A Fortran 90 Genetic Algorithm Module for Composite Laminate Structure Design

M. T. McMahon[1], L. T. Watson[2], G. A. Soremekun[3], Z. Gürdal[4], and R. T. Haftka[5]

[1] *Graduate Research Assistant, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061-0106 USA*

[2] *Professor, Departments of Computer Science and Mathematics, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061-0106 USA*

[3] *Graduate Research Assistant, Department of Engineering Science and Mechanics, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061-0219 USA*

[4] *Professor, Department of Engineering Science and Mechanics, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061-0219 USA*

[5] *Professor, Department of Aerospace Engineering, Mechanics, and Engineering Science, University of Florida, Gainesville, FL 32611-6250 USA*

Abstract. *The design of the stacking sequence for a composite laminate involves a set of discrete variables (ply material and ply orientation), and is thus well-suited to genetic algorithms for design optimization. Such algorithms have typically been custom-designed in FORTRAN 77 to suit specific optimization problems. Fortran 90 is a modern, powerful language with features that support important programming concepts, including those used in object-oriented programming. The Fortran 90 genetic algorithm module is used to define genetic data types, the functions which use these data types, and to provide a general framework for solving composite laminate structure design problems. The language's support of abstract data types is used to build genetic structures such as populations, subpopulations, individuals, chromosomes, and genes, and these data types are combined and manipulated by module subroutines. The use of abstract data types and long variable names makes the code useful and easily understood, while dynamic memory allocation makes the module flexible enough to be used in design problems of varying size and specification.*

Keywords. Genetic Algorithms; Composite Laminate Structure Design; Fortran 90

1. Introduction

Since their formal introduction in 1975 by John Holland [1] at the University of Michigan, genetic algorithms have increasingly been applied to the solution of combinatorial problems in artificial systems. Search and optimization with genetic algorithms is a technique based on natural selection, where individuals (designs) evolve and fit individuals persist from generation to generation based on Darwinian survival of the fittest. A genetic algorithm (GA) simulates evolution by means of random genetic changes (genetic operators) and an environment (fitness function) provided by the user.

Since their introduction, GAs have been used in a variety of fields, including medicine, business, and engineering. Goldberg [2] provides an excellent introduction to the use of genetic algorithms in search and optimization. Composite laminate design is often formulated as a continuous optimization problem, with ply thickness and orientation used as design variables. In practice, ply thicknesses are often fixed and ply orientations are limited to a small set of angles by manufacturing considerations. The design optimization becomes a stacking sequence optimization using discrete (integer) values, a problem well-suited to genetic algorithms.

GAs have been used extensively in the design of composite laminates (e.g., [3]–[6]) and stiffened panels (e.g., [7], [8]) . Many computer programs have been written to implement GAs for specific composite laminate design problems, mostly in FORTRAN 77. FORTRAN is still the most widely used scientific programming language, and Fortran 90 introduces many modern programming features into the FORTRAN standard. In addition to having powerful new array features, the ability to create user-defined data types, and the ability to dynamically allocate memory at program run time, Fortran 90 is also backwards compatible with FORTRAN 77 making it possible to incorporate Fortran 90 programs into existing work.

This paper describes the implementation of a Fortran 90 framework for solving composite laminate design problems. The practices of existing FORTRAN 77 GA implementations are preserved, extended, and generalized by creating a Fortran 90 GA module containing data structures and GA operators to work with those structures. Attention is given to representing and manipulating composite structures in a way that provides for flexibility in designing GA operators and in implementing a GA optimization.

As observed by Kerrigan [9], the Fortran 90 module provides a simple but highly adaptable method to compartmentalize code. A module can replace all uses of FORTRAN 77's INCLUDE statements and COMMON blocks, while still retaining the concepts of shared declarations and globally accessible data. The GA module is used to hold the specifications for genetic data types, constants, and related subprograms, and any program which uses the module has access to these. The module enables code to be consolidated in one easily maintained package, and separates data specification from use.

The next two sections of this paper describe the composite laminate design problem and introduce the genetic algorithm. The following section describes the aspects of Fortran 90 useful in the design of the GA module GENERIC_GA, and discusses the package of GA operators developed for use with the GA module GENERIC_GA. A prototype main program is described which uses existing FORTRAN 77 analysis code.

2. Composite Laminate Design

Because of their high stiffness-to-weight and strength-to-weight ratios and their ability to be customized to the particular environment they are used in, composite materials have received substantial attention as manufacturing materials. A composite structure usually consists of one or more laminates. Laminates are comprised of stacks of thin layers of material (plies), where each layer is composed of small-diameter fibers of a particular orientation and material. The goal in designing composite laminates is to find the number of plies, along with their material type and orientation, that will yield the best performance (minimum cost or weight—the objective function) for a given set of constraints, such as buckling strength and manufacturing restrictions.

