Using Automatic Differentiation with the Quasi-Procedural Method for Multidisciplinary Design Optimization*

by

Steve Altus[†], Chris Bischof[‡], Paul Hovland[‡], and Ilan Kroo[†].

Submitted to
34th AIAA Aerospace Sciences Meeting

Abstract

As computers have become increasingly powerful, the field of design optimization has moved toward higher fidelity models (involving many more variables) in the early stages of design. One way in which this movement has manifested itself is in the increasing popularity of multidisciplinary design optimization (MDO). Because the models used in MDO are large and very complicated, a modular design is desirable. Because there are many design parameters to optimize, derivatives must be computed accurately and efficiently. This paper describes how the quasi-procedural program architecture developed by Takai and Kroo [9] and the technique of automatic differentiation [6] can be combined to effectively address these needs. The two techniques are explained, the manner in which they were integrated into a single framework is described, and the result of using this framework for an optimization problem in airplane design is presented.

1 Introduction

Over the past several years, there has been a movement in the field of optimization toward multidisciplinary design optimization (MDO), incorporating several design goals into a single optimization procedure. For example, an airplane designer may incorporate fluid dynamics and structural analysis into a single model. For optimization of the design parameters for this multidisciplinary model to remain practical, the model should have a modular design and derivatives must be computed efficiently. Modularity is important because the multidisciplinary model may be created by several development teams and also because it simplifies the integration of new code when a better model for any of the disciplines becomes available. The combination of automatic differentiation and the quasi-procedural method provides an attractive environment for doing MDO. The quasi-procedural method

*This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, the National Aerospace Agency under Purchase Order L25935D, the U.S. Department of Defense through an NDSEG fellowship, and the National Science Foundation through NSF Cooperative Agreement No. CCR-9120008.

[†]Department of Aeronautics and Astronautics, Stanford University, Stanford, CA 94305, {altus, kroo}@leland.stanford.edu.

[‡]Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, {bischof,hovland}@mcs.anl.gov.

is a modular framework for optimization and provides efficiency by avoiding redundant computations. Automatic differentiation provides efficiency through fast, accurate derivatives and supports the integration of new code by automatically creating modules that compute derivatives quickly with minimal user intervention.

This paper describes how automatic differentiation and the quasi-procedural method were applied to the optimization of an airplane design. Sections 2 and 3 introduce the quasi-procedural method and automatic differentiation, respectively. Section 4 describes how the two techniques were incorporated into a single environment and Section 5 presents our experimental results using this environment. Section 6 concludes with an analysis of how the combined approach might facilitate MDO, and also an assessment of when this approach might not be appropriate.

2 The quasi-procedural method

The quasi-procedural method [4, 7] is a form of non-procedural programming. Unlike conventional procedural programs in which computation proceeds from inputs to outputs according to a rigid structure, non-procedural systems are free to reorganize computations as necessary to compute the desired outputs. Thus, it is the outputs that drive the computation, rather than the inputs. However, non-procedural programming is of limited value at fine granularity, as programmers can often utilize knowledge of a computation to develop extremely efficient small procedural subprograms.

2.1 A composite system

The quasi-procedural method attempts to exploit the best of both methods by allowing the programmer to develop efficient subprograms and providing a system for linking these subprograms so that they can be executed non-procedurally. This linkage system is demand-driven. When the value of a variable is requested, the executive system determines which subprogram is responsible for computing that value and runs the appropriate subroutine. If that routine requires inputs, it informs the executive; the executive provides the desired inputs either by looking them up in a database or by executing additional routines.

This type of request-driven execution is depicted in Figure 1. For example, if the value of I is desired, the executive invokes subprogram 5. Subprogram 5 requires values for G and H, which causes the executive to invoke subprograms 3 and 4. This sequence of requests continues until values for A, B, C, and D are provided, at which point execution of the subprograms commences, and a value for I is produced. This approach, together with a method for assessing the validity of the values of intermediate variables (so that upon a request for an input value, the executive system can determine whether to provide a previously computed value or to recompute the value), constitutes the quasi-procedural method.

