FINGER: A Symbolic System for Automatic Generation of Numerical Programs in Finite Element Analysis

PAUL S. WANG†
Department of Mathematical Sciences,
Kent State University, Kent, Ohio 44242, U.S.A.

(Received 7 November 1985)

FINGER is a LISP-based system to derive formulas needed in finite element analysis, and to generate FORTRAN code from these formulas. The generated programs can be used with existing, FORTRAN-based finite element analysis packages. This approach aims to replace tedious hand computations that are time consuming and error prone. The design and implementation of FINGER are presented. Techniques for generating efficient code are discussed. These include automatic intermediate expression labelling, interleaving formula derivation with code generation, exploiting symmetry through generated functions and subroutines. Current capabilities include generation of material matrices, strain-displacement matrices and stiffness matrices. FINGER contains a package, called GENTRAN, that translates symbolic formulas into FORTRAN. GENTRAN can generate functions, subroutines and entire programs. Thus, it is also of interest as a general-purpose FORTRAN code generator. Aside from the finite element application, the techniques developed and employed are useful for automatic code generation in general.

1. Introduction

Symbolic computation specialises in the exact computation with numbers, symbols, formulas, vectors, matrices and the like. Numerical computation, on the other hand, uses floating-point numbers and approximate computations to solve problems. The two computational approaches are complementary and, when combined in an integrated approach, will provide enormous computing power. In the last few years, we have been interested in using symbolic computing techniques in finite element analysis which involves extensive numerical computations.

Finite element analysis has many applications in structural mechanics, heat transfer, fluid flow, electric fields and other engineering areas. It plays a vital role in modern Computer Aided Design. Large numerical packages such as NMAP (Chang, 1980) and NASTRAN (1985) exist for finite element analysis. They provide facilities for frequently used models and cases. Only slight modifications of the “canned” computational approaches are allowed via parameter setting. Without extensive reprogramming of the formulas involved, these “canned” packages cannot be used in situations where new formulations, new materials or new solution procedures are required. Our research is on the construction of a software system to automate the derivation of formulas in finite element analysis and the generation of programs for the numerical calculation of these formulas.

† Work reported herein has been supported in part by the National Science Foundation under Grant DCR-8504824, by the Department of Energy under Grant DE-AC02-ER7602075-A013, and by the US National Aeronautics and Space Administration under Grant NAG 3-298.
Some previous work in this area can be found in Cecchi & Lami (1977), Korncoff & Fenves (1979) and Noor & Andersen (1979, 1981). The promise and potential benefit of such an approach are clearly indicated. However, it is not enough for the approach to work well on simple problems that are limited in size and complexity. Practical problems in finite-element analysis involve large expressions. Without more refined techniques, the formula derivation can become time consuming and the generated code can be very long and inefficient. Thus, several problems must be solved before this approach can become widely accepted and practiced:

(i) the derivation of symbolic formulas must be made efficient and resourceful to handle the large expressions associated with practical problems;
(ii) methods must be employed to reduce the inefficiencies that are usually associated with automatically generated code; and
(iii) the system and its user interface must be designed for ease of use by engineers and scientists who have no extensive computer experience.

The system we have constructed is called FINGER (FINite element code GEnerator). FINGER is a self-contained package written in franz LISP running under MACSYMA (1977) at Kent State University. The computer used is a VAX 11/780 under Berkeley UNIX (4.2 bsd). The design goals and the organisation of the code modules are discussed in the next two sections. Sections 4 and 5 describe formula derivation and code generation for finite element analysis. Section 6 contains a brief description of the FORTRAN code translation package, GENTRAN. Techniques of code derivation and code generation are discussed in section 7. Aside from application in finite element analysis, these techniques are useful in the general context of automatic symbolic mathematical derivation interfaced to automatic code generation.

