NASA Contractor Report 196695

New Approaches to Optimization in Aerospace Conceptual Design

Peter J. Gage

Stanford University
Department of Aeronautics and Astronautics
Stanford, CA 94305

Prepared for
Ames Research Center
CONTRACT NAG2-640
March 1995

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

Acknowledgments

This work was primarily supported by NASA Ames, under Grant NAG 2-640. Funding was also provided by NASA Langley, particularly under Grant NAG 1-1494 and Grant NAG 1-1558. This support is gratefully acknowledged.

Sincere thanks are due to my adviser, Ilan Kroo, who has always been enthusiastic about this work, and ready to discuss any detail of it. I relied on his insight to clarify difficult issues, and to suggest creative approaches for this research. Valuable advice and software support have been provided by John Gallman, at NASA Ames, Bobby Braun at NASA Langley, and Ian Sobieski at Stanford. Sean Wakayama has rescued me from countless computational nightmares. Steve Morris has helped to guide the investigation of genetic optimization. I deeply appreciate the contributions of these colleagues.

Contents

List of Tables

List of Figures

Glossary

Building block: Substrings (sections of genetic string) which correlate strongly with above-average fitness. A genetic string containing a good building block should produce a design of relatively high fitness.

Calculus-based: An optimization method which uses gradient information to guide the search for improvement.

Candidate design: A particular design, or set of values for the design parameters, which is evaluated during optimization.

Constraint: An equality or inequality relation which must be satisfied by solutions to an optimization task.

Convergence: A sequence of optimizer steps converges when iteration $i + 1$ produces the same output as iteration i.

Crossover: A genetic operator which recombines sections of the genetic string from two parent designs, to produce a new design which includes features from both parents.

Deceptive: A design space where combination of building blocks from different parents, expected to produce offspring with improved performance, actually degrades fitness.

Design space: The N-dimensional space (where N is the number of design variables) in which the optimizer searches for the best design.

Design variable: One of the parameters used to describe the design, the value of which can be varied by the optimizer during search for improvement.

Encoding: A mapping between the phenotype (particular design) and genotype(genetic string). The genetic string is decoded to produce the actual design.

Environment: The conditions in the region of a particular design, characterized according to constraint activity and to the sensitivity of the objective function to changes in design variable values.

Epistasis: Coupling of design variables. Appropriate value for one design variable depends on value for another variable. Epistasis produces a nonlinear search space.

Evolution: Continuous genetic adaptation of organisms to the environment.

Evolution strategy: A search algorithm based on the operators observed in natural evolution. Minimal set of operators includes selection and mutation.

Expert system: A set of rules which formalizes the expertise of a human, represented in a computer so that the problem-solving ability of the expert can be approximated by the automated system.

Expression: The decoding of a section of a genetic string. Information contained in the genetic string, but not decoded, is not expressed.

Extended encoding: A genetic string which contains several values for each variable. Only one of the alternative values is expressed.

Feasible: A design which satisfies all constraints.

Generation: The group of candidate designs produced by an iteration of a genetic algorithm. Each iteration produces a new generation.

Genetic algorithm: A search algorithm based on the operators observed in natural evolution. Minimal set of operators includes selection, mutation and crossover.

Genetic string: A concatenated list of encoded design variables.

Global optimum: The point in design space which has the best value of the objective function while satisfying all constraints.

Gradient-based: An optimization method which uses gradient information to guide the search for improvement.

Infeasible: A design which violates one or more constraints.

Local optimum: A design point which has a better value of the objective function than all neighboring points, and satisfies all constraints.

Mating restriction: A restriction on recombination of genetic strings from different designs. Some designs are prevented from mating with each other.

Mutation: A modification of the value at a locus (or several loci) in a genetic string.

Natural selection: The survival of the relatively fit, resulting in the adaptation of a species to its environment.

Objective function: A scalar figure of merit, used to rank alternative designs.

Optimization: A formal process for seeking improvement by modifying the values of design variables.

Parameterization: The modeling of a physical design by a set of parameters. The chosen parameters are used as input variables for analysis software which estimates performance parameters for the design.

Penalty function: A function which reflects violation of a constraint. Penalty functions are appended to the objective function, so that a constrained problem is described as an unconstrained problem with a modified objective.