The importance of maintaining information about the validity of intermediate values can be seen by again considering the program in Figure 1. Suppose we compute the value of I as before, then decide we want to change the value of design parameter C and recompute I. Since the values of E, F, and G do not depend on C, there is no need to recompute these values by executing subprograms 1, 2, and 3. Instead, it is sufficient to recompute H and

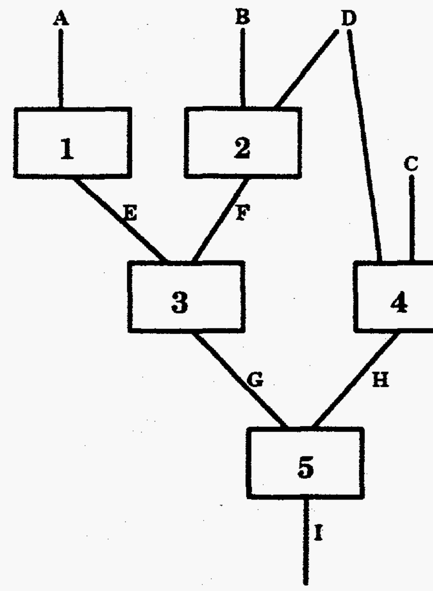

Figure 1: A sample computational path

I by executing subprograms 4 and 5. This potentially very large computational savings is the benefit of using the quasi-procedural method.

2.2 Modularity

An important characteristic of the quasi-procedural method in addition to its efficiency is its modularity. Each of the subprograms represents a separate module, and it is easy to replace one of these subprograms without affecting the rest of the computation. The ability to incorporate new code without having to rewrite, or even recompile, modules corresponding to other parts of the computation is very important to MDO. New or better modules will frequently be added to the multidisciplinary model, but other development teams should be unaffected by these modifications. Thus, modularity is an important consideration for any framework to be used in support of MDO.

2.3 GENIE

GENIE is a generic framework for engineering computations [4,7]. GENIE provides a set of routines that facilitates interfacing a set of modeling routines to the quasi-procedural method. This interface is provided via GET and PUT routines, and the computation proceeds in the following manner:

1. The optimizer issues a GET operation, signaling a request for a specified objective function or constraint value.

2. The GET routine selects the appropriate analysis routine and calls it.

3. The analysis routine issues one or more calls to GET to load the required input variables. In the event that the values of these variables are not known or are invalid, the GENIE executive calls the appropriate analysis routine, and the process repeats.

Thus, GET is called recursively until the values of all required input variables are known. This method enables quantities to be computed only as needed and without a fixed execution path. A sample analysis routine is shown in Figure 2.

As was mentioned above, a great deal of the efficiency of the quasi-procedural method can be attributed to avoiding unnecessary recomputing of intermediate quantities. Thus a major function of the GENIE framework is maintaining information about the validity of the values of variables. Whenever one of the input parameters for a routine is modified (marked 'invalid'), the quasi-procedural executive marks all outputs of that routine 'invalid.'

3 Automatic Differentiation and ADIFOR

In general, multidisciplinary design optimization requires the derivatives of an objective function and several constraints with respect to many design parameters. Since the function is typically described by an extremely complicated computer program, using a symbolic manipulator, such as Maple [3] is usually not an option. Similarly, developing derivative code by hand is unattractive, because it is complicated, tedious, and prone to errors. This approach is also ill-suited for rapid proto-typing, where parts of the system model may change several times, requiring additional code development for each new part. Consequently, optimization often relies on divided difference approximations to the desired derivatives. However, if an appropriate step size is not selected, these approximations can be grossly inaccurate. This hinders rapid proto-typing, because finding a good step size can be difficult and time-consuming, and a new step size must be determined each time the system model changes. Divided difference approximations may also take a long time to compute.

An alternative to all of these techniques is automatic differentiation. Automatic differentiation is a technique for computing the derivatives of a complicated function expressed in the form of a computer program [6]. The execution of a computer program consists of the composition of many elementary functions (such as multiplication, square root, and hyperbolic cosine), for each of which an analytic expression for derivatives is well known. So, by simply applying the chain rule

$$\frac{\partial}{\partial t} f(g(t))\Big|_{t=t_0} = \left(\frac{\partial}{\partial s} f(s)\Big|_{s=g(t_0)}\right) \left(\frac{\partial}{\partial t} g(t)\Big|_{t=t_0}\right) \tag{1}$$

