The Application of Automatic Differentiation to Problems in Engineering Analysis *

Shirish Chinchalkar†

Abstract

Automatic differentiation is a technique of computing the derivative of a function or a subroutine written in a higher level language such as FORTRAN or C. Significant progress has been made in this field in the last few years. Here, we give a short exposition to automatic differentiation and demonstrate its applicability to several fields of engineering analysis.

1 Introduction

Engineering analysis often requires the calculation of gradients of functions written in the form of a program or a subroutine in a higher level language such as FORTRAN or C. Often, such functions are not available in an explicit form. A common practice is to approximate the gradients by finite differences. However, finite difference gradients are known to be expensive and inaccurate in several circumstances. Symbolic differentiation can be used if the function being computed is available in the form of a set of expressions. Analytical methods of differentiation have been developed in several areas, but they can become quite complex. On the other hand, automatic differentiation can be used to automatically generate code that computes the gradient of a user supplied function.

Automatic differentiation has several advantages over symbolic differentiation. First, it can operate on arbitrary programs containing do loops, if conditions, I/O statements and function calls. Second, an automatic differentiator can operate directly on code written in FORTRAN, C, or a similar higher level language; it is not necessary to ‘translate’ this code into a different language before differentiation. Third, an automatic differentiator, similar to an optimizing compiler, can generate efficient code in a higher language by recognizing common sub-expressions and reusing temporary variables. To be fair, symbolic differentiation also has some advantages. Symbolic packages such as Mathematica and Macsyma are mature commercial products, whereas development of robust automatic differentiation

---

*This research was partially supported by NSF New Technologies Program, #9108787, and by the Cornell Theory Center, which receives major funding from the National Science Foundation and IBM Corporation, with additional support from the State of New York and its Corporate Research Institutes.

†Advanced Computing Research Institute, Cornell Theory Center, Cornell University, Ithaca, NY 14853.
packages is a fairly new field. Symbolic packages can do various other things such as compute integrals, limits, and simplify expressions. Thus, automatic differentiation is not meant to replace symbolic differentiation, but to complement it.

2 Automatic differentiation

Automatic differentiation (see [9], [13], [6]) is a simple and efficient technique for computing the derivative of a function represented by means of a program written in a higher level language such as FORTRAN or C. An automatic differentiator can be used as a precompiler that can take as input a FORTRAN/C subroutine that computes a function of several independent variables and write as output a program that computes not only the function, but also the gradient of the function with respect to the independent variables. This can be done because of the following features common to FORTRAN/C programs:

- Most of the functions computed are piecewise differentiable
- Do loops, conditional statements, and goto statements only alter the sequence in which instructions are executed. Instructions are basically arithmetic operations or calls to library functions.
- At every step, only previously computed variables are used.

This suggests that the chain rule can be used repeatedly to differentiate statements in a higher level language in such a way that the derivatives are propagated along with the original variables. Derivatives computed by automatic differentiation are 'closed form'. They are accurate to machine precision, unlike derivatives computed by finite differences. They are also significantly cheaper to compute than finite difference derivatives. The reader should refer to [8] for a summary of available automatic differentiation tools.

There are two methods of automatic differentiation; the forward mode and the reverse mode. They are briefly described below. To illustrate the difference between the two methods, consider the code in figure 1, which is general enough to model any FORTRAN or C program. For convenience, it is assumed that $x_1, \ldots, x_l$ are the independent variables, $x_{l+1}, \ldots, x_M$ are the intermediate variables, and $f$ is the function being computed. The $i^{th}$ instruction assigns to $x_{\mu_i}, \mu_i \in \{1, \ldots, M\}$ the value of the function $\phi_i$ operating on an arbitrary subset $P_i$ of the variables $x_1, \ldots, x_M$.