2. Functional Specifications

From input provided by the user, either interactively or in a file, FINGER will derive finite element characteristic arrays and generate FORTRAN code based on the derived formulas. The initial system handles the isoparametric element family. Element types include 2-D, 3-D, and shell elements in linear and non-linear cases. The system allows easy extension to other finite element formulations. From a functional point of view, FINGER will

(1) assist the user in the symbolic derivation of mathematical expressions used in finite elements, in particular the various characteristic arrays;
(2) provide high-level commands for a variety of frequent and well-defined computations in finite element analysis, including linear and non-linear applications, especially for shell elements (Chang & Sawamiphakdi, 1981);
(3) allow the mode of operation to range from interactive manual control to fully automatic;
(4) generate, based on symbolic computations, FORTRAN code in a form specified by the user;
(5) automatically arrange for generated FORTRAN code to compile, link and run with FORTRAN-based finite element analysis packages such as the NMAP package (Chang, 1980);
(6) provide for easy verification of computational results and testing of the code generated.
3. Organisation of Code Modules

FINGER meets the above specifications except item (6), which is still under investigation. Its code is organised into separately compilable modules. Each module implements a different sub-task.

(a) User interface module—This contains the menu-driven, interactive user environment for FINGER. The input handling features include: free format for all input with interactive prompting showing the correct input form; editing capabilities for correcting typing errors; the capability of saving all or part of the input for later use; and the flexibility of receiving input either interactively or from a text file.

(b) Shape function module—It consists of routines to derive the shape functions from user defined parameters. Symbolic differentiation and matrix operations are involved.

(c) Strain-displacement matrix module—Routines that derive the strain-displacement matrix based on user input and the shape functions are here. As the derivation is carried out, FORTRAN code is generated into a file.

(d) Stiffness matrix module—Implemented here are special integration routines that derive the stiffness matrix. The integrand involves the material properties and strain-displacement matrices.

(e) Material matrix module—Routines here are for deriving material properties matrices that are tedious to compute. Current application is in elasto-plastic materials.

(f) Automatic labelling and symmetry handling module—This module collects functions for the automatic naming of subexpressions to avoid re-computation and re-generation of the same code. Routines for taking advantages of symmetry and the automatic generation of FORTRAN functions are also here.

(g) GENTRAN package modules—This self-contained package deals with generation of FORTRAN code from LISP structures. It includes an expression parser, a code translator, a code formatting routine, and a template file processor. This package can be used separately under MACSYMA. It has already been ported to the REDUCE (Hearn, 1985) system.

(h) Compile and link module—This is a UNIX "makefile" which directs the compilation of generated FORTRAN programs and combine the object codes with NFAP to form the executable image. This module is UNIX dependent.

(i) Demo module—This contains examples and test cases.

The code modules are well commented. There is also a demo file of examples. Other documentation, examples and a user’s manual are contained in Hui Tan’s Ph.D. dissertation (1986). Only one module (h) is dependent on UNIX. Porting to MACSYMA on other machines is relatively easy. With a little effort, it can also be ported to other LISP-based symbolic computation systems. The GENTRAN modules have already been ported to run under REDUCE. Having an idea of the modular organisation, we can proceed to examine the way FINGER works.

4. Generation of Element Characteristic Arrays

To illustrate how FINGER works, we shall look at the sequence of steps for the derivation of the strain-displacement matrix $[B]$ and element stiffness matrix $[K]$. The computation can be divided into five logical phases (Fig. 1).
4.1. PHASE I: DEFINE INPUT PARAMETERS

User input specifies the element type, the number of nodes, the nodal degrees of freedom, the displacement field interpolation polynomial, the material matrix, etc. The basic input mode is interactive with the system prompting the user at the terminal for needed input information. While the basic input mode provides flexibility, the input phase can be tedious. Thus we also provide a menu-driven mode where well-known element types together with their usual parameter values are pre-defined for user selection.

4.2. PHASE II: JACOBIAN AND [B] MATRIX COMPUTATION

The strain-displacement matrix [B] is derived from symbolically defined shape functions in this phase. Let \( n \) be the number of nodes, then \( H = (h_1, h_2, \ldots, h_n) \) is the shape function vector whose components are the \( n \) shape functions \( h_1 \) through \( h_n \). The specific expressions of the shape functions will be derived in a later phase. Here we simply compute with the symbolic names. Let \( r, s \) and \( t \) be the natural coordinates in the isoparametric formulation and \( HM \) be a matrix

\[
HM = \begin{bmatrix}
H_r \\
H_s \\
H_t
\end{bmatrix},
\]

where \( H_r \) stands for the partial derivative of \( H \) with respect to \( r \). The Jacobian \( J \) is then

\[
J = HM \cdot [x, y, z],
\]

where \( x \) stands for the column vector \( [x_1, \ldots, x_n] \) etc. Now the inverse, in full symbolic form, of \( J \) can be computed as

\[
J^{-1} = \frac{INVJ}{\det(J)}.
\]

By forming the matrix \( DH = (INVJ \cdot HM) \) we can then form the [B] matrix.