repeatedly, it is possible to compute the derivatives of the function. For example, the code segment:

```
y = 2*x*x + 3*x + 7
z = 4 * sin(x)
f = sqrt(y*y + z*z)
```

may be converted into:

```
C---------------------------------------------------------------
      subroutine Qpsf

C     Profile:
C--------------------
C     This routine computes the dynamic pressure for a given
C     density and velocity.

C     Declarations:
C--------------------
      implicit none

      real Vknots, rho, Vfps, DynamicPress

      logical Abort

C     Required inputs:
C--------------------
      call GET(Vknots,'Speed'   )
      call GET(rho ,'Density')
      if(Abort()) return

C     Calculations:
C--------------------
      Vfps = Vknots*1.69
      DynamicPress = .5 * rho * Vfps*Vfps

C     Repack database:
C--------------------
      call PUT(DynamicPress,'DynamicPress')

      return
      end
```

Figure 2: A sample analysis routine

```
y = 2.0*x*x + 3.0*x + 7.0
g_y = 4.0*x*g_x + 3.0*g_x
z = 4.0 * sin(x)
g_z = 4.0*g_x*cos(x)
f = sqrt(y*y + z*z)
g_f = -0.5 * (2.0*y*g_y + 2.0*z*g_z)/ f
```

Here, g_*var* represents the derivatives of *var* with respect to the independent variables, in this case **x**. Thus, if **g_x** is initialized to 1.0 (since $\frac{\partial x}{\partial x} = 1.0$), upon exit the value of **g_f** will be $\frac{\partial f}{\partial x}$. If **x**,**y**,**z**, and **f** were vectors, then the appropriate initial value for **g_x** would be an identity matrix and **g_f** would still represent $\frac{\partial f}{\partial x}$. For this problem, the value of **g_x** is propagated through the derivative computation; hence, **g_x** is termed the *seed matrix* for this computation [2].

While symbolic differentiation uses the rules of calculus in a more or less mechanical way, automatic differentiation is intimately related to the program for the computation of the function to be differentiated. By applying the chain-rule step by step to the elementary operations executed in the course of computing the "function," automatic differentiation computes exact derivatives (within the limits of finite precision arithmetic) and avoids the potential pitfalls of divided differences. The technique of automatic differentiation is directly applicable to complex functions with branches and loops.

Automatic differentiation is amenable to modular program design. The ability to control which derivatives are computed through an appropriate initialization of the seed matrix means that we do not need to know which design parameters are being optimized at the time the derivative module is created. Furthermore, we can process individual modules separately, then connect the derivative modules using seed matrices. For example, suppose module **A** computes $y(x)$ and module **B** computes $z(y)$, and that we process these modules to yield derivative modules **g_A** and **g_B**. Then, if we initialize g_x (the seed matrix for module **g_A**) to an identity matrix, module **g_A** will compute $g\_y = \frac{\partial y}{\partial x}$. If we then pass g_y to module **g_B** as a seed matrix, this module will compute $g\_z = \frac{\partial z}{\partial y} \times g\_y = \frac{\partial z}{\partial y} \times \frac{\partial y}{\partial x} = \frac{\partial z}{\partial x}$ directly, exactly as if $z(x)$ was computed in a single module.

Many tools have been developed to support automatic differentiation. We used ADIFOR for our application. ADIFOR is a tool which provides automatic differentiation for programs written in Fortran 77 [1]. Given a Fortran subroutine (or collection of subroutines) for a function f, ADIFOR produces Fortran 77 subroutines for the computation of the derivatives of this function. ADIFOR differs from other approaches to automatic differentiation by being based on a source translator paradigm (as opposed to operator overloading) and by having been designed from the outset with large-scale codes in mind. Both features make it possible to easily incorporate derivative computation into a quasi-procedural system initially designed only to compute function values.

4 Adding automatic differentiation to the QPM framework

In order to allow GENIE to work together with ADIFOR-generated code, several enhancements were made to the GENIE framework. First, additional storage in the database was allocated, so that the database manager would have additional space to store gradients.