2.1 Forward mode

In the forward mode of automatic differentiation, the gradients with respect to the independent variables are propagated using the chain rule. For each variable, $x_i$, one needs to store and update derivatives with respect to $x_j$, $j = 1 : l$. Let $x_i^j$ denote the derivative of $x_i$ with respect to $x_j$ and for convenience assume that the independent variables are not modified by the above algorithm. Since the independent variables do not depend on one another, $x_i^j = 0, i = 1 : l, j = 1 : l, j \neq i$. Figure 2 shows how the forward mode of automatic differentiation can be used to differentiate the code in figure 1.
2.2 Reverse mode

In the reverse mode of automatic differentiation, the code is first executed as usual and a trace of the execution is stored. For each variable, an ‘adjoint’ variable is defined which is used to store the derivative of the dependent variable, \( f \), with respect to that variable. Then the adjoints are updated in an order opposite to the execution of the original code. Figure 3 shows the derivative of the code in figure 1 computed using the reverse mode of automatic differentiation.

An interesting computational property of the reverse mode of automatic differentiation is that the number of floating point operations required to compute the gradient of a scalar function of several variables is bounded above by a small multiple of the number of floating point operations required to compute the function itself [6]. Specifically,

\[
T(f, \nabla f) \leq \alpha T(f)
\]

where \( \alpha \) is a constant of the order of 3. Note, that \( T(f, \nabla f) \) is independent of the number of variables, \( l \). If finite difference gradients are computed,

\[
T(f, \nabla f) \approx (l + 1)T(f)
\]

Currently available automatic differentiators that use the reverse mode require space proportional to the number of floating point operations performed. For problems requiring billions of floating point operations, such a method is impractical. However, the principles involved in the reverse mode can still be used to perform the same task manually, while making obvious improvements to the algorithm. Methods that require memory proportional to the logarithm of the number of operations are under development [7].

As a rule of thumb the forward mode should be used when the number of dependent variables is greater than the number of independent variables and the reverse mode should be used when the number of dependent variables is less than the number of independent variables. However, implementation details such as storage space can affect this decision ([2]).

Even though automatic differentiation is applicable to a wide class of programs, there are pathological cases where this technique might fail. For example, automatic differentiation, when applied to the code in figure 4, would yield a derivative of 0.0 at \( x = 5.0 \), whereas the exact answer is 10.0. This is because an automatic differentiator looks at each executable statement and differentiates it and fails to recognize that even though \( y \) is set to a constant when \( x = 5.0 \), the function is continuous overall and that its derivative is 10.0. The user thus needs to be aware of possible problems with this method (see [3], [5]).

Since the reverse mode of automatic differentiation needs to store the values of intermediate variables, it should be used to compute the gradient as well as the function value. It is not necessarily efficient for computing the gradient alone.
3 Example 1: Adjoint sensitivity analysis

We illustrate the principles of automatic differentiation with a simple, well-known example. Consider structural optimization, where the objective function is a function of the design variables, $\phi$, as well as the dependent variables, $u$, which are the nodal displacements in a finite element mesh. We consider the case where $F$ is a function of $\phi$, but $K$ is independent of $\phi$.

$$\min_{\phi} \psi \equiv \psi(\phi, u(\phi))$$

(3)

The displacements are obtained by solving the finite element system of equations

$$Ku = F$$

(4)

where $K$ is the global stiffness matrix and $F$ is the vector of externally applied nodal loads. The optimization program needs the gradients of the objective function, $\frac{\partial \psi}{\partial \phi}$. Differentiating $\psi$ with respect to $\phi$, we get

$$\frac{d\psi}{d\phi} = \frac{\partial \psi}{\partial \phi} + \frac{\partial \psi}{\partial u} \frac{\partial u}{\partial \phi}$$

(5)

where $\frac{\partial \psi}{\partial \phi}$ and $\frac{\partial \psi}{\partial u}$ are the partial derivatives of $\psi$ with respect to $\phi$ and $u$ respectively assuming $u$ to be independent of $\phi$.