4.3. PHASE III: SHAPE FUNCTION CALCULATION

Based on the interpolation polynomials and nodal coordinates the shape function vector \( H \) is derived and expressed in terms of the natural coordinates \( r, s \) and \( t \) in the
isoparametric formulation. Thus the explicit values for all $h_i$ and all their partial derivatives with respect to $r$, $s$ and $t$, needed in $\mathbf{H}$ are computed here. FINGER also allows direct input of shape functions by the user.

4.4. PHASE IV: FORTRAN CODE GENERATION FOR $[B]$ 

A set of FORTRAN subroutines for the numerical evaluation of the strain-displacement matrix $[B]$ is generated. Several techniques for improving the efficiency of the generated code are applied here. These will be discussed in a later section. Code generated is used in combination with an existing finite element analysis package, the NFAP (Chang, 1980). This package is a large FORTRAN based system for linear and non-linear finite element analysis. It is developed and made available to us by T. Y. Chang of the University of Akron. It has been modified and made to run in FORTRAN 77 under UNIX. GENTRAN is called to translate MACSYMA LISP constructs into FORTRAN code.

4.5. PHASE V: GENERATE CODE FOR THE STIFFNESS MATRIX $[K]$ 

The inverse of the Jacobian $J$ appears in $[B]$. By keeping the inverse of $J$ as $\text{INV}J/\text{det}(J)$, the quantity $\text{det}(J)$ can be factored from $[B]$ and, denoting by $[BJ]$ the matrix $[B]$ thus reduced, we have

$$[K] = \int_{-1}^{1} \int_{-1}^{1} \int_{-1}^{1} \frac{[BJ]^T \cdot [D] \cdot [BJ]}{\det(J)} \, dr \, ds \, dr.$$ (1)

The determinant of the Jacobian involves the natural coordinates $r$, $s$ and $t$. This makes the exact integration in the above formula difficult. In this case we generate the integrand matrices and leave the integration to numerical quadrature. However, in other formulations (e.g. the hybrid-mixed formulation (Wang et al., 1986)), the integrands involve only polynomials in $r$, $s$, and $t$. In these cases we can perform the integration easily and generate efficient FORTRAN code for the stiffness matrix. This avoids numerical quadrature at run time and makes the numerical code much faster.

To avoid intermediate expression swell, the integrand matrix is not formed all at once, instead each entry is computed and integrated individually. The FORTRAN code for each entry is generated into a file immediately after the entry is computed. If the matrix is symmetric, only the upper triangular part need be computed. We use a specially designed integration program to gain speed and efficiency. The integration is organized to combine common subexpressions and produce compact and efficient FORTRAN code.

5. Material Property Matrix Generation 

Research in materials involves mathematical modelling and predicting non-linear responses of materials. The derivation of material properties matrices for use in finite element analysis is important. The mathematical derivation leading to the material properties matrix is quite tedious and error prone. Although the manipulations involved are straightforward. By automating this process, many weeks of hard computation by hand can be avoided.

The computation involves vectors, matrices, partial differentiation, matrix multiplication, etc. Expressions involved can be quite large. Thus, care must be taken to label intermediate expressions and to use symmetry relations in symbolic derivation and in generating code. The first applications of FINGER have been on elasto-plastic materials.
It is interesting to note that, using the package, we have found an error (a term missing) in the plastic matrix generally accepted in the literature. In Chen (1975), for example, equation 12.100 on p. 580 shows

$$\frac{1}{\omega} = (1 - 2v)(2J_z + 3\rho^2) + 9\nu p^2 + \frac{H(1 + \nu)(1 - 2\nu)}{E} [2J_z + 3\rho^2]^4(1 - \frac{1}{2}Bp). \quad (2)$$

Our material matrix module derived the following

$$\frac{1}{\omega} = (1 - 2v)(2J_z + 3\rho^2) + 9\nu p^2$$

$$+ \frac{H(1 + \nu)(1 - 2\nu)}{E} [2J_z + 3\rho^2 + 2(\tau_{xy}^2 + \tau_{yz}^2 + \tau_{xz}^2)]^4(1 - \frac{1}{2}Bp). \quad (3)$$

After we found the discrepancy between the equations (2) and (3), painstaking hand computation was undertaken which verified equation (3). It is conceivable that there exist numerical finite element packages that use the incorrect material matrix formula (2).

6. The FORTRAN Code Translator GENTRAN

Actual generation of FORTRAN code from symbolic expressions or constructs is performed by the GENTRAN package which is part of FINGER. This package goes beyond satisfying the needs of producing finite element code. It can serve as a general purpose FORTRAN code generator/translator. It has the capability of generating control-flow constructs and complete subroutines and functions. Large expressions can be segmented into subexpressions of manageable size. Code formatting routines enable reasonable output formatting of the generated code. Routines are provided to facilitate the interleaving of code generation and other computations. Therefore, bits and pieces of code can be generated at different times and combined to form larger pieces. For example, consider the following sequence of steps.

(1) A FORTRAN function header line is generated for the function XYZ.

(2) Declarations of formal parameters of XYZ are generated.

(3) Computation proceeds for the derivation and generation of the function body.

(3.1) Some assignment statements are generated.

(3.2) Another FORTRAN function ABC now needs to be generated (into a different output file).

(3.3) The function ABC is generated.

(3.4) More statements are generated for the function XYZ. Some such statements may call the function ABC.

(4) The generation of XYZ completes.

The flexibility afforded by GENTRAN is evident from this example. To allow the user to control finer details of code generation and to specify the exact form of certain parts of the final code, GENTRAN allows a user-supplied "template" file to guide code generation. The template file contains literal parts and variable parts. The literal parts follow regular FORTRAN syntax. The variable parts contain code derivation and generation statements. When the template file is used to guide code generation, its literal parts stay and its variable parts are replaced by generated codes. Thus, after being processed, the template file is transformed into the desired FORTRAN code (Fig. 2). With properly specified
templates, the generated code can be directly combined with existing FORTRAN code whether it is the NFAP package or something else. GENTRAN can also generate RATFOR or C code.

During his visit to Kent State University in the summer of 1983, Hans van Hulzen made available to us a REDUCE-based code optimiser (van Hulzen, 1983) which allowed us to do some experiments on optimising the generated code (Wang et al., 1984). Later, Barbara Gates visited Twente University of Technology for a year and ported GENTRAN (Gates, 1985) to REDUCE (Hearn, 1985). GENTRAN originally produced RATFOR code which was then translated into FORTRAN using the UNIX RATFOR facility. In the process of porting, the capability of generating FORTRAN 77 code directly has been added. The REDUCE-version of GENTRAN is available for distribution (Gates, 1985). User’s manuals for GENTRAN exist for both the REDUCE and MACSYMA versions.

7. Techniques for Generating Efficient Code

Previous work in employing systems such as MACSYMA for finite element computation was based on user-level programs which do not allow much control over how computations are carried out. As a result, the ability of handling realistic cases in practice is limited mostly due to intermediate expression swell.

The integration needed to compute the stiffness coefficients is an example. The MACSYMA top-level integration command is not particularly suited. Special purpose integration routines are written which, among other things, avoid expanding inner products involving coordinate vectors. This requires a new data representation and consequently new manipulative routines. It is next to impossible to do this at the user level.
The element characteristics arrays are used in the innermost loop of the iterative process for finite element analysis. Thus, the efficiency of the generated code becomes important. Let us discuss here some techniques we have applied to generate better FORTRAN code. Although these were used in finite element code generation, they are general techniques which should be helpful for other symbolic code derivation and generation applications.

7.1. AUTOMATIC EXPRESSION LABELLING

Straightforward FORTRAN code for two array entries $sk(1,1)$ and $sk(1,2)$ are shown in Fig. 3. Figure 4 contains a different version of the code for the same entries. One can see that the latter is much more efficient. The key is to automatically generate and use the labelled expressions $t_0$, $t_1$ and $t_2$ that appear repeatedly in the $sk(1,1)$ and $sk(1,2)$ computations. This means that in the mathematical derivation of these coefficients certain intermediate results should be generated with machine created labels. These results can be saved on an association list to prevent the re-computation and re-generation of the same expressions in subsequent computations. The LISP function intermediate is used for this purpose.

(defun intermediate (operand alist fn labelname labelcnt file)
  (prog (exp label arts)
    (setq ans (assoc operand (cdr alist)))
    (cond (ans (return (cdr ans))) ;; label previously defined
      (fn (setq exp (apply fn (list operand))))
      (t (setq exp operand)))
    ;; makelabel creates a new label and increments labelcnt
    (setq label (makelabel labelname labelcnt))
    ;; now generate assignment code
    (cond ((null file) (ratfor (list `(msetq) label exp)))
      (t (ratfor (list `(msetq) label exp) file)))
    ;; record operand-label pair in alist
    (setq alist (rplacd alist (cons (cons operand label) (cdr alist))))
    (return label)))

Fig. 3. Code for two stiffness coefficients.

Fig. 4. Code for two stiffness coefficients with automatically labelled expressions.

(defun intermediate (operand alist fn labelname labelcnt file)
  (prog (exp label arts)
    (setq ans (assoc operand (cdr alist)))
    (cond (ans (return (cdr ans))) ;; label previously defined
      (fn (setq exp (apply fn (list operand))))
      (t (setq exp operand)))
    ;; makelabel creates a new label and increments labelcnt
    (setq label (makelabel labelname labelcnt))
    ;; now generate assignment code
    (cond ((null file) (ratfor (list `(msetq) label exp)))
      (t (ratfor (list `(msetq) label exp) file)))
    ;; record operand-label pair in alist
    (setq alist (rplacd alist (cons (cons operand label) (cdr alist))))
    (return label)))

(defun intermediate (operand alist fn labelname labelcnt file)
  (prog (exp label arts)
    (setq ans (assoc operand (cdr alist)))
    (cond (ans (return (cdr ans))) ;; label previously defined
      (fn (setq exp (apply fn (list operand))))
      (t (setq exp operand)))
    ;; makelabel creates a new label and increments labelcnt
    (setq label (makelabel labelname labelcnt))
    ;; now generate assignment code
    (cond ((null file) (ratfor (list `(msetq) label exp)))
      (t (ratfor (list `(msetq) label exp) file)))
    ;; record operand-label pair in alist
    (setq alist (rplacd alist (cons (cons operand label) (cdr alist))))
    (return label)))
This function is called when automatic labelling is needed. Input parameters to *intermediate* are:

1. **operand:** the expression on which an operation specified by the parameter *fn* is to be performed;
2. **alist:** an association list of dotted pairs each in the form (operand, label). It is initially nil;
3. **fn:** the intended operation on the parameter *operand* (no operation if *fn* is nil);
4. **labelname:** an atom which serves as a prefix for the automatically generated label;
5. **labelcnt:** an integer count, associated with a given *labelname*, which is incremented after each new label is formed. A label is created by concatenating *labelname* with *labelcnt*;
6. **file:** a file to which any new code generated by *intermediate* will be appended.

### 7.2. Using Subroutines in Template Files to Eliminate Repeated Computations

As an example of this technique let us look at Fig. 5 where a portion of the [B] code is shown which is produced by deriving [B] directly in the LISP environment. But instead of computing [B] in LISP, we can generate the FORTRAN array "gb", corresponding to DH as shown in Fig. 6. A FORTRAN subroutine (contained in the template file) is then used to fill the array [B] by simply taking an appropriate entry of the array "gb" or zero. This requires only 1/3 of the total computation as in Fig. 5.

In forming "gb", note also that another subroutine "inner" is used to form inner products of linear arrays.

### 7.3. Using Symmetry by Generating Functions and Calls

Symmetries arise in practical problems and these symmetries are reflected in the mathematical formulation for solving the problem. Therefore techniques for taking advantage of symmetry are of great interest. For example, the expression \(x + y - z\) is related to \(x - y + z\) by symmetry, although the two cannot be regarded as identical computations. If we have a function \(F(x, y, z) = x + y - z\), then the latter expression is \(F(x, z, y)\). If \(F(x, y, z)\) is a large expression, then we can simplify the resulting code generated by first generating the function definition for \(F(x, y, z)\), then generate calls to \(F\) with the appropriate arguments wherever \(F\) or its symmetric equivalent occurs. We are not proposing an exhaustive search for symmetric patterns in large expressions. The symbolic derivation phase should preserve and use the symmetry in the given problem.

This technique greatly reduces the volume of the generated code in the finite element applications. The generated code is also more structured for reading. The price to pay is...
the additional function calls at run time which is insignificant if the functions contain non-trivial computations.

It is also possible that many calls to the same function involve the same parameter values. It is nice if repeated computation can be avoided in such cases, especially when the function is complicated. This can be achieved by generating FORTRAN functions with a memory array indexed by the values of the formal parameters. This can be done by passing integer indices to arrays containing the parameter values. The called function first uses the indices to reference its local memory for any previously computed result. If the result is there, it is returned without further computation. Otherwise, the result is computed and stored in the memory array before returning the value.

Figure 7 shows that functions $g_{ll}$, $plpl$, $qlql$ and $qlpl$ that are automatically generated with appropriate declarations in RATFOR. Then calls to these functions are generated to compute $t_0$, $r_1$ and $r_2$. The function names are program generated. These functions are generated by interleaving calls to code generation routines with the formula

\[
gb(1, 1) = \frac{(-2y_4 + r(2y_3 - 2y_4) + s(2y_2 - 2y_3) + 2y_2)}{\text{det}}
gb(1, 2) = 0
gb(1, 3) = \frac{(r(2y_4 - 2y_3) + s(2y_1 - 2y_4) + r(2y_2 - 2y_1) - 2y_2)}{\text{det}}
gb(1, 4) = 0
gb(1, 5) = \frac{(s(2y_1 - 2y_2) - 2y_2 + r(2y_1 - 2y_2) + 2y_2)}{\text{det}}
\]

Fig. 5. FORTRAN code for $[B]$. 

\[
\begin{align*}
gb(2, 1) &= \text{inner}(jinv1, hm1)/\text{det} \\
gb(2, 2) &= \text{inner}(jinv2, hm1)/\text{det} \\
gb(2, 3) &= \text{inner}(jinv2, hm2)/\text{det} \\
gb(2, 4) &= \text{inner}(jinv2, hm3)/\text{det} \\
\end{align*}
\]

Fig. 6. The array "gb".
derivation steps, resulting in great flexibility and control of the code generated. For more
details the reader is referred to Wang (1985).

7.4. OPTIMISING THE FINAL EXPRESSIONS BEFORE CODE GENERATION

Our experiments with the REDUCE code optimiser (van Hulzen, 1983) have shown that,
in addition to the above techniques, a systematic common subexpression search,
 Immediately before code output, can help reduce code size and increase code efficiency.
For more details see Wang et al. (1984). Because the code optimiser does not run under
MACSYMA yet, FINGER currently does not apply exhaustive search for common
subexpressions.

8. Conclusions

We have discussed the use of a symbolic computation system to automatically derive
and generate numerical code. The design and implementation of a software system using
this approach for generating finite element code has been presented. Several techniques
for improving the size and efficiency of the generated code are discussed. The ability to
automatically generate functions and subroutines is needed to exploit symmetry. Also
needed is the flexibility gained by interleaving code derivation and code generation steps.
Using a symbolic computation system to generate numerical code is a practical way of combining the powers of numeric computation and symbolic computation for problem solving in science and engineering. It is hoped that the techniques discussed here will find many other applications.

Dr T. Y. Chang and Dr Atef Saleeb of the Department of Civil Engineering, Akron University, actively participated in this research project, providing engineering expertise and making available the NFAP package. Dr Hans van Hulzen of the Department of Informatics, Twente University of Technology has helped by making the REDUCE code optimiser available for experimentation and by sponsoring a visit by Barbara Gates, a Kent State graduate student, to Twente University resulting in the porting of GENTRAN to REDUCE. Other graduate research assistants involved were H. Tan and P. Young.

References