	QPM-DD	QPM-AD	ratio
PASS1	46.8	5.0	9.36
PASS2	14.9	4.2	3.55
PASS3	19.9	5.6	3.55
PASS4	19.9	5.5	3.62

Table 1: Results for the PASS problem on an RS6000

This storage is provided by an array which is parallel to the array in which the regular variables are stored. This enables lookup, validity checking, and other database maintenance operations to be performed for both a variable and the associated gradient at the same time. Second, the executive was modified so that the GET and PUT routines would return and store the values of a variable and the associated gradient (actually, new routines, called G_GET and G_PUT, were created to provide this functionality). Finally, one of the GENIE initialization routines was modified to perform the seed matrix initialization required by ADIFOR-generated code. By zeroing all elements of the array containing gradients, then setting the I^{th} element of the gradient associated with the I^{th} design variable equal to 1.0, an identity seed matrix is created automatically.

Making these changes to the GENIE framework has two important benefits. First, adding support for automatic differentiation to the framework makes using AD easier. The user does not have to deal with derivative object allocation, seed matrix initialization, or interfaces between modules. All of these tasks are done automatically in the executive. Once an analysis routine has been run through ADIFOR, the user does not even need to be aware that automatic differentiation is being used. Second, creating a direct association between variables and their derivative objects in the database means that consistency maintenance for the derivative objects can be done with zero additional overhead.

5 Experimental Results

To examine the suitability of the QPM-AD combination for multidisciplinary optimization, we applied the techniques to a complete aircraft model. The problem being studied is the synthesis of a twin-engined, 100-passenger, medium-range commercial transport. The objective is to minimize direct operating costs, subject to certain constraints in performance measures such as range and maximum field lengths. The design variables are weights, wing and tail size and shape parameters, takeoff engine size, cruise altitudes, and takeoff flap deflection.

The multidisciplinary analysis routines for the PASS aircraft model were processed using ADIFOR, which automatically replaced calls to GET and PUT with calls to G_GET and G_PUT, respectively. When these routines were compiled and linked with the enhanced GENIE framework and the NPSOL optimizer [5], we were able to do airplane design optimization using the quasi-procedural method and automatic differentiation.

The times required for the optimizer to find a minimum for various problems are reported in Tables 1–3. Problem PASS1 involves the optimization of 14 design parameters.

	QPM-DD	QPM-AD	ratio
PASS1	DNC	11.5	n/a
PASS2	DNC	9.8	n/a
PASS3	37.3	12.0	3.11
PASS4	30.7	10.4	2.95

Table 2: Results for the PASS problem on a Sun SPARC IPX

	QPM/DD	QPM/AD	ratio
PASS1	11.7	1.2	9.75
PASS2	3.7	1.0	3.70
PASS3	4.9	1.3	3.77
PASS4	5.0	1.2	4.17

Table 3: Results for the PASS problem on an IBM SP1 node

This problem was difficult to optimize, and the observation that one design variable was not critical to the design led to it's removal, yielding problem PASS2. Problems PASS2 and PASS3 involve the optimization of 13 design parameters, from different starting points. Problem PASS4 is the same as PASS3, but the optimality tolerance is reduced. Results are reported for the quasi-procedural method using divided difference approximation, abbreviated QPM-DD, and for the augmented quasi-procedural method, using derivative code generated by ADIFOR, abbreviated QPM-AD. The ratio of QPM-DD to QPM-AD is also reported. The abbreviation DNC is used when the optimization terminated without converging. We believe that due to the sensitivity of divided differences to step size, and the limitations of finite precision arithmetic, the NPSOL optimizer occasionally could not verify that it had a Kuhn-Tucker point on the SPARC workstation, but could on the IBM machines. This problem did not occur when derivatives were computed using automatic differentiation.

Design optimization using the quasi-procedural method is typically faster than design optimization without the quasi-procedural method [4]. Because the quasi-procedural method employs consistency maintenance and does not recompute values unless necessary, a great deal of computational cost can be eliminated. Furthermore, the quasi-procedural method with automatic differentiation often performs much better than the quasi-procedural method using divided difference approximations. The reason for the improvement is twofold:

1. The hybrid mode of automatic differentiation implemented by ADIFOR is often more efficient than divided differences. This is especially true of programs with many assignment statements with complex arithmetic expressions on the right hand side. The derivatives of these expressions are computed using the reverse mode of automatic differentiation, which requires a constant multiple of the time required to evaluate the

expression. Divided differences and the forward mode of automatic differentiation require time linear in the number of design variables. This is a general speedup seen in many applications of ADIFOR to engineering codes, and is not specific to the quasi-procedural method.