When equation (4) is differentiated with respect to $\phi$, we get

$$K \frac{\partial u}{\partial \phi} = \frac{\partial F}{\partial \phi}$$

(6)

$\frac{\partial u}{\partial \phi}$ can be substituted into equation (5) to yield

$$\frac{d\psi}{d\phi} = \frac{\partial \psi}{\partial \phi} + \frac{\partial \psi}{\partial u} K^{-1} \frac{\partial F}{\partial \phi}$$

(7)

If $\phi \in R^{m \times 1}$ and $u \in R^{n \times 1}$, it is advantageous to compute the product $\frac{\partial \psi}{\partial u} K^{-1}$ first as it requires solving only one system of equations. If $K^{-1} \frac{\partial F}{\partial \phi}$ is computed first, one would need to solve $m$ systems of equations. The algorithm to compute $\frac{d\psi}{d\phi}$ is shown in figure 5.

We can derive the same algorithm by using the reverse mode of automatic differentiation. In order to do this, we must first write an algorithm for computing $\psi$ as shown in figure 6. Using the reverse mode of automatic differentiation outlined in section 2.2, we can derive an algorithm that computes both $\psi$ and $\frac{d\psi}{d\phi}$ by differentiating the algorithm in figure 6. It is shown in figure 7. It can be readily seen that it is identical to the algorithm in figure 6 followed by the algorithm in figure 5. The forward mode of automatic differentiation for this example is equivalent to computing $\frac{\partial u}{\partial \phi}$ first and then substituting it into the equation for $\frac{d\psi}{d\phi}$. As explained above, this is not efficient since this involves solving $m$ systems of equations. Moreover, from section 2.2, we know that since we are differentiating a scalar function of a vector, the reverse mode is more efficient than the forward mode.

4
4 Example 2: Groundwater remediation

The second example comes from the field of groundwater remediation. The removal of contaminants from groundwater is an important and expensive task. The contaminant concentration is controlled by pumping the contaminated groundwater out, or, in some cases, by injecting pure water into the ground. The reduction of contaminant to acceptable levels can take several years and the cost of the operation is a major concern. The cost is a combination of the pump installation cost, water treatment cost, and the pumping cost.

Mathematically, there are two distinct issues involved. First, given the geometry, hydraulic parameters, and pumping rates, one must compute the contaminant distribution as a function of space and time. This is done by a simulation involving finite element analysis over space and finite difference over time. The pumping rates need not be constant over time; in fact, by varying them in accordance with the contaminant location, significant savings in cost can be realized. Second, a suitable pumping strategy must be designed so that the contaminant concentration at each of the several ‘observation wells’ is below the required level and the cost of the cleanup operation is minimized. The problem can be formulated mathematically as an optimization problem where the sum of the pumping cost and the water treatment cost is minimized subject to the requirement that at the end of the cleanup operation, the contaminant concentration is below a specified level:

$$\min_U \psi = f(h_1, h_2, \ldots, h_N, c_1, c_2, \ldots, c_N, u_1, u_2, \ldots, u_N)$$  \hspace{1cm} (8)

subject to:

$$u_t \leq 0$$  \hspace{1cm} (9)$$

$$c_{N,i} \leq c_{\text{max}}$$  \hspace{1cm} at each observation well $i$  \hspace{1cm} (10)

where $h_i, i = 1 : N$ is the vector of nodal heads at time step $i$, $c_i, i = 1 : N$ is the vector of nodal concentrations at time step $i$, $u_i, i = 1 : N$ is the vector of nodal nodal flows at time step $i$, $N$ is the number of time steps, $n$ is the number of nodes and $m$ is the number of nodal flows. $c_{\text{max}}$ is a specified concentration level, $c_{N,i}$ denotes the concentration at node $i$ at time $N$, and

$$U = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix} \in \mathbb{R}^{N \times m}$$  \hspace{1cm} (11)

is a vector representing all pumping rates. The bounds on pumping rates ensure that water is extracted.