Given the variety of design variables and constraints in a composite structure, conventional design methods often prove to be impractical or expensive. Calculus (derivative) based methods of optimization are not directly applicable to the inherently discrete laminate design problem, and approximate continuous optimizations tend to find local minima. Enumerative searches, in which all designs are considered, provide optimal solutions but are prohibitively expensive; analysis of composite structures is often computationally intensive, and it is impractical to enumerate all designs. The genetic algorithm uses random choice as a tool in a directed search process, a method which has proven to be very efficient for composite design.

The target problem is a stiffened panel made from an assemblage of laminated composite plates. Each plate is composed of several plies made of one or more materials with defined properties. Each ply is made of a fiber reinforced plastic defined by specifying the material and the orientation of the fiber with respect to the plate. The ply orientation angles are limited to a discrete set of angles, and the design problem, which is combinatorial, is to find the optimum set of materials and optimum set of ply orientations. In addition, continuous design variables may be used to describe the geometry of the panel.

3. Genetic Algorithms

As succinctly stated by Goldberg [2], genetic algorithms are "search procedures based on the mechanics of natural selection and natural genetics." With GAs, the adaptive processes of natural systems are brought to bear on artificial systems by coding a design problem, represented by one or more strings (chromosomes); randomly creating an initial population of designs (individuals); and evolving these designs by the random application of genetic operators. Survival of the fittest is ensured by iteratively applying these operators to the fittest members of the previous population to yield a new population.

A stiffened panel, also referred to here as an "individual" or individual design, is composed of a number of laminated plates. For a given design, the encoding of the string represents the design variables, and the objective function represents the analysis of fitness to be performed on a given individual. As a new population is created, the objective function is applied to each new design to determine its fitness with respect to the other individuals in the population. This process is repeated for a fixed number of cycles (generations) or until there is no improvement in the best individuals over several generations.

For the problem of composite laminate design optimization, the essential elements of genetic algorithms are preserved. The GA module described in this paper uses the representation depicted in Figure 1 to describe a population. A chromosome is used to represent a laminate (laminate chromosome). In turn, the laminate is comprised of genes representing the ply orientations and material types. The number of laminates in an individual design and the maximum number of ply genes in each laminate (corresponding to an upper bound on the laminate thickness) are specified through user input. The coding of a gene is discrete, using an integer alphabet specified by user input. A chromosome is also used to represent continuous variables (geometry chromosome) which are not encoded. While continuous variables have in the past been coded with discrete strings, the geometry genes are provided to enable geometry variables to be represented by their real values. The distinction between the two types of chromosomes is made to allow the application of appropriate genetic operators to a given gene type. Use of the geometry gene type is optional. As with the laminate gene type, the number of geometry chromosomes and the number of geometry genes in each chromosome are specified by user input. An individual may be comprised of any number of laminate chromosomes and any number of geometry chromosomes.

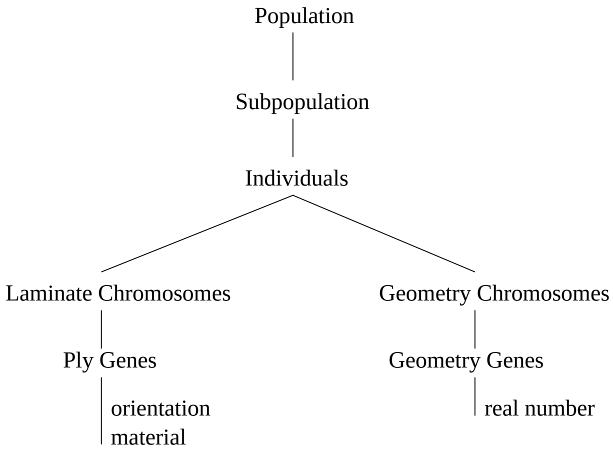

Figure 1—The structure of a population.

In describing the structure and use of the GA module, it is useful to describe a prototype main program and package of genetic operators which use the module. The prototype main program illustrates how the details of representing and manipulating module data types are hidden from the module user (abstraction and encapsulation). The user is concerned only with how to control the flow of the optimization process, not with the details of the module data structures. A prototype main program for the optimization process is provided below. Note the distinction made between user written code (program), the universal GA module that is not changed by the user (module), and a package of GA operators that could be modified if the user chooses (GA package).

1. Read user input—call (program) subroutine `read_input` .
2. Create initial parent population—call (module) subroutine `initialize_population` .
3. Analyze initial parent population—call (program) subroutine `analyze_population` .
4. Rank initial parent population—call (GA package) subroutine `rank_population` .

Repeat optimization loop 5.–9. until some termination criterion is met.

5. Create a child population—call (GA package) subroutine `apply_operators` .
6. Analyze child population— call (program) subroutine `analyze_population` .
7. Rank child population— call (GA package) subroutine `rank_population` .
8. Select new population— call (program) subroutine `elitist_selection` .
9. Rank new parent population— call (GA package) subroutine `rank_population`.