2. The perturbing of design variables needed to compute approximate derivatives using divided differences destroys the validity of the values of variables depending on those design variables. Thus, much of the efficiency of the quasi-procedural method is lost. GENIE compensates for this inefficiency by employing a specialized technique [4] to avoid recomputation insofar as possible. However, this advanced technique is inferior to the ability to compute derivatives without affecting the validity of any values, as is provided by automatic differentiation.

6 Conclusions

Multidisciplinary optimization has three distinguishing features: the system is often modelled by a large, complicated program developed my many different teams; there may be many changes to the system model, due either to rapid proto-typing or model refinement; and there is a need for derivatives with respect to many different design parameters, but not necessarily the same set of parameters from iteration to iteration or run to run. The quasi-procedural method and automatic differentiation can together provide a framework which is well-suited for optimization problems of this nature.

Large, complicated programs with multiple authors are most easily expressed in a modular fashion. Both quasi-procedural programming and automatic differentiation support this paradigm. Changes to the system model also create a need for modularity, as well as a mechanism for developing derivative code for new modules as quickly and easily as possible. Automatic differentiation is capable of automatically creating derivative code from function code, with minimal user intervention. Differentiating with respect to many design parameters creates a need for efficient derivative computation. The combination of automatic differentiation and the quasi-procedural method provides derivative code that is much faster than standard divided difference approximations. This speedup can be attributed to the efficiency of the hybrid mode of automatic differentiation, the capability of quasi-procedural programming to avoid redundant computation, and the synergistic effects of using the two methods together. Furthermore, derivatives computed using automatic differentiation are more accurate than divided difference approximations, which may lead to more rapid convergence for some applications. The seed matrix interface of automatic differentiation also provides a convenient mechanism for handling variation in the set of parameters with respect to which we wish to differentiate.

Automatic differentiation and the quasi-procedural method are not without their limitations. There is an overhead associated with the executive of the quasi-procedural method, and if the granularity of the modules is too fine, performance will suffer. On the other hand, if the granularity is too coarse, or if modules are highly interconnected, it may be necessary to execute every module for every iteration, effectively mimicking the behavior of a procedural program. In addition, automatic differentiation produces code which computes accurate *local* derivatives. However, if the function being differentiated is "wiggly," this information might not be useful for optimization. Derivative-free optimization techniques such as simulated annealing and genetic algorithms used in combination with the quasi-procedural method have proven effective in handling such functions [8]. Despite these minor limitations, the combination of the quasi-procedural method and automatic differentiation promises to be an effective tool for multidisciplinary optimization.

Acknowledgments

We thank Peter Gage for his contributions to the development of the combined QPM-AD framework and for his assistance in the refinement of this paper. We also thank Alan Carle for his instrumental role in the ADIFOR project.

References

[1] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[2] Christian Bischof and Paul Hovland. Using ADIFOR to compute dense and sparse Jacobians. Technical Report ANL/MCS-TM-158, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[3] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *Maple V Language Reference Manual.* Springer Verlag, New York, 1991.

[4] P. Gage and I. Kroo. Development of the quasi-procedural method for use in aircraft configuration optimization. Technical Report AIAA-92-4693, Fourth AIAA/USAF/NASA/OAI Symposium on Multidisciplinary Analysis and Optimizations, September 1992.

[5] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. User's guide for NPSOL (version 4.0): A Fortran package for nonlinear programming. Technical Report SOL 86-2, Department of Operations Research, Stanford University, Stanford, CA, 1986.

[6] Andreas Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108, Amsterdam, 1989. Kluwer Academic Publishers.

[7] I. Kroo. An interactive system for aircraft design and optimization. Technical Report AIAA-92-1190, AIAA Aircraft Design Conference, February 1992.

[8] I. Kroo, S. Altus, R. Braun, P. Gage, and I. Sobieski. Multidisciplinary optimization methods for aircraft preliminary design. Technical Report AIAA-94-2543, Fifth AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization, September 1994.

[9] I. Kroo and M. Takai. A quasi-procedural, knowledge-based system for aircraft design. Technical Report AIAA-88-4428, AIAA Aircraft Design, Systems, and Operations Meeting, September 1988.