Calculation of the objective function in equation (8) and constraints requires the solution of the groundwater flow and contaminant transport equations (see [12], [4]). A finite element discretization using the Galerkin formulation is used to solve the governing differential
equations. The time derivatives are approximated by:

\[
\frac{d\bar{h}}{dt} = \frac{h_{t+1} - h_t}{\Delta T} \tag{12}
\]

\[
\frac{dc}{dt} = \frac{c_{t+1} - c_t}{\Delta T} \tag{13}
\]

and an implicit backward finite difference scheme is used to determine the nodal concentrations over time. We end up with systems of equations of the form

\[ A h_{t+1} = B h_t + P u_t \tag{14} \]

\[ \mathcal{N}(h_{t+1}) c_{t+1} = \mathcal{M} c_t \tag{15} \]

where \( A, B, \mathcal{M}, \) and \( \mathcal{N} \) are \( n \times n \) banded matrices and \( P \) is an \( n \times n \) sparse matrix of zeros and ones. The above equations have to be solved over time \( (t = 1 : N) \) given \( h_0 \) and \( c_0 \), the initial heads and concentrations.

Figure 8 summarizes the simulation algorithm.

The optimization program that solves equation 8 requires the derivatives of the objective function and constraints with respect to the pumping rates. We can apply automatic differentiation to the algorithm in figure 8. Figure 9 shows the derivative of the simulation algorithm using the forward mode of automatic differentiation while figure 10 shows the derivative of the simulation algorithm using the reverse mode. The reverse mode requires calculation of \( \frac{\partial \mathcal{N}}{\partial h} \). This is accomplished using the forward mode on an element-by-element basis. It is more economical to use the forward mode to compute \( \frac{\partial \mathcal{N}}{\partial h} \) because the number of dependent variables is equal to the number of entries in \( \mathcal{N} \), which is significantly greater than the number of independent variables, which equals the number of entries in \( h \).

Ahfeldt et al. [1] present two methods of computing gradients using the adjoint sensitivity scheme. The ‘left-to-right’ method is equivalent to the reverse mode of automatic differentiation, whereas the ‘right-to-left’ method is equivalent to the forward mode of automatic differentiation. The algorithms in figures 9 and 10 are, in fact, generalizations of the algorithms in [1], where they considered pumping rates that were constant over time. It is interesting to note that the same algorithms have been derived by using the purely mechanical process of automatic differentiation.

5 Example 3: Shape sensitivity analysis

Structural shape optimization methods that use the implicit differentiation approach require the sensitivity of the stiffness matrix with respect to shape parameters. The shape parameters are used to define a ‘velocity field’ which specifies the movement of every point in the domain as the parameters are changed. The displacement sensitivities using the implicit differentiation approach are given by the formula:

\[
\frac{\partial u}{\partial \phi} = K^{-1}(\frac{\partial F}{\partial \phi} - \frac{\partial K}{\partial \phi} u) \tag{16}
\]
where $\phi$ is a vector of shape parameters. The $(x,y)$ coordinates of any node in the finite element mesh are given by:

$$
x_{new} = f_x(x, y, \phi)
$$

$$
y_{new} = f_y(x, y, \phi)
$$

(17)

The stiffness matrix sensitivity can be computed using finite differences or by using an analytical method. When finite differences are used, the accuracy is low. Analytical methods can become quite cumbersome given the complexity of the function relating the stiffness to the nodal coordinates (see [14]). To automate the calculation of the sensitivities of the stiffness matrix with respect to the shape variables, the forward mode of automatic differentiation can be used.

To use automatic differentiation to calculate $\frac{\partial K}{\partial \phi}$ one must write a subroutine that computes $K$ given the shape parameters, $\phi$, and, optionally, the nodal coordinates in some default configuration. This subroutine needs to call another subroutine that computes the nodal coordinates of any given node as a function of the shape parameters, $\phi$, according to equations (17). Figure 11 shows portions of the two subroutines. The subroutine calc_KQ8() can then be input to ADIFOR, an automatic differentiator for FORTRAN programs ([2]). The resulting code computes $\frac{\partial K}{\partial \phi}$, which is used in equation (16) to compute the displacement sensitivities.