This process has applicability to GA optimizations in general. GA package operators, or user-written operators which use the GA module, are guaranteed to work with GA module data types. The GA module and GA package provide general GA subroutines for initializing, applying operators to, and ranking populations. Specific actions, such as selection and analysis, can be programmed for specific problems. The (program) analysis subroutine is the interface to the user analysis code. Previously existing or new analysis code is called from within the analysis subroutine for each member of a population. Again, using the GA module for a given design optimization ensures that a general framework for GA optimization is available, without the need to reprogram or redesign genetic data types or operators.

4. Implementing a GA Program

4.1. Why Fortran 90?

Several features of Fortran 90 enhance its suitability for implementing a GA framework for composite laminate structure design problems. The most important features include modularity (the MODULE statement), abstract data types (the TYPE statement), dynamic memory allocation (ALLOCATE and DEALLOCATE statements), and expanded array operations. Other new Fortran 90 features which ease programming and enhance program safety include explicit procedure interfaces, specification of the intended use of procedure arguments (the INTENT keyword), and operator overloading (with the INTERFACE statement). This section gives specific examples of how these features are used in the module GENERIC_GA and in user programs which make use of the module.

4.2. Modularity

When designing programs for reusability, modularity is always a design goal, and FORTRAN programmers have historically striven to keep program units separate from each other. The Fortran 90 module provides a standard means of keeping a collection of declarations and subprograms in a separate syntactic unit, the module. This encapsulation improves maintainability, and allows for reusability—the module, or parts of it, may be used in any program which can benefit from its features. The module GENERIC_GA contains the genetic data types and associated subroutines used in composite laminate design optimization. The complex interplay of INCLUDE statements and COMMON blocks previously used in FORTRAN programs to achieve this end is no longer necessary.

The functionality of the module is introduced into a user program with the Fortran 90 USE statement, as follows:

```
PROGRAM optimize
USE GENERIC_GA
⋮
END PROGRAM optimize
```

Through the USE statement, the user program gains access to the public entities of the module GENERIC_GA, with all publicly accessible data types and procedures made available to the program. Optionally, the USE statement can exclude module entities from usage with the ONLY attribute:

```
PROGRAM optimize
USE GENERIC_GA, ONLY :                        &
    population, initialize_population
⋮
END PROGRAM optimize
```

This makes only the `population` data type and the `initialize_population` module procedure available to the user program. Such use might occur if all the GENERIC_GA module procedures other than `initialize_population` were to be replaced by user written procedures. Finally, the USE statement can designate which of a variety of subprograms is associated with a single convenient name. For example, it might be desirable to change which selection scheme will be used by the program, with a single change to the code. Assuming that a variety of selection subprograms is available in the module, the desired scheme may be designated in the user program as follows:

```
PROGRAM optimize
USE GENERIC_GA, select => single_elitist
⋮
END PROGRAM optimize
```

makes the single elitist selection scheme (implemented in the procedure `single_elitist`) accessible as the procedure `select`, while

```
PROGRAM optimize
USE GENERIC_GA, select => variable_elitist
⋮
END PROGRAM optimize
```

makes the variable elitist selection scheme (implemented in the procedure `variable_elitist`) accessible as `select`. Subsequent calls of `select` will invoke the selection subprogram specified in the USE statement, obviating the need to alter the code making the call.

4.3. User-defined Data Types

Fortran 90 allows user-defined (abstract) data types to be built from intrinsic data types. In previous FORTRAN standards, an array could consist of only a single intrinsic data type. Abstract data types allow the programmer to create and name objects which include related groups of intrinsic data types and/or other abstract data types. These groups can contain any data type in any combination, as well as arrays of any data type. Thus, a data object which might have required several different arrays of different types in the past can be represented with one Fortran 90 data type. Abstract data types, when placed in modules, complete Fortran 90's support for encapsulation. Encapsulation of the GA data types and procedures yields the module GENERIC_GA. As an example, `individual` is defined in the module as shown in Figure 2. (Comments begin with an "!".)

Thus, a variable of type `individual` contains the components `geometry_array` and `laminate_array`. Each of these components, in turn, is also an abstract data type, and inherits the attributes of its type: `geometry_array` is implicitly an array of type `geometry_gene`, and `laminate_array` is implicitly an array of type `ply_gene`. This capability ultimately enables the population structure depicted in Figure 1 to be built through inheritance, yielding a data type called `population`. The use of the `pointer` attribute allows a component of the data type to point to an allocatable array. The pointer allows the fixed-size data

```
! Gene data types:
type ply_gene
    integer (KIND=small)  ::orientation
    integer (KIND=small)  ::material
end type ply_gene
!
type geometry_gene
    real (KIND=R8)  ::digit
end type geometry_gene
!
! Chromosome data types:
type laminate_chromosome
    type (ply_gene), pointer, dimension(:):: ply_array
end type laminate_chromosome
!
type geometry_chromosome
    type (geometry_gene), pointer, dimension(:):: geometry_gene_array
end type geometry_chromosome
!
! Individual (structure) data type:
type individual
    type (laminate_chromosome),pointer,dimension(:):: laminate_array
    type (geometry_chromosome),pointer,dimension(:):: geometry_array
end type individual
```

Figure 2—Data types for an individual.

type to point to an array of variable length. The `dimension` statement specifies the rank of the array (one colon gives rank one), and the array is allocated at run time as described in the following section.

4.4. Dynamic Memory Allocation

For the creation of a GA module for general use, the ability to specify an array's size at program run time is one of the most useful array features in Fortran 90. This feature enables the module to support structure designs of any size and shape. The old FORTRAN programming practice of initializing arrays to the largest size that the program might need is no longer necessary, so resources are used more efficiently. Along with the new POINTER and ALLOCATABLE variable attributes, an array can be dynamically created based on user input. This is illustrated by

```
!Allocate laminates:
type (individual) :: child
allocate                                          &
  (child%laminate_array(individual_size_lam))
```

which creates space for the child laminate array of the size specified by the laminate size variable `individual_size_lam`. The character `%` indicates a component of an abstract data type—see the definition of `individual`, of which `child` is an instance, in Figure 2. When the allocated object is no longer needed, its allocated memory may be freed:

```
!Deallocate laminates:
deallocate (child%laminate_array)
```

Another extremely useful feature is automatic arrays, which are local arrays of a subprogram whose size can be defined as a function of the subprogram arguments. Again, allocatable arrays and automatic arrays allow the module GENERIC_GA and GA operators to fully support composite laminate structures of variable size and shape.

4.5. High Level Array Operations

In addition to support for allocatable arrays, several expanded array operations have been introduced in Fortran 90. These operations simplify array handling and make programming with arrays more efficient. Fortran 90 allows operations which were previously confined to scalar data elements to be performed on entire arrays. Thus, the code

```
! Declare ply array:
integer, dimension (3,10) :: ply_array
:
:
! Initialize ply_array to zero:
ply_array = 0
```

initializes an entire array with one statement, without the need to use a nested DO-loop to initialize each individual array element. The array could also have been initialized in the declaration statement.

Fortran 90 array sections allow portions of an array to be accessed as a single object. For example, `ply_array(5:17)` is the section of `ply_array` between elements 5 and 17. The statements

```
type (laminate_chromosome) ::                &
    child, parent(2)
⋮
!Crossover from parent 1:
child%ply_array(1:cross_point) =             &
    parent(1)%ply_array(1:cross_point)
!Crossover from parent 2:
child%ply_array(cross_point+1:) =            &
    parent(2)%ply_array(cross_point+1:)
```

implement part of a crossover operation, by copying array sections from parent chromosome ply arrays to a child chromosome ply array, without a DO-loop. This is a very useful feature for the types of operations done in genetic algorithms.

4.6. Other Fortran 90 Features

A common problem in software development with earlier FORTRAN standards is procedure calls that do not match the interface specified by the procedure being called. Mismatched or missing procedure arguments remain undetected by the compiler.

A Fortran 90 module can contain explicit interfaces to external procedures. When a program unit uses a module, the interfaces in the module are visible to the program, and the compiler can ensure that procedure invocations match procedure definitions. Furthermore, interfaces to module procedures are explicit, and any invocation of a module procedure will be checked for correctness by the compiler.

It is desirable to provide interfaces to legacy analysis codes. Fortran 90 is compatible with the FORTRAN 77 standard and Fortran 90 programs can invoke existing FORTRAN 77 procedures. The interface statement can be used to ensure that the calling Fortran 90 statement is consistent with the FORTRAN 77 analysis procedure being called. The following interface block should be used in every program unit that calls the existing FORTRAN 77 two-material analysis subroutine ANLZ (used in the test program):

```
interface
!
  subroutine ANLZ(subpopulation_size,        &
                  individual,                &
                  orientation_array,         &
                  material_array,            &
                  laminate_size,             &
                  fitness_array,             &
                  geometry_array_x,          &
                  geometry_array_y)
! define parameters
! (parameters are required by
!  FORTRAN 77 legacy analysis code):
  integer, parameter ::
      subpop_maxsize=500,                    &
      laminate_maxsize=200
! define interface argument requirements:
  integer ::                                 &
      orientation_array(subpop_maxsize,      &
                        laminate_maxsize),   &
      material_array(subpop_maxsize,         &
                     laminate_maxsize),      &
      subpopulation_size,                    &
      individual,                            &
      laminate_size
    double precision ::                      &
      fitness_array(subpop_maxsize)          &
      geometry_array_x(subpop_maxsize),      &
      geometry_array_y(subpop_maxsize)
  end subroutine ANLZ
!
end interface
```

Any call to the external FORTRAN 77 analysis code ANLZ is required to conform to this explicit interface. The Fortran 90 `parameter` attribute has exactly the same meaning as the FORTRAN 77 PARAMETER statement.