Population: A group of candidate designs which exist together. Members of the population compete to participate in reproduction.

Recombination: The collection of sections of genetic encoding from different parent strings, and the assembly of those sections into a new genetic string.

Repair: A constraint-handling scheme, in which the values of design variables are systematically modified to ensure constraint satisfaction prior to evaluation of the objective function.

Reproduction: Selected genetic strings participate in the creation of a new generation of candidate designs.

Roulette-wheel selection: The probability of a given design being selected for reproduction is given by the ratio of its fitness value to the sum of fitness values for the entire population.

Schema: Similarity template describing a subset of genetic strings which share identical values at specified loci in the string.

Selection: The choosing of a design from the current population. Selection criteria are related to the fitness of competing designs.

Sharing function: A function used to degrade the fitness of a candidate design when other candidate designs are very similar (nearby in design space). These functions are used to encourage the formation of separate sub-populations, by penalizing a tight cluster of many population members.

Simulated annealing: A randomized search method based on analogy with the annealing of metals. Modifications to the current design are randomly directed, but the maximum change is reduced as the number of iterations increases, according to an artificial annealing, or cooling, schedule.

Species formation: The formation of distinct subpopulations, which cluster around different local minima in the design space.

Termination criteria: Conditions for terminating optimizer search.

Topology: The geometry of the design space.

Topological optimization: Optimization of the set of parameters describing a design, rather than optimization of the shape and size of a fixed set of parameters.

Tournament selection: A selection scheme in which several population members are randomly selected to participate in a tournament. The candidate design with highest fitness wins the tournament, and is selected.

Variable-complexity parameterization: The number of parameters used to describe a design can change during optimization, which corresponds to a change in complexity of the design being described.

Variable-length string: A genetic string which can change length. This means that the amount of encoded information can change during optimization, and the complexity of candidate designs can vary.

Chapter 1

Introduction

1.1 Motivation

The standard methodology used in aerospace design is, in a broad sense, an optimization process. A parametric description of a concept is generated, and informed judgement is used to estimate appropriate initial values for the parameters. A merit function is defined, and minimum performance requirements are specified. The proposed design is evaluated by analysis, and improvement is sought by systematic modification of the parameters.

The suitability of a formal optimization approach to aerospace design has long been recognized. When Ashley surveyed aeronautical uses of optimization in 1981 [1], more than 8000 relevant journal articles, reports and dissertations were found. One conclusion from that survey was: "At the preliminary design stage, optimization has great potential as a sound way of choosing among alternative concepts". Hundreds of academic publications related to aeronautical optimization continue to be produced each year, yet multidisciplinary optimization remains underutilized by industry [2]. In this chapter, the following three issues that have restricted the effectiveness of automatic search for design improvement are discussed:

- Integration of analyses and optimizers

- The need of calculus-based optimization algorithms for accurate gradient information and a smooth search space
- Fixed parameterization of the design

These issues are discussed in the next three sections of this introduction. New approaches which address these deficiencies of existing systems are introduced in the thesis. Each new development is compared with existing methodology, by application to tasks for which optimization results have been reported in the literature. These applications include aircraft synthesis studies, interplanetary trajectory design, structural design of trusses, and aerodynamic design of lifting surfaces. They demonstrate the advanced search capability of the new system, and its suitability for diverse design studies.

1.2 Integration of Analyses and Optimization Software

Large assemblies of complex analysis modules are required for adequate assessment of proposed aeronautical configurations. The system that links them together should permit introduction of new modules to assess advanced technologies. The environment should be flexible, to allow the user to explore freely a wide range of concepts. Efficient execution of analyses is also vital, because thousands of performance evaluations are required for large optimization tasks. In this section, the necessity for a complex system is first explained, and then methods for integrating the system components and controlling their interaction are discussed.

1.2.1 Analysis Requirements

Analysis methods that predict performance with great precision are essential tools for aircraft designers. Development costs for aerospace designs are huge, and most of the outlay is committed very early in the design process (Fig. 1.1).

New aircraft designs often have only a narrow expected competitive advantage over existing alternatives. It can be financially disastrous if it is found, at the detailed design stage (or later), that the predicted performance advantage was due to inaccuracies in the preliminary assessment, and cannot be realised in the final product.

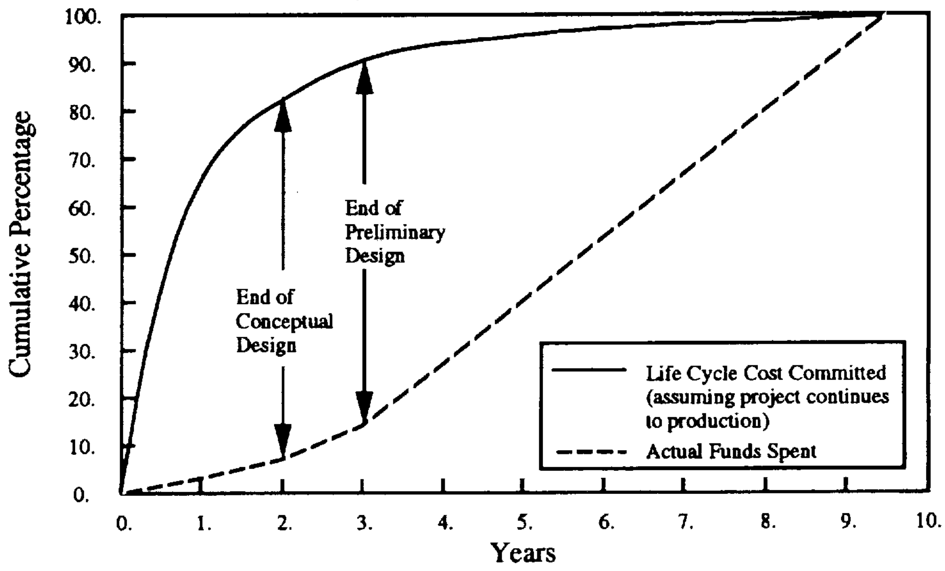


Figure 1.1: Cost committed and actual funds spent on a typical aircraft project. (From Ref. [3])

The requirement for accurate analysis is complicated by the tight coupling of aircraft components, which makes it difficult to isolate the influence of any single departure from an existing configuration. The importance of exhaustive analysis is strikingly demonstrated in a study of the joined wing concept. Gallman [4] found that a joined-wing designed for cruise alone could produce 11% better performance than a conventional configuration designed for cruise alone, but takeoff rotation constraints caused the joined-wing, optimized for the full mission, to be marginally worse than a standard configuration.

Historically, the need for rapid assessment of the entire aircraft has meant that analysis methods must be quite simple, so algebraic and statistical relations have been commonly applied [5, 6, 7]. These statistical methods incorporate empirical knowledge from previous, similar aircraft. The predictive accuracy of the tools is thereby improved, but their range of application is severely limited. Novel concepts cannot be evaluated with similar precision, and the consequent development risks are too great to justify serious investigation beyond the conceptual design phase. The complexity of analysis tools for preliminary assessment of aerospace concepts has rapidly increased, in step with the explosive growth in computational capacity available to designers. It is now possible to construct a multidisciplinary system that uses analyses based more on physical principles rather than simple statistical correlations. This broadens the range of concepts that can be accurately assessed, but increases the complexity of a system that must be integrated, yet flexible, extensible, and efficient.

1.2.2 Integration Methods

Management of the complex analysis modules used in aerospace design is a challenging task. Various approaches have been proposed, and several of them are incorporated in existing systems. A categorization of integration architectures is described here, and the merits of different methods are discussed.

Techniques for linking independent programs are categorized in four groups: close-coupled interfacing, close-coupled integration, loose-coupled interfacing and loose-coupled integration [8]. Close-coupling fixes the execution path at compilation, whereas loose-coupling allows execution to be adapted at run-time, as analysis requirements are altered. Interfacing uses intermediate files to communicate between modules, while integration uses shared memory to transfer information.

Close-coupled integration provides efficient execution, but it produces inflexible systems that are difficult to extend. Synthesis programs that are developed for a single task, and are small enough to be created and maintained by an individual, can use this approach. Modern systems for general aerospace design rely on loose-coupling to link independent analysis modules [8, 9, 10, 11]. Interfacing is used when the source code for an existing module is not available, because knowledge of the internal structure of the module is not required [9, 10]. When the source code of the analysis modules is available, integration provides a more efficient method for transferring information. It requires the insertion of additional code into each existing source module, to permit direct communication with a central database which contains all data shared between modules. Automated procedures to guide the insertion of additional code into a complex existing program should be provided [8], but they are often not included with the database system.

Although loose-coupled integration allows the execution path through the synthesis program to be modified, most executives require the user to supply the relevant procedure. The user must have detailed knowledge of the dependencies between modules, to avoid executing subroutines before their input variables have been computed by another routine. The flexibility of the system is increased by automatic generation of the necessary computational path.

Paper-Airplane [14, 15] uses a non-procedural constraint-propagation method to control execution of modules. It is limited by the need for invertible functions, and the need for user-specified *design functions*, which describe the sequence of operations that compute each database variable. The procedures generated by this executive can be inefficient, because they update all variables invalidated by a modification to the database, even when the user is interested only in a few outputs.

The quasi-procedural program architecture [16, 17, 18], employs a consistency maintenance scheme to control execution. The user simply requests the value of any variable that can be computed, and the system identifies and runs only the modules needed to make that result valid. Execution efficiency of the analyses is thereby maximized, although the consistency maintenance scheme contributes some additional overhead for the database system. A synthesis system that links algebraic and statistical analyses with a simple calculus-based numerical optimizer, and is controlled by the quasi-procedural executive, was developed and tested by Takai [16]. Impressive performance was achieved for several design tasks. Similar efficiency for large-scale design systems, and incorporation of existing complex analyses in this database framework, is reported in this thesis, to assess the general practicality of this integration method.

1.3 The Limited Scope of Calculus-Based Optimization

Most aerospace design programs use calculus-based optimization algorithms. These techniques have a well-established mathematical basis. Convergence conditions are well-specified, so termination criteria are easily checked. Calculus-based algorithms search efficiently, and they are guaranteed to locate an optimum when the search domain is appropriate.

Unfortunately, the restrictions on acceptable search domains cause severe difficulties for practical engineering applications of these algorithms. Their efficiency derives from the use of gradient information to guide the search, but accurate gradient information is often unavailable. Variation of calculated output may not be smooth due to numerical inaccuracy. Linear interpolation of tabular data, or piecewise linear functions such as the standard atmosphere, can produce sudden changes in gradient value. The gradient is not even defined for discrete-valued variables, such as number of engines or number of passengers.

Analytic derivatives are rarely available in complex engineering programs, so finite-differencing is used to estimate gradients. The cost of these estimates scales with the number of design variables, so finite-differencing becomes expensive for large problems. The limited precision of estimated derivatives can also affect the accuracy of search.

Calculus-based optimization algorithms tend to have difficulty when the initial design is far from the optimum [19], particularly when constraints are severely violated. They search from a single point, and use local information to choose the direction of improvement. They are not guaranteed to find a global optimum, and in multi-modal domains they will generally be trapped in a local optimum close to the starting point.

Some limitations of calculus-based optimization can be alleviated using expert knowledge of the domain and the search algorithm. Several investigators [10, 16, 20, 21] have introduced expert systems to help inexperienced designers to generate reasonable starting points for the numerical optimizer, or to provide advice when the optimizer terminates at an unacceptable point. These goals have been satisfactorily achieved in a number of design environments, but the degree of success depends on the quality of the rules tailored for each domain. They are similar to the statistical correlations used in simple analyses, because the rules are generally based on practical experience with existing designs and are consequently less useful when applied to new configurations. Expert systems also do not help when gradient information is not available.

Alternative search techniques are required for multi-modal problems, nonsmooth (even discontinuous) topologies, and domains in which the data are noisy. Genetic algorithms provide an alternative methodology for these circumstances [22]. There have been several aeronautical applications of genetic algorithms, and mixed results have been obtained. Significant success in control system optimization by genetic methods has been achieved by Krishnakumar and co-authors [23, 24]. Hajela [25] has performed structural optimization in nonconvex spaces, although convergence was very slow even for problems with only a few design variables, and difficulties with constraint-handling were noted. Bramlette and Cusic [26] used a genetic method for the parametric design of aircraft, but its performance was generally inferior even to simulated annealing unless a special mutation operator was introduced. Crispin [27] used a genetic algorithm for aircraft conceptual optimization, but the populations seemed to converge prematurely to a sub-optimal design. Tong [9, 10] has employed a genetic algorithm in the preliminary design of turbines, although the method was applied only when gradient techniques had stalled. Even then it simply shifted the system away from constraint boundaries, so that gradient methods could be effectively resumed. The various outcomes of these applications indicate that the use of a genetic algorithm is not universally appropriate. A more general investigation into the potential role of genetic search in aerospace conceptual design is conducted in this thesis.

1.4 Fixed Complexity Problem Formulations

The design space available to automated search algorithms is prescribed by the user's formulation of the problem. For numerical optimization, the space is defined by the parameters chosen as design variables. Parameters that describe general functional components, such as *lifting surface*, permit investigation of a wider range of concepts than parameters that refer to specific physical components, such as *wing* and *tail*. Object-oriented formulations have been introduced in recently-developed design systems [13, 15], to allow a wider range of configurations to be investigated.

Even when a general parameterization is used, standard optimization algorithms operate on a fixed set of design variables. They are able to locate the best values for those variables, but they cannot change the set. The design description has constant complexity throughout the optimization process, even if the analysis complexity is changed [28]. A sequence of separate optimization studies can be performed, with the complexity being increased for each run, but a richer algorithm would allow the description of candidate designs to be altered during optimization. Successful features can be identified in simple designs, which subsequently evolve into designs of greater complexity. This is similar to the process used by designers, who generally determine parameters such as wing area and span before choosing airfoil sections. Development of an algorithm with this capability, which considerably extends the range of design tasks handled by automated search, is reported in this work.

1.5 Thesis Outline

The current investigation of optimization for aerospace conceptual studies begins with the development of a baseline analysis management and optimization system, which incorporates the best features from programs described in this chapter. When the performance of this system has been established, new optimization methods, based on evolutionary strategies, are introduced. They are shown to significantly expand the range of design tasks that can be addressed by formal optimization approaches. Integration with the quasi-procedural executive improves flexibility of operation and efficiency of execution for these new methods.

Chapter 2 begins by briefly describing the key features of the quasi-procedural architecture. A survey of calculus-based optimizers is presented, and a sequential quadratic programming algorithm is chosen for the baseline system. Implementation details of the optimizer are modified to exploit the efficiency of the quasi-procedural method. The structure of the analysis subroutines is shown to have a strong impact on optimizer performance. Complex analyses, including a vortex-lattice aerodynamic model and a finite element structural model, are integrated into the system. A comparison is made between the quasi-procedural executive and a standard procedural method.

A standard genetic algorithm is studied in Chapter 3, because this class of optimizer has been claimed to be robust and effective in a wide range of domains. Critical evaluation of the statistical mechanism at the heart of genetic optimization reveals that these methods are sensitive to the topology of the design space and to the details of specific problem implementation. Applications to spacecraft trajectory development demonstrate the influence of the constraint-handling method on optimizer performance. Non-standard speciation and mating restriction operators increase the utility of the basic genetic algorithm.

Calculus-based methods and standard genetic algorithms all operate on a fixed set of design variables. A description of all possible elements must be explicitly included in the genetic string throughout the optimization process, even when the best parameter set is not known *a priori*. A modifed genetic algorithm is developed, to permit the number of design variables to change during optimization. The new algorithm is described in Chapter 4, and an application to a simple block-stacking problem illustrates the fundamental advantage of increasing complexity during optimization.

Standard genetic algorithms have been used by several researchers for topological design of trusses. In Chapter 5, the performance of the variable-complexity algorithm is directly compared with earlier results in this domain. The new algorithm allows the use of a simpler encoding, which greatly reduces the search effort required to reach the optimum. A calculus-based optimizer is used to identify the best values of continuous variables each time a candidate design is evaluated by the genetic algorithm.

Chapter 6 describes the application of the variable-complexity genetic algorithm to aerodynamic design of wings. Careful constraint-handling is essential to avoid premature convergence to a sub-optimal design. A new encoding of the variables in the genetic string allows adaptation in response to changing constraint activity during optimization. The algorithm successfully identifies the nature of the optimal design, but final convergence to the exact solution is better achieved by switching to a gradient-based method.

The influence of the quasi-procedural executive on the genetic optimizers is explored in Chapter 7. The flexibility of operation is enhanced, just as it was for calculus-based methods. Execution efficiency improves when the population is arranged so that similar designs are evaluated consecutively.

The key requirements for successful optimization in the conceptual design environment are summarized in Chapter 8. Emphasis is given to the new capabilities introduced in this thesis. The benefits of combining different algorithms into hybrid optimization methods are discussed. Finally, suggestions are provided for further investigation into the role of automated search in the design process.

Chapter 2

A Baseline Optimization System

The primary goal of this chapter is to develop a system that provides a convenient and efficient environment for design studies using optimization. The importance of flexible and extensible integration of analyses was discussed in the introduction, and the quasi-procedural architecture, developed by Takai and Kroo [16, 17, 18], was identified as an executive that provided these features. Improvements to the pilot version of this architecture are described in this chapter, and the enhanced executive is included in the baseline optimization system.

A survey of calculus-based optimization methods is included here. NPSOL [31], an implementation of the sequential quadratic programming algorithm, is selected as the best available software. Minor modifications to the optimizer are introduced, to exploit the efficiency of the quasi-procedural executive.

Complex analyses for the design of a medium-range commercial transport aircraft are incorporated in the system to demonstrate its capability. Existing modules, which use vortex-lattice modelling for aerodynamic prediction and finite-element analysis for structures, and were previously close-coupled, must be modified to communicate through the database. The key steps required to automate this modification are presented. The computational efficiency and overhead cost of the quasi-procedural executive are compared with standard procedural execution. This study establishes that the baseline design system provides an excellent environment for optimization.

2.1 Quasi-Procedural Method

The quasi-procedural method differs from conventional programming architectures in that the program is not strictly procedural. While a conventional program proceeds from all of the inputs to all of the outputs, a quasi-procedural program invokes only the subroutines required to produce a valid value for an output variable requested by the user. The necessary sequence of subroutines is constructed in reverse order, starting with the subroutine to be executed last, and stepping back to the subroutine to be executed first. The quasi-procedural method is named for this automated construction of an efficient computational path, which precedes procedural execution. A more detailed description of the path generation is included in Appendix A.

A design system that uses the quasi-procedural method has three parts: analysis subroutines, a database to store the parameters that define the design task, and an executive system that integrates the analyses and controls their execution. Automatic search tools and a user interface are included in the executive.

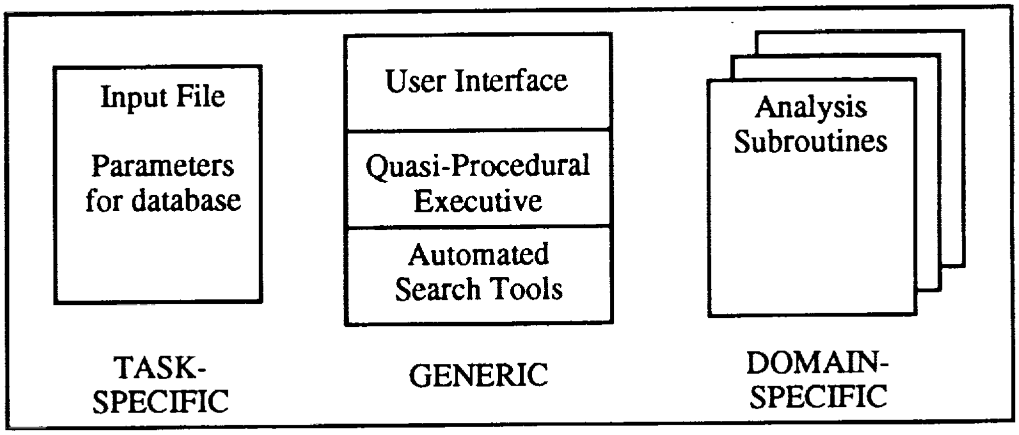


Figure 2.1: Elements of baseