinet design is to satisfy customer needs. A number of customers have called and complained that the file cabinets are too heavy to move. However, you know from past experience that many people try to move the file cabinets while they are full and the cabinets must be strong enough to handle that situation. The objective of this new design then is to redesign the cabinets so that they are lighter yet still strong enough that they are not damaged when moved full. So as the designer and analyst need optimization to aid in finding a suitable new design. However, AML does not currently have a built in optimization algorithm, but it can interface with other codes including optimization algorithms.

To interface with another code, the program developer can either have AML create an input file, launch a stand-alone program and read the result file back into AML or the developer can write an AML interface to call a function that is written in another language, such as FORTRAN, C or C++. In this paper, the latter option is used to interface DOT, an optimization method written in FORTRAN, to AML.

Automatic Differentiation Methodology

Optimization of functions of continuous variables requires gradient or derivative information. By applying the rules of differentiation to a computer code, additional code can be automatically generated which computes the derivatives of the original code. The derivatives of the code that are generated are those specified by the user of the automatic differentiation algorithm. For the optimization problem the derivatives of concern are those of the objective and constraints with respect to the design variables.

Mathematical Formulation

The fundamental principle used in this effort is the chain rule as it applies to functions of several variables. This can be stated as: Given a function *u* of several variables, *x*, *y*, *z*, and *t*, some of which are functions of one of the variables, *t*

$$u = f(x, y, z, t)\,,\ x = g(t)\,,\ y = h(t)\,,\ z = k(t)\,. \quad (1)$$

The derivative of the function *u* with respect to *t* may be written as

$$\frac{du}{dt} = \frac{\partial u}{\partial x}\frac{dx}{dt} + \frac{\partial u}{\partial y}\frac{dy}{dt} + \frac{\partial u}{\partial z}\frac{dz}{dt} + \frac{\partial u}{\partial t}\,. \quad (2)$$

Likewise the partial derivatives may be expanded by the chain rule for several variables. For example, if the function u has the form $u = f(x, y, z, q(r(t), s(t)))$, then

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial q}\frac{\partial q}{\partial t} = \frac{\partial u}{\partial q}\left(\frac{\partial q}{\partial r}\frac{\partial r}{\partial t} + \frac{\partial q}{\partial s}\frac{\partial s}{\partial t}\right). \quad (3)$$

Implementation

The Adaptive Modeling Language (AML) is an object-oriented package that uses the syntax characteristics of LISP. In this syntax a function and its arguments are given in a list, when the function is the first item in the list. The result of executing that list is the result of the function.

Two classes of objects are created for doing automatic differentiation in AML: the TOTAL-DERIVATIVE-OBJECT and the PARTIAL-DERIVATIVE-OBJECT. Both inherit form PROPERTY-OBJECT to give them all the features of a value-formula property. The automatic differentiation algorithm (ADA) puts formulas that look like Eq. (2) into the TOTAL-DERIVATIVE-OBJECT. The ADA uses Eq. (3) to symbolically differentiate the formula of the dependent variable (property) with respect to the intermediate independent variable (property) and the resulting formula is stored in the PARTIAL-DERIVATIVE-OBJECT.

In finding a derivative in AML, the ADA first checks to see if the derivative already exists. If the derivative does not exist, a TOTAL-DERIVATIVE-OBJECT is added as a property of the dependent variable object. If the derivative already exists, the ADA checks to see if the formula for the dependent variable has changed. If it has changed (or the TOTAL-DERIVATIVE-OBJECT has just been created), a new formula is computed for the TOTAL-DERIVATIVE-OBJECT. The TOTAL-DERIVATIVE-OBJECT creates, if necessary, all PARTIAL-DERIVATIVE-OBJECT and TOTAL-DERIVATIVE-OBJECT instances that it needs.

To create the formula for a total derivative, the dependencies of the dependent property are searched to see if they include the independent property. This search includes both direct and indirect dependencies. If no dependency is found, the formula, and hence the value, of the derivative is zero. If dependencies are found, partial derivatives and total derivatives called for by direct dependencies are created, if necessary. In creating a total derivative, additional partial and total derivatives may be required and are created, if they do not already exist, through recursion.

The partial derivative objects are initiated in much the same way as the total derivative object. First, the ADA checks to see if the partial derivative object exists then adds one if it does not exist. The formula of the dependent variable is checked to see if it has changed and if it has, a new partial derivative formula is created. The primary difference in the partial and total derivative properties is the way in which the derivatives are computed. Whereas, the total derivative object only implements Eq. (2) (*i.e.*, the formula for total derivatives is simply the sum of products of other total and partial derivatives), the partial derivative formulas are the derivatives of the actual formulas.

The list nature of AML aids in the automatic partial differentiation of functions. Each list is processed by looking at the first element of the list; this element is the function name. A large case construct directs the differentiation procedure to the appropriate rules for that function. For example, if the function name is SIN, the case selector would call the function which partial differentiates the SIN function. Since the SIN function takes only one argument, it uses the chain rule to return a formula that is the product of COS of the argument and the partial derivative of the argument with respect to the independent variable. The partial derivative of the argument is obtained by recursively calling the large case construct that directs the partial derivative operations. This recursive calling of the partial differentiation function is in effect an implementation of the chain rule of differentiation.

In order to clarify the process a little better, a small example is given. A class called cube is defined in Fig. 1. The circumflex (^) preceding the names in the formulas tells the formula to look for the property (or subobject) that is under the cube. Also note that it does not matter in which order the properties are listed. This is due to the dependency-tracking and demand-drive calculation features of AML. Upon creating a model of class cube, one would find that it has the properties height, width, depth, and cost. None of these properties have subproperties at the time that they are created. The dependencies between the properties are shown in Fig. 2.

It is desired to find the sensitivities of cost to the geometric parameters (*e.g.*, height and depth). In computing the derivative of cost with respect to depth, a TOTAL-DERIVATIVE-OBJECT, d_depth, is added to the cost property and is given the formula (+ (* (the cost p_width) (the width d_depth)) (the cost p_depth)). In generating this formula, additional derivative properties are generated in a recursive fashion: (the cost p_width), (the cost p_depth) and (the width d_depth). If one were to examine the formula for (the width d_depth), one would find that another object, (the width p_depth) is also needed to compute that formula. The recursive nature of the ADA will create each of the needed derivatives until all of the derivatives are completed.

It is appropriate at this time to interpret the formula for d_depth. A formula is given by a set of nested lists. A list is defined by a matching pair of parentheses. The first item in the list is the function name, so for this formula the functions used are "+", "*" and "the". The first two functions take the remaining items in its list and add or multiply them together as appropriate. The "the" function is a means of referring to a property or subob-

```
(define-class cube
    :inherit-from (box-object)
    :properties(
        height          4.0
        width           (+ (* 6.0 ^depth)
                           (* 0.8 ^height)
                           )
        depth           0.1
        cost            (* ^depth
                           (+ ^height ^width)
                           )
    )
)
```

Figure 1. Example of an AML class definition

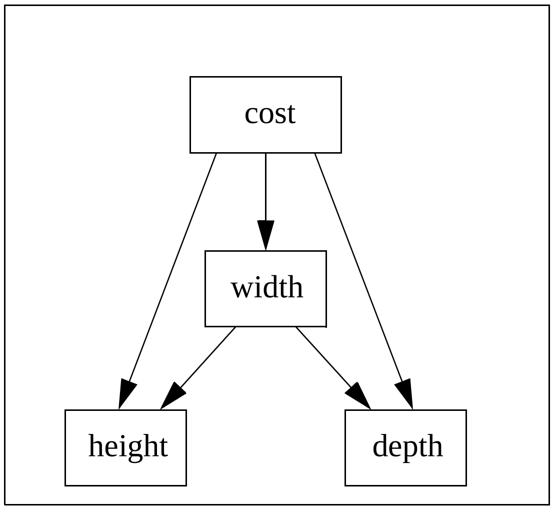

Figure 2. Dependency structure of cube class properties

ject. The circumflex seen earlier is a special form of the “the” function. The “d_” and “p_” notations show that the property is a total derivative or partial derivative object, respectively. The reference (the width d_depth) points to the property that is the total derivative of width with respect to depth. Likewise, the reference (the cost p_width) points to the property that is the partial derivative of cost with respect to width.

The derivatives of cost with respect to height could similarly be obtained and would contain a reference to (the cost p_width). Since this was already generated while getting (the cost d_depth), it is not generated again. A simple conformation that the formula for cost has not changed would prevent the ADA from calculating these derivatives again. Similarly, if width is a constraint, the formulas for the derivatives of width with respect to height and depth would have already been computed.

The ADA does not actually compute the values of the derivatives, but simply computes the formulas. The values of the derivatives are then obtained when the derivative is requested either directly or indirectly via demand-driven calculations. For example, if the value of (the cost d_depth) is requested directly, the value of (the cost p_width), (the cost p_depth), (the width d_depth) and (the width p_depth) are all requested indirectly and are evaluated at that time, if they have not already been evaluated.

The implementation of the ADA in AML allows the user to create additional derivatives as desired. All the derivative objects that are automatically generated by the ADA could have been defined in the class definition. If properly named, the ADA will recognize the derivative object and would not create a new one. The ADA would however check to make sure that the formula is up to date, and if not generate a new derivative formula. The code developer can also write functions that define the derivatives of a function. If the function is properly named, and either loaded into the AML environment, or stored in the appropriate file, then that definition can be used by the ADA, and the ADA will not generate its own derivative. The ability to use pre-defined functions is particularly useful if a function is written in a different language. The ADA would not be able to differentiate that function, but if derivative functions existed in AML or the other language, the ADA could make use of that function if its AML interface function is properly named.

Although the dependent variables may depend on objects of classes that are not derived from the property-object class, sensitivities to these objects are assumed to be identically zero. This is a safe assumption since only classes that inherit from the property-object class contain a formula and value. Other classes represent discrete objects and hence, their derivatives are identically zero. If design studies are to be done with these discrete objects as independent variables, a non-gradient based optimization procedure is required.

Current Status of Automatic Differentiation in AML

Since the first paper on automatic differentiation in AML was written (Veley and Zweber, 1998), several features have been added to or enhanced in it. Two enhancements to the differentiation of methods have been implemented. The first allows properties of the object to be independent variables. The other allows derivative functions and methods to be accessed from source files. Additional capabilities include the ability to follow the object tree structure better (*i.e.* the dependent and independent objects are not necessarily properties of the same object), the ability to differentiate vectors and arrays including inherent vector functions, and the ability to differentiate the LOOP function for the more common clauses.

The automatic differentiation algorithm is still far from perfect and requires additional work. In particular, the ADA has trouble when an argument to a function is itself a function, and the dependency tracking of formulas does not work properly.

DOT and the AML-DOT Interface

The Design Optimization Tools (DOT) (Vanderplaats, 1995) is a set of FORTRAN subroutines that allow several optimization algorithms to be performed. The default algorithm is the modified method of feasible directions (MMFD), which is the algorithm that is used in the example presented later. Although only one of the possible algorithms is used, all algorithms within DOT are accessible to AML through the one interface that is made.

Some optimization algorithms are coded so that the name of an analysis subroutine is passed to the optimizer. The optimizer is called once and the optimizer calls the analysis module as many times as it needs for the optimization process. When the optimization is complete, the optimization routine returns control to the calling routine. This set up is not convenient for interfacing with AML because it requires to interfaces: one where AML calls the optimization algorithm and the other where the optimization calls the analysis module which is also in AML.

With DOT, only one interface is required. This is the interface where AML calls DOT. When DOT requires an analysis to be performed, gradient information or is complete, DOT returns control to the calling routine with a flag that indicates which of these options is to be executed. Thus, DOT is to be called many times from within a loop that also includes the objective, constraint and gradient calculations. This feature of DOT allows optimization to act as a module rather than a driver. It also makes optimization easier to integrate into a computer program.

In creating the AML-DOT interface, one object class, one foreign function and one method are defined in AML. The object class defined is called the DOT-OPTIMIZATION-OBJECT and it inherits its properties from the object-class that is a generic class for which the dependency tracking capability is established. An object of this class will be a subobject of the object being optimized and is used merely to store the information that is sent to and from DOT. The DOT-OPTIMIZATION-OBJECT contains no subobjects. The properties in this object are the parameters that are passed back and forth to DOT through the subroutine's argument list. In order that the current value of the design variables, constraints and objectives may be used, the DOT-OPTIMIZATION-OBJECT stores the pointers to those properties.

A foreign function definition is the actual interface between AML and code that is written in another language. The AML foreign function definition describes the function or subroutine that is to be called from AML. This function is described by its name, the type of its return value (AML assumes that it is interfacing with a function and not necessarily with a subroutine), the language in which the foreign function is written, and the arguments to the function along with their data types.

A method in AML is like a function except that a method applies to a particular class of objects. Thus several methods may be written with the same name but each acting on a different class. The method developed here, DOT-OPTIMIZE, interfaces the DOT-OPTIMIZATION-OBJECT class to the foreign function by converting the properties of the object to the proper data type. Values associated with the pointers for the design variables, constraints and objective are obtained at this time and placed in vectors that are passed to DOT. At this time, if any of the values that are being passed to DOT have not been computed, they will be computed because of the demand-driven calculation capability of AML. DOT is then called a single time through the AML foreign function definition for DOT. Upon return from DOT, DOT-OPTIMIZE places the results (values returned from DOT) into the appropriate properties of the DOT-OPTIMIZATION-OBJECT. The design variable pointers contained in the DOT-OPTIMIZATION-OBJECT are used to direct the new design variable values to the appropriate properties. Since the computations are demand-driven in AML and properties maintain their value (once computed) until one of the properties it depends on is smashed or changed, it is not prudent to change all of the design variable values that are returned from DOT, but only the ones that have changed. This is particular useful (reduces computational time) when DOT's internal finite-difference gradient method is used, since it changes two design variables at a time, at most.

These three parts provide an interface between DOT and AML, but do not call DOT repeatedly as needed for the optimization process. An additional method, OPTIMIZE, is defined which operates on the AML-CLASS, the top level class from which all other classes inherit. This allows the method to work on all objects regardless of class. This method adds the DOT-OPTIMIZATION-OBJECT as a subobject to the object upon which the method is operating if it does not already exist as a subobject of that object. OPTIMIZE then proceeds to call the DOT-OPTIMIZE method and checks the code returned from DOT to see how to proceed. The DOT-OPTIMIZE function is called recursively passing either object and constraint data or gradient data until DOT returns an optimization complete code. The results of the optimization are then stored in the DOT-OPTIMIZATION-OBJECT and control is returned to the calling function.

As is mentioned above, DOT returns a flag which requests either the objective and constraints or the sensitivities of the objective and constraints to be provided for the current values of the design variables. The OPTIMIZE method deciphers the flag and generates the correct information. DOT can either request sensitivity data to be provided, or it can use the finite difference method and compute derivatives itself. This option is specified by one of the many parameters in the arguments list that is passed to DOT. If sensitivity data is to be calculated by the user, automatic differentiation, an emerging feature recently added to AML (Veley and Zweber, 1998), would make this task of developing derivative information easier. The OPTIMIZE method ties into the automatic differentiation algorithm to obtain sensitivity data.

An efficiency of DOT is the way it requests user supplied sensitivities. DOT is normally not concerned with the sensitivities of all of the constraints, but only those that are violated and those that are active (close to the limit). Thus, the set of sensitivities requested is usually not the full set. This limited request of constraint sensitivities plays well with the demand-driven calculation capability of AML. AML will only compute those derivatives that are needed by DOT.

Example

A conceptual level design study is performed to assess the trade-offs between aerodynamics and structures for selected wing dimensions. The wing planform is shown in Fig. 3. A coarse description of the aerodynamic force coefficients for lift, drag and moment are given by

$$C_L = \frac{0.45\pi^3}{180}\frac{c_r b}{S}\alpha + 0.22 \tag{4}$$

$$C_D = \frac{C_L^2}{\pi A_R} \tag{5}$$

$$C_M = -0.03 C_L + 0.046 \tag{6}$$

where C_L, C_D and C_M are the lift, drag and moment coefficients, respectively, c_r is the root chord, b is the span, S is the wing area, α is the angle of attack and A_R is the aspect ratio given by $A_R = b^2/S$. An elliptic distribution of the total normal force coefficient as a function of spanwise location is given by

$$C_n(y) = \frac{\bar{c}}{c(y)}\left(\frac{2C_n}{\pi}\right)\sqrt{1 - \frac{4y^2}{b^2}}\,. \tag{7}$$

where $\bar{c}$ is the mean aerodynamic chord length, $c(y)$ is the chord length at spanwise location y and C_n is the normal coefficient of the wing. These relations are used to compute the shear force and bending moment at the wing root for a positive high angle of attack condition using the following relations:

$$V_i = V_{i-1} + q\frac{(c_i C_{ni} + c_{i-1}C_{n,\,i-1})}{2}(y_{i-1} - y_i) \tag{8}$$

$$M_i = M_{i-1} + \frac{V_i + V_{i-1}}{2}(y_{i-1} - y_i) \tag{9}$$

where V_i is the shear force at station i, M_i is the moment at station i, y_i is the spanwise location of station i, q is the aerodynamic pressure and the stations begin at the tip of the wing where the shear and moment are both zero. These loads are then used to compute the shear flow in the skin and webs and the direct stress in the booms of the idealized wing structure shown in Fig. 4 (see Megson, 1990).

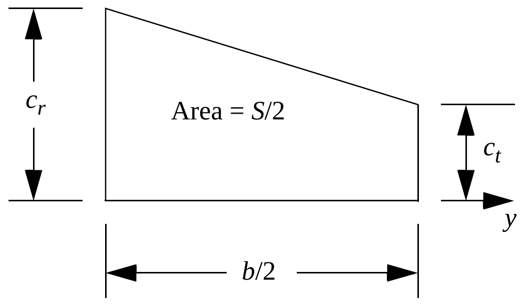

Figure 3. Wing planform.

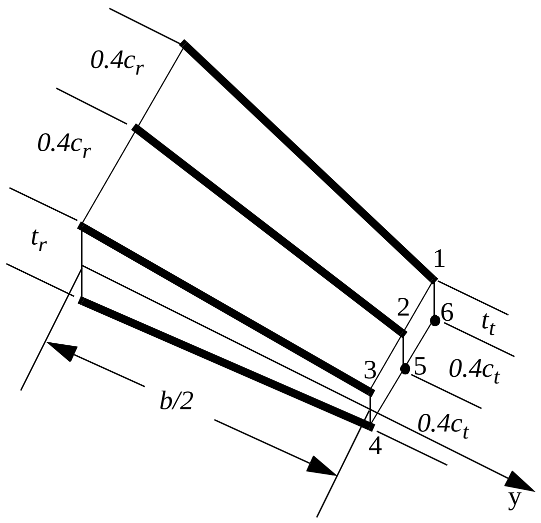

Figure 4. Wing structure

The wing is to be designed for minimum mass (initial value 224 kg) with constraints on the direct stress in the booms, on the shear flows in the webs and on the total lifting capability of the wing at cruise conditions. The equations for the aerodynamic load coefficients are assumed to remain suitable for changes in the wing geometry within the design space. The initial values and side constraints of the design variables are given in Table 1. The areas of the four corner booms are linked to a single design variable and the areas of the midchord booms are linked to .a single design variable. The design variables are the areas of booms 1 and 2 (B_1 and B_2), the root chord length (c_r), the tip chord length (c_t), the span (b), the skin thickness (t_s), the web thickness (t_w), the thickness of the wing at the root (t_r) and the thickness of the wing at the tip (t_t). It is assumed that the shear load acts along the center spar. Then at any spanwise location, the cross-section is symmetrical and the shear flows will be symmetrical. Thus, constraints are placed on the shear stresses in webs 1-6 and 2-5, (τ_{ij} where i and j denotes the booms on either side of the web). Due to the symmetry of the cross-section, only two different stresses are realized in the booms. Therefore, constraints are placed only on the direct stresses in booms 1 and 2 (σ_{1z} and σ_{2z}). In order to keep the wing design from being governed totally by structural considerations, a constraint is also placed on the total lifting force of the wing at cruise conditions. The constraint limits are given in Table 2.

Table 1: Initial values of design variables and side constraints

	lower bound	initial value	upper bound
B_1, mm^2	600	1000	1500
B_2, mm^2	600	1200	1500
c_r, m	2.5	3.2	3.8
c_t, m	1.75	2.0	3.2
b, m	8.0	12.0	18.0
t_s, mm	1.0	3.0	10.0
t_w, mm	1.0	2.0	10.0
t_r, mm	600	750	1500
t_t, mm	300	400	1000

Table 2: Constraints

Constraint	Relation	Limit	Initial Value
σ_1, N/mm^2	$\leq$	100	110
σ_2, N/mm^2	$\leq$	100	110
τ_{16}, N/mm^2	$\leq$	25.0	25.5
τ_{25}, N/mm^2	$\leq$	25.0	33.8
F_n, N	$\geq$	97,000	96,900

The optimization was performed in two ways: using the finite difference capability within DOT and using the user supplied gradients obtained from automatic differentiation. The results given in Table 3 show that the two methods each obtained the same design

The real interest in this study though is the timing of the optimization process. A cpu timer was initiated at the beginning of the optimization loop for both the automatic differentiation supplied gradient and the internal finite difference gradients. The finite difference method took 14.3 cpu seconds and the automatic differentiation supplied gradients run took 17.1 cpu seconds. The finite difference gradient method required 243 function evaluations and the automatic differentiation supplied gradients required only 151 function evaluations and 10 gradient evaluations. Assuming that the average function evaluation took the same time for both cases, the average gradient calculation would have taken 14 times

Table 3: Results

	Finite difference	Automatic differentiation
B_1, mm^2	823	824
B_2, mm^2	823	824
c_r, m	3.80	3.80
c_t, m	3.20	3.20
b, m	9.41	9.41
t_s, mm	1.00	1.00
t_w, mm	2.15	2.15
t_r, mm	842	841
t_t, mm	300	300
σ_1, N/mm^2	100	100
σ_2, N/mm^2	100	100
τ_{16}, N/mm^2	25.1	25.1
τ_{25}, N/mm^2	25.1	25.1
F_n, N	96,700	96,700
m_w, kg	129	129

as long as a function evaluation. However, the assumption may not be a valid one.

The demand driven calculations feature may have significantly influenced the results. In the general case of looking at a new function evaluation or gradient evaluation, all of the design variables would have changed thus creating the greatest amount of demand-driven computation that needs to be performed for the respective evaluation. However, when DOT is computing gradients with the finite difference method, at most two design variables have changed. Therefore, the number of calculations needed are reduced for those iterations. On the other hand, the gradient evaluations using the AD supplied gradients will always have all design variables different from the previous time.

It is also possible that inefficiencies in the code generated by the automatic differentiator exist that cause the AD supplied gradient method to be slower. The slow down may also be a function of the problem. It is not unheard of for code generated by automatic differentiation in other languages to be as slow as or slower than using finite difference gradients. However, it remains that if automatic differentiation in the Adaptive Modeling Language is to survive, additional assessments into the efficiency of that code need to be made.

Summary

Optimization in the Adaptive Modeling Language is as easy as it is in any other language. DOT works well with AML and the automatic differentiation in AML by promoting the modularity of optimization and the demand-driven computations of only active constraint sensitivities. The demand-driven calculations capability of AML even enhances the computational efficiency of the finite difference gradients even to the point where they are competitive with automatic differentiation sensitivities. A timing comparison between the optimization results using the finite difference gradients and those using the automatic differentiation supplied gradients show that the automatic differentiation algorithm could use some improvements in the efficiency of the code that it generates.

Bibliography

DOT Design Optimization Tool (1995), Vanderplaats Research & Development, Colorado Springs CO.

Megson, T. H. G. (1990), *Aircraft Structures for Engineering Students*, Second Edition, Halstaed Press, New York, NY.

Veley, D. E., Blair, M., and Zweber, J. V. (1998), “Aerospace Technology Assessment System,” Proceedings of the 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St. Louis, MO, September 2-4, 1998.

Veley, D. E. and Zweber, J. V. (1998), “Automatic Differentiation in the Adaptive Modeling Language,” Proceedings of the Australasian Conference on Structural Optimization, Sydney, Australia, February 11-13, 1998.