Interface blocks can also be used to overload (assign a context dependent meaning) existing Fortran 90 operators (+, −, *, .EQ., etc.). FORTRAN has always allowed operators to apply to more than one data type. The test for equality (==), for example, applies to more than one type of data. In the GA module, the equals symbol is overloaded to invoke the module function `individual_compare` when its arguments are of type `individual`.

```
interface operator (==)
    module procedure individual_compare
end interface
```

This allows code like

```
use GENERIC_GA
type (individual) :: child1, child2
logical           :: unique
  ⋮
if (child1==child2) unique=.FALSE.
  ⋮
```

to be used to compare variables of type `individual`. Whereas the interface block provides explicit type checking for procedure arguments, the intended use of arguments is specified with the `intent` keyword: an argument of `intent(in)` cannot be defined or redefined in the subprogram, and an argument of `intent(out)` must be defined before it is referenced. The following declarations illustrate the use of `intent` in the GA package procedure `laminate_crossover` :

```
subroutine laminate_crossover (parent1,     &
    parent2, child1, child2)
use GENERIC_GA
!Declare argument intents:
type (individual), intent(in) ::            &
    parent1, parent2
type (individual), intent(out) ::           &
    child1, child2
  ⋮
 end subroutine laminate_crossover
```

The parent individuals can not be modified, because they are of `intent(in)`, and the child individuals must be defined in the subroutine, because they are of `intent(out)`.

With Fortran 90's interface blocks, rigourous checking of subprogram argument types is enabled, and with the specification of intent, user programs are forced to comply with the intended use of subprogram arguments. Thus, while the Fortran 90 standard provides the flexibility of modules, abstract data types, pointers, dynamic memory allocation, and powerful array functions, the subprogram interface blocks and intent checking ensure that procedures are invoked correctly and that variables are used as they were intended. All these features are brought to bear in the module GENERIC_GA, the GA operators package, and the two-material test program to yield a safe, flexible, and portable template for use in the design optimization of composite laminate structures.

5. GA Package Operators

The genetic algorithm package written for use with the module GENERIC_GA includes a flexible set of genetic operators: (1) crossover, which randomly crosses over the chromosomes from two randomly selected parents and creates new child chromosomes (crossover is applied to both discrete (laminate) and continuous (geometry) chromosomes), (2) mutation, which introduces random changes into both laminate and geometry chromosomes, (3) ply addition, which randomly adds genetic information to laminate chromosomes, (4) ply deletion, which randomly deletes genetic information from laminate chromosomes, and (5) ply permutation, which randomly reorders sequences of genes in laminate chromosomes. Because the package operators use the GA module, they are guaranteed to work with any composite laminate structure designed using the module data types.

The operators included in the package are typical of genetic algorithms, with the exception of the continuous variable crossover operator. This operator is included to allow optimization with continous variables that are not represented by discrete genes. A continuous variable (i.e., a structure geometry variable) is represented by its actual real value, and crossover between two parents is conducted as follows. Given parents p_1 and p_2, the geometry crossover first creates a randomly distributed $N(\mu, \sigma)$ real number r, where

$$\mu = \frac{p_1 + p_2}{2}, \qquad \sigma = \frac{|p_1 - p_2|}{2},$$

and then enforces physical limits by taking the child value as

$$c = \min\{\max\{r, L\}, U\},$$

where L, U are lower, upper limits on the geometry variable. Note that with probability 0.68 (the probability that a normal variate lies within one standard deviation of the mean μ), c lies between the parent values p_1 and p_2, but that c can also be well outside the segment $[p_1, p_2]$. This is essentially how evolutionary algorithms [10] work, where the value of σ itself can be adapted as the evolution proceeds.

5.1. A General Main Program

A main program `main`, which uses the module GENERIC_GA, contains three features of general interest:

1. `main` uses the module GENERIC_GA and the GA package of Fortran 90 genetic operators.
2. `main` uses a Fortran 90 analysis subroutine `analyze_population` which converts the Fortran 90 genetic data representations to representations understandable to FORTRAN 77, and then calls some previously-existing FORTRAN 77 analysis package for fitness calculations.
3. `main` calls a subroutine `read_input` which uses Fortran 90 standard NAMELIST input to read a data file. NAMELIST input enables genetic

structures of any number and type of chromosome and population size to be specified in the input file, without modification to the input subroutine.

Because it uses the GA module, `main` requires no additional programming to create and manipulate populations. Because it uses the GA package, applying genetic operators is effected with a single subroutine call. Thus, the user written main program subroutines are concerned only with functionality specific to the type of analysis being done: reading the input file, calling the external analysis code, and ranking populations.

The subroutine `analyze_population` provides an interface between the Fortran 90 program and a previously designed external analysis package written in FORTRAN 77. The subroutine can link to an arbitrary analysis package by converting the Fortran 90 representation of individuals in the population to the representations required by that package. For example, ply orientation data stored in a variable of the GA module type `population` could be converted to a double precision array `orientation_array` for the analysis code, as follows:

```
orientation_array(I, 1:laminate_size)=       &
  child_pop%subpopulation_array(subpop)      &
  %individual_array(I)                       &
  %laminate_array(L)                         &
  %ply_array(1:laminate_size)%orientation    &
```

After the required conversions are made, the external FORTRAN 77 analysis code is called from within the subroutine `analyze_population` by, for instance,

```
call ANLZ(subpopulation_size,                &
   individual,                               &
   orientation_array,                        &
   material_array,                           &
   laminate_size,                            &
   fit_array(subpop, 1: subpopulation_size)  &
   geometry_array_x,                         &
   geometry_array_y)
```

where each of the parameters in the call is a FORTRAN 77 style procedure argument, with the exception of `fit_array`, which is an array section. The variable `fit_array` uses the Fortran 90 array section feature to place the return data into the appropriate array locations, and is of the data type required by the ANLZ subroutine and specified in an INTERFACE block.

5.2. A Test Program

A test program was developed using the above guidelines for a main program. The program uses the full module GENERIC_GA and uses NAMELIST input to read genetic data. The analysis chosen for the test program is a previously existing FORTRAN 77 analysis code [6].

The test code analyzes a panel which is 36 in. long, 30 in. wide, and simply supported on all four sides. The panel can be loaded under any combination of axial and shear loads. Each ply in the panel may be made of either graphite-epoxy or Kevlar-epoxy, and can have any ply orientation between $-75°$ and $90°$, in increments of $15°$. Thus, the ply genes in a laminate chromosome contain information describing the material type and the orientation of the ply. The analysis code determines the load-handling capability of the panel by computing the margin of safety for the critical buckling load and the margin of safety for the critical ply strains. The fitness value returned by the analysis code for each panel is a measure of the panel's weight penalized by small or violated safety margins, and this value is minimized by the GA (see [6] for a complete description of the physical analysis problem). Test runs conducted to compare the performance of the test program with the performance of the original FORTRAN 77 code yielded optimum designs identical to those found by the original implementation. These runs used an elitist selection scheme and operator probabilities of crossover = 1.00, material mutation = 0.05, orientation mutation = 0.05, ply addition = 0.05, ply deletion = 0.10, and permutation = 0.25. However, program run times are slower for the Fortran 90 implementation: for the two-material problem, the Fortran 90 version ran 1250 generations of 25 individuals in 4 min. 23 sec., vs. 2 min. 55 sec. for the FORTRAN 77 version. This difference is attributable to two factors.

First, additional time overhead is introduced in the Fortran 90 implementation by the use of array sections. The high-level array operations described in this paper engender facile programming and easy to understand code, but there is a run time cost associated with copying arrays and locating the array section elements in memory. This cost is not incurred in FORTRAN 77 codes, because accessing sections of arrays is effected directly through low-level user-written code (DO loops). To quantify this, a Fortran 90 program was written in which all the elements in each row of a large two-dimensional array were shifted right five positions, inside a loop. This test code took 9.1 sec. using Fortran 90 array sections, whereas hard-coded FORTRAN 77 DO loops required only 4.6 sec. Array assignments using DO loops are more efficient than the equivalent array section assignments, at least for current Fortran 90 compilers.

Second, abstract data types make code more understandable and easy to use, but sometimes at the expense of efficiency. For example, the hierarchy depicted in Figure 1 could be represented by a set of five-dimensional arrays rather than by an abstract data type, but the usefulness, convenience, and understandability of the `population` type and the close association of this type with the GENERIC_GA module would be diminished by using the more efficient arrays. Conversely, using data types requires dereferencing of the data type elements (or fields) in order to access the desired information. For example, accessing a gene in individual `ind` in subpopulation `subpop` of a population using GENERIC_GA data types is achieved with a reference to the population data structure like

```
gene =                                        &
  population%subpopulation_array(subpop)      &
            %individual_array(ind)            &
            %laminate_array(1)%ply_array(1) &
            %orientation
```

which requires dereferencing at run time to locate the subpopulation, individual, laminate, and ply in memory. A single hard-wired array access is more efficient. To illustrate this, a test run was conducted in which a Fortran 90 variable of type `popltn` was initialized 1000 times. An equivalent set of five-dimensional FORTRAN 77 style arrays was also initialized 1000 times. The Fortran 90 code ran in 6.8 sec., while the FORTRAN 77 code ran in 2.9 sec.

Other aspects of the Fortran 90 code presented here provide the possibility for inefficiency at run time. For example, good programming practice requires the introduction of functions to replace redundant code in a program. GENERIC_GA module "shorthand" functions may be used to access elements in the population hierarchy without having to write the code for the entire hierarcy, and these functions are used in module subroutines. With the shorthand functions, the above reference to a single gene in the population data structure can be replaced with

```
gene =                                        &
 forientation(population, subpop, ind, 1, 1)
```

Although using the shorthand functions allows the programmer to access genetic data without having to write the code for the entire population hierarchy, there is a run time cost incurred by function calls. It is advisable when making use of such functions in a context where they are called many times (e.g., in the inner loop of a GA optimization) to invoke the inline compiler option. This option causes the compiler to place the code for the function inline at the point of the function call, eliminating the call and therefore eliminating the overhead associated with making that call.

The time differences described above reflect the way modern programming constructs are handled by current compiler technology; there is a tradeoff between the ease of programming in a modern language such as Fortran 90 and the run time efficiency of FORTRAN 77. Note that the analyses required for the two material test problem here were very cheap (fraction of a second per analysis), and for more expensive analyses (say one minute per analysis for a stiffened panel) the Fortran 90 overhead would be comparatively insignificant.

6. Conclusion

Genetic algorithms are used extensively for composite laminate structure design optimization in conjunction with legacy FORTRAN 77 programs. The design effort described in this paper has yielded a Fortran 90 framework for GA optimization that is flexible and general enough to be used with a variety of composite laminate structure design problems. The test implementation described in this paper yields optimal designs, matching those from the design-specific FORTRAN 77 implementation.

By using Fortran 90, the power and popularity of the FORTRAN standard is retained and modern programming practices (such as modular programming and the use of abstract data types) are brought to bear on composite laminate structure design. Placing the genetic data types and their associated functions in a module makes it easy for the user to use those components of the module that are suited to a particular design implementation. The user may build and optimize composite structures consisting of single or multiple laminates, may specify the sets of possible ply orientations and ply materials, may specify the structure geometry variables, and may specify which genetic operators to apply in a GA optimization. With this separation of data type definitions from their use in a user-written program (encapsulation), the author of an application program need not be concerned with how the module is implemented, and the module programmer is free to create a general GA module without being constrained by the specifics of a particular design problem. Similarly, the development of a GA operators package is tied to the GA module and not to any particular problem. The result is a flexible, general, reusable GA optimization tool, with some run time cost associated with using Fortran 90 data types and array features.

Finally, by adhering to the Fortran 90 standard, FORTRAN standard compatibility is ensured, along with a migration path to parallel computing architectures via the Fortran 90-based High Performance FORTRAN (HPF) standard. Future work includes expansion of the GA operators package to include an expanded choice of selection schemes, and development of a distributed parallel implementation of the software described in this paper.

References

1. Holland, J. H. (1976), *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, MI, USA.
2. Goldberg, D.E. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, USA.
3. Le Riche, R.; Haftka, R.T. (1993), "Optimization of Laminate Stacking Sequence for Buckling Load Maximization by Genetic Algorithm", *AIAA Journal,* **31**(5), 951–956.
4. Le Riche, R.; Haftka, R.T. (1994), "Improved Genetic Algorithm for Minimum Thickness Composite Laminate Design", Proceedings of International Conf. on Composite Engineering, New Orleans, LA, USA.
5. Kogiso, N.; Watson, L.T.; Gürdal, Z.; Haftka, R. T.; Nagendra, S. (1994), "Design of Composite Laminates by a Genetic Algorithm with Memory", *Mechanics of Composite Materials and Structures,* Vol. 1, 95–117.
6. Soremekun, G. A. (1997), *Genetic Algorithms for Composite Laminate Design and Optimization*, MS Thesis, Department of Engineering Mechanics, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA.
7. Nagendra, S.; Haftka, R.T.; Gürdal, Z. (1993), "Design of Blade Stiffened Composite Panels by a Genetic Algorithm Approach", Proceedings of the 34th AIAA/-ASME/AHS SDM Conf., 2418–2436, La Jolla, CA, USA.
8. Nagendra, S.; Haftka, R.T.; Gürdal, Z. (1993), "PASCO-GA : A Genetic Algorithm based Design Procedure For Stiffened Composite Panels under Stability and Strain Constraints", Tenth DOD/NASA/FAA Conf. on Fibrous Composites in Structural Design Hilton Head, SC, USA.
9. Kerrigan, J. F. (1994), *Migrating to Fortran 90*, O'Reilly, Sebastopol, CA, USA.
10. Bäck, T. (1996), *Evolutionary Algorithms in Theory and Practice*, Oxford University Press, NY, USA.

7. Appendix A—The GENERIC_GA Module

```
module GENERIC_GA
!=================================================================
!The module GENERIC_GA provides data structures for use in designing
!composite materials with genetic algorithms.
!Data structures are supplied for all entities found in a population of
!composite structures.
!
!A population is a structure defined in this module.
!Variables of type (POPLTN) inherit this structure and can be initialized
!and manipulated by module functions and subprograms.  Schematically,
!the population structure is:
!
!                       Population
!                            |
!                      Subpopulations
!                            |
!                       Individuals
!                       |         |
!    Laminate Chromosomes         Geometry Chromosomes
!            |                            |
!         Ply Genes                 Geometry Genes
!            |                            |
!            |orientation                 |real number
!            |material
!
!
!The individual attributes for a population in this module are stored
!in a structure, inherited by using type (INDIVIDUAL_ATTRIBS).  The
!module global variable INDIVIDUAL_ATTRIBUTES of type (INDIVIDUAL_ATTRIBS)
```

```
!is used to define all attributes of an individual.  INDIVIDUAL_ATTRIBUTES
!serves as a means of communicating user-defined data (e.g., from an
!input file) to the module.  Schematically, the INDIVIDUAL_ATTRIBUTES
!structure is:
!
!                  Individual Attributes
!                   |                 |
!                   |size_lam         |size_geom
!                   |                 |
!    Laminate Attributes         Geometry Attributes
!           |                             |
!           |laminate chromo size         |geometry chromo size
!           |crossover type               |upper bounds array
!           |mutation type                |lower bounds array
!           |empty plies                  |
!           |                             |prob_mutation
!           |num materials                |prob_crossover
!           |material array
!           |
!           |num orientations
!           |orientation array
!           |
!           |prob_crossover
!           |prob_mut_orientation
!           |prob_mut_material
!           |prob_ply_addition
!           |prob_ply_deletion
!           |prob_intra_ply_swap
!           |prob_permutation
!
!Variables used in defining a population:
!
! POPULATION_SIZE    specifies the number of subpopulations in a population.
! SUBPOPULATION_SIZE specifies the number of individuals in a subpopulation.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Create specifications for numbers.
!  2-digit integers
    integer, parameter :: small  =  selected_int_kind(2)
!  64-bit IEEE reals
    integer, parameter :: R8  =  selected_real_kind(15,307)
! Population variables:
    integer  ::   population_size, subpopulation_size
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Gene data types:
    type ply_gene
        integer (KIND=small)  ::orientation
        integer (KIND=small)  ::material
    end type ply_gene
    type geometry_gene
        real (KIND=R8)  ::digit
    end type geometry_gene
! Chromosome data types:
    type laminate_chromosome
```

```
        type (ply_gene), pointer, dimension(:)::   ply_array
    end type laminate_chromosome
    type geometry_chromosome
        type (geometry_gene), pointer, dimension(:)::                        &
            geometry_gene_array
    end type geometry_chromosome
! Individual (structure) data type:
    type individual
        type (laminate_chromosome),pointer,dimension(:)::                    &
            laminate_array
        type (geometry_chromosome),pointer,dimension(:)::                    &
            geometry_array
    end type individual
! Subpopulation data type:
    type subpopulation
        type (individual), pointer, dimension(:)::                           &
            individual_array
    end type subpopulation
! Population data type:
    type popltn
        type (subpopulation), pointer, dimension(:)::                        &
            subpopulation_array
    end type popltn
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Data types for individual attributes:
! Types for each laminate's attribute definitions
    type laminate_attributes
        integer :: laminate_size
        integer :: crossover_type
        integer :: mutation_type
        logical :: empty_plies
        integer :: num_materials
        integer, pointer, dimension(:)  :: material_array
        integer :: num_poss_orientations
        integer, pointer, dimension(:)  :: orientation_array
        real (KIND = R8) :: prob_crossover, prob_mut_orientation,            &
                            prob_mut_material,                               &
                            prob_ply_addition, prob_ply_deletion,            &
                            prob_intra_ply_swap, prob_permutation
    end type laminate_attributes
! Types for each geometry's attribute definitions:
    type geometry_attributes
         integer                                    :: geom_chromo_size
         real (KIND = R8), pointer, dimension(:)     :: lower_bounds_array
         real (KIND = R8), pointer, dimension(:)     :: upper_bounds_array
         real (KIND = R8) :: prob_mutation, prob_crossover
    end type geometry_attributes
! Type for individual attributes:
    type individual_attribs
        integer              :: individual_size_lam
        integer              :: individual_size_geom
        real (KIND = R8)     :: prob_inter_ply_swap
        type (laminate_attributes), pointer, dimension(:)::                  &
```

```
            laminate_definition_array
        type (geometry_attributes),pointer, dimension(:)::                                    &
            geometry_definition_array
    end type individual_attribs
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!Global variables:
    ! The variable INDIVIDUAL_ATTRIBUTES is used to specify an
    ! individual.
    ! It must be initialized by the user before this module is useable.
    type (individual_attribs) :: individual_attributes
    type (individual)         :: empty_individual
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!Interface definition:
!The following interface allows the comparison of individuals by means of a
!check for equality ('==' or '.eq').
interface operator (.eq.)
    module procedure findividual_compare
end interface
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
contains
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Inserted in the module here are shorthand functions for accessing
! genetic data.
! The shorthand functions provide a means of accessing data without the need
! to reference through the hierarchy of data types
!
! For example,
!       value = forientation(pop, 1, 2, 3, 4)
! is tantamount to
!       value = pop%subpopulation_array(1)%individual_array(2)% &
!               laminate_array(3)%ply_array(4)%orientation .
!
! Also included here are procedures for creating an individual, creating a
! population, and comparing two individuals.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
end module GENERIC_GA
```