To illustrate the above procedure, we consider a simple example for which the results can be verified by hand calculations. Figure 12 shows a bar modeled by a single element subject to a uniformly distributed load of 3 per unit length. The work equivalent load is shown in the figure. The length of the bar is $L + \phi$, where $\phi$ is a scalar shape parameter. Even though we use a simple 'sizing' variable in this example, the derivative of the code in figure 11 is flexible enough to accommodate general shape variations as well, by using an appropriate, problem dependent, subroutine calc_nodal_coords(). calc_KQ8() computes the stiffness matrix as a function of $n$ shape variables $\phi(1:n)$. It is desired to compute the sensitivity of the axial displacement at the right end of the bar. A FORTRAN program that computes the stiffness matrix was written and differentiated using ADIFOR. The value of $\frac{\partial \mathbf{p}}{\partial \mathbf{E}}$ obtained is $3.0 \times 10^{-2}$, which matches the analytical value of $\frac{\partial \mathbf{p}}{\partial \mathbf{E}}$.

6 Example 4: Structural response variability

We point out another area of application of automatic differentiation: probabilistic finite element analysis. There are situations where the geometry of a structure is not known precisely. For example, in structural biomechanics, the shape of the bone is determined by Computed Tomography (CT) scans. A CT scan provides gray values of pixels in several cross-sections of the bone. Whether a pixel in a scan is bone or soft tissue is decided based on a threshold for the gray value; those pixels with gray values greater than the threshold are considered to be bone and those below it are discarded. Since the threshold is more or less arbitrary (withing a certain range), it can introduce uncertainty in the shape of
the bone. It may be considered necessary in such cases, to compute the variability of the response due to the variability in the geometry of the structure.

To illustrate this, we use a simple two-member truss, similar to the truss in [10] as shown in figure 13. The x coordinate of node 1 is a normal random variable with mean 20 and standard deviation 1. It is desired to compute the mean and standard deviation of the x displacement of node 1 using a second order approximation. A second order Taylor series approximation of the nodal displacements about the mean value of $x_1$ is given by:

$$ u = u^{(0)} + u^{(1)}(x_1 - \bar{x}_1) + \frac{1}{2} u^{(2)}(x_1 - \bar{x}_1)^2 $$

(18)

where $u^{(0)}$, $u^{(1)}$, and $u^{(2)}$ are obtained using a second order perturbation as described in [11]:

$$ u^{(0)} = K^{-1}F $$

(19)

$$ u^{(1)} = -K^{-1}(K^{(1)}u^{(0)}) $$

(20)

$$ u^{(2)} = -K^{-1}(2K^{(1)}u^{(1)} + K^{(2)}u^{(0)}) $$

(21)

where $K \in \mathbb{R}^{2 \times 2}$ is the global stiffness matrix evaluated at the mean value of $x_1$; $K^{(1)}$ and $K^{(2)}$ are the first two derivatives of $K$ with respect to $x_1$, $u^{(0)} \in \mathbb{R}^{2 \times 1}$ is the vector of nodal displacements at the mean value of $x_1$ and $u^{(1)}$ and $u^{(2)}$ are the first two derivatives of the nodal displacement vector with respect to $x_1$.

The mean and covariance of $u$ can be obtained as follows:

$$ \mathcal{E}[u] = \bar{u} = u^{(0)} + \frac{1}{2} u^{(2)} \sigma^2_{x_1} $$

(22)

$$ \text{Cov}[u] = \mathcal{E}[(u - \bar{u})(u - \bar{u})^T] $$

$$ = (u^{(1)})(u^{(1)})^T \sigma^2_{x_1} - \frac{1}{6} (u^{(2)})(u^{(2)})^T \sigma^4_{x_1} $$

(23)

where $\sigma_{x_1}$ is the standard deviation of $x_1$.

The global stiffness matrix for the problem in figure 13 is given by:

$$ K = \frac{AE}{L_1} \begin{bmatrix} c_1^2 & c_1s_1 \\ c_1s_1 & s_1^2 \end{bmatrix} + \frac{AE}{L_2} \begin{bmatrix} c_2^2 & c_2s_2 \\ c_2s_2 & s_2^2 \end{bmatrix} $$

(24)

where $A$ is the area of cross-section of the member, $E$ is the Young’s modulus and $L_1$ and $L_2$ are the lengths of the two members. The lengths are given by the formulae:

$$ L_1 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} $$

(25)

$$ L_2 = \sqrt{(x_3 - x_1)^2 + (y_3 - y_1)^2} $$

(26)

and

$$ c_1 = \cos(\theta_1) $$

(27)

$$ s_1 = \sin(\theta_1) $$

(28)

$$ c_2 = \cos(\theta_2) $$

(29)

$$ s_2 = \sin(\theta_2) $$

(30)
where

\[
\theta_1 = \tan^{-1} \left( \frac{y_2 - y_1}{x_2 - x_1} \right)
\]

\[
\theta_2 = \tan^{-1} \left( \frac{y_3 - y_1}{x_3 - x_1} \right)
\]

The calculation of \( K^{(1)} \) and \( K^{(2)} \) requires tedious algebraic manipulations if their values are required in closed form. [10] used a symbolic differentiation package called SMP to compute the derivatives. Here, we use automatic differentiation to compute the derivatives.

Currently, ADIFOR is incapable of computing higher order derivatives of a given subroutine. In principle, one could use ADIFOR to compute the first derivative and then differentiate the output again to compute the second derivative. However, currently, the output of ADIFOR is not suitable for use as input to ADIFOR. Therefore, instead of using ADIFOR, the behavior of ADIFOR was imitated by hand. Since the forward mode of differentiation used by ADIFOR is a purely mechanical process operating on individual statements only, it is a simple task even when performed manually. Using this technique, \( K^{(1)} \) and \( K^{(2)} \) were computed. The final results are:

\[
\epsilon[u_{x1}] = 1.6545192345374E-01
\]

\[
\epsilon[(u_{x1} - \epsilon[u_{x1}])^2] = 2.385986190966291E-04
\]

7 Conclusion

In this paper we have demonstrated the application of automatic differentiation to problems in engineering analysis. Several areas where automatic differentiation can be useful are identified. Since this technique can be completely automated, it can relieve the applications programmer of the burden of writing code for computing sensitivities. Even when an automatic differentiator is not used, the algorithms used in automatic differentiation can provide valuable insight into the calculation of sensitivities. We feel that this technique is promising and continuing research in development of automatic differentiators will help the engineering community.

8 Acknowledgements

This research was partially supported by NSF New Technologies Program, # 9108787, and by the Cornell Theory Center, which receives major funding from the National Science Foundation and IBM Corporation, with additional support from the State of New York and its Corporate Research Institutes. The author would like to thank Chris Bischof for his help with ADIFOR on the computers at the Advanced Computing Research Facility of the Mathematics and Computer Science Division of the Argonne National Laboratory.
References


for $i = 1 : M$
\[ x_{\mu_i} = \phi_i(\{x_{P_i}\}) \]
end
\[ f = x_M \]

Figure 1: Pseudo code representing a C or FORTRAN program

for $i = 1 : M$
\[ x_{\mu_i} = \phi_i(\{x_{P_i}\}) \]
for $j = 1 : l$
\[ t = 0 \]
for $k \in P_i$
\[ t = t + \frac{\partial \phi_i}{\partial x_k} x_j^k \]
end
\[ x_{\mu_i}^j = t \]
end
end
\[ f = x_M \]
for $i = 1 : l$
\[ \frac{\partial f}{\partial x_i} = x_M^i \]
end

Figure 2: Forward mode applied to the code in figure 1
for $i = 1:M$
\[ x_{\mu_i} = \phi_i(\{x_F\}) \]
end
\[ f = x_M \]
\[ z_1 = x_2 = \ldots = z_{M-1} = 0 \]
\[ z_M = 1 \]
for $i = M:-1:1$
\[ x_k = x_k + \frac{\partial \phi_i}{\partial x_k} x_{\mu_i} \quad \forall k \in P_i, k \neq \mu_i \]
\[ x_{\mu_i} = \frac{\partial \phi_i}{\partial x_{\mu_i}} x_{\mu_i} \]
end
for $i = 1:l$
\[ \frac{\partial f}{\partial z_i} = z_i \]
end

Figure 3: Reverse mode applied to the code in figure 1

```python
if (x == 5.0) {
    y = 25.0;
} else {
    y = x*x;
}

(a) Code that computes $y$ as a function of $x$

```python
if (x == 5.0) {
    y = 25.0;
    dydx = 0.0;
} else {
    y = x*x;
    dydx = 2.0*x;
}

(b) Code that computes $y$ and $\frac{\partial y}{\partial x}$ as a function of $x$

Figure 4: A simple program for which automatic differentiation can give the wrong answer
calculate $\frac{\partial \psi}{\partial \phi}$
calculate $\frac{\partial \psi}{\partial u}$
$\Lambda = K^{-1} \frac{\partial \psi}{\partial u}$
calculate $\frac{\partial \psi}{\partial \sigma}$
$\frac{\partial \psi}{\partial \sigma} = \frac{\partial \psi}{\partial \phi} + \Lambda^T \frac{\partial F}{\partial \phi}$

Figure 5: Algorithm for computing the gradient of the objective function using the adjoint variable method

$F = f(\phi)$
$K \rightarrow LL^T$
for $i = 1 : n$
    $w(i) = (F(i) - L(i, 1 : i - 1) * w(1 : i - 1))/L(i, i)$
end
for $i = n : 1$
    $u(i) = (w(i) - L(i + 1 : n, i)^T * u(i + 1 : n))/L(i, i)$
end
$\psi = \psi(u, \phi)$

Figure 6: Algorithm for evaluating the objective function in equation 3
\[ F = f(\phi) \]
\[ K \rightarrow LL^T \]
for \( i = 1 : n \)
\[
w(i) = (F(i) - L(i, 1 : i - 1) \ast w(1 : i - 1))/L(i, i) \]
end
for \( i = n : 1 \)
\[
u(i) = (w(i) - L(i + 1 : n, i) \ast u(i + 1 : n))/L(i, i) \]
end
\[
\psi = \psi(u, \phi) \\
\tilde{\psi} = 1; \tilde{\phi} = 0; \tilde{u} = 0; \tilde{\omega} = 0; \\
\bar{u} = u + \frac{\partial \psi}{\partial u} \tilde{\psi} \\
\bar{\phi} = \phi + \frac{\partial \psi}{\partial \phi} \tilde{\psi} \\
\bar{\psi} = 0 \\
for i = 1 : n \\
\bar{\omega}(i) = \bar{\omega}(i) + 1/L(i, i) \bar{u}(i) \\
\bar{u}(i + 1 : n) = \bar{u}(i + 1 : n) - L(i + 1 : n, i)/L(i, i) \bar{u}(i) \\
\bar{u}(i) = 0 \\
end
for i = n : 1 \\
\bar{\bar{F}}(i) = \bar{F}(i) + 1/L(i, i) \bar{\omega}(i) \\
\bar{\omega}(1 : i - 1) = L(i, 1 : i - 1) \bar{\omega}(1 : i - 1)/L(i, i) \bar{\omega}(i) \\
\bar{\omega}(i) = 0 \\
end \]
\[
\bar{\phi} = \bar{\phi} + \frac{\partial \bar{F}}{\partial \phi} \bar{\omega} \]

Figure 7: Algorithm for evaluating the gradient of the objective function in equation 3

\[ \text{read } h_0, c_0 \]
for \( i = 1 : N \)
\[
h_i = A^{-1}(Bh_{i-1} + Pu_i) \\
c_i = \mathcal{N}(h_i)^{-1} \mathcal{M}c_{i-1} \]
end
\[
\psi = f(h_1, ..., h_N, c_1, ..., c_N, u_1, ..., u_N) \]

Figure 8: Simulation algorithm
read $h_0, c_0$
for $i = 1 : N$
  $h_i = A^{-1}(B h_{i-1} + P u_i)$
  for $j = 1 : N$
    $h^j_i = A^{-1} B (h^j_{i-1} + P u^j_i)$
  end
  $c_i = N(h_i)^{-1} M c_{i-1}$
  for $j = 1 : N$
    $c^j_i = N(h_i)^{-1} (M c^j_{i-1} - N(h_i)^j c_i)$
  end
end
\psi = f(h_1, ..., h_N, c_1, ..., c_N, u_1, ..., u_N)
for $j = 1 : N$
  $\psi^j = 0$
  for $k = 1 : N$
    $\psi^j = \psi^j + \frac{\partial f}{\partial h_k} h^j_k + \frac{\partial f}{\partial c_k} c^j_k + \frac{\partial f}{\partial u_k} u^j_k$
  end
end

Figure 9: Forward mode applied to code in figure 8
read $h_0, c_0$
for $i = 1 : N$
    $h_i = A^{-1}(Bh_{i-1} + Pu_i)$
    $c_i = N(h_i)^{-1}M c_{i-1}$
end

$\psi = f(h_1, ..., h_N, c_1, ..., c_N, u_1, ..., u_N)$
$h_1 = ... = h_N = 0$
$c_1 = ... = c_N = 0$
$u_1 = ... = u_N = 0$
$\psi = 1$
for $i = 1 : N$
    $h_i = \frac{\partial f}{\partial h_i}$
    $c_i = \frac{\partial f}{\partial c_i}$
    $u_i = \frac{\partial f}{\partial u_i}$
end
for $i = N : 1$
    $c_{i-1} = c_{i-1} + M^T N^{-T}(h_i) c_i$
    $\bar{h}_i = \bar{h}_i - c_i^T \left( \frac{\partial N}{\partial e_i} \right)^T N^{-T}(h_i) c_i$
    $c_i = 0$
    $\bar{h}_{i-1} = \bar{h}_{i-1} + B^T A^{-1} \bar{h}_i$
    $u_i = u_i + P^T A^{-1} \bar{h}_i$
    $h_i = 0$
end
$h_0 = 0$
$c_0 = 0$

Figure 10: Derivative of the algorithm in figure 8 using the reverse mode of automatic differentiation. Upon termination of the algorithm, the derivatives of $\psi$ with respect to $u_i$ are in $\bar{u}_i$.
subroutine calc_KQ8(n, phi, x, y, K, yng, thick, ...)
integer n
real*8 phi(n), x(8), y(8)
real*8 K(16,16), yng, thick
:
C .... obtain nodal coordinates
   do 10 i=1,8,1
      call calc_nodal_coords(n, phi, x(i), y(i), xnew(i), ynew(i))
   10 continue
C .... compute element stiffness matrix
   :
   return
end

subroutine calc_nodal_coords(n, phi, x, y, xnew, ynew)
integer n
real*8 phi(n), x, y, xnew, ynew
:
C .... compute xnew and ynew as a function of x, y, and phi
   return
end

Figure 11: FORTRAN subroutine that computes the stiffness matrix of a Q8 element

Figure 12: Uniform tension on a single element
Figure 13: Two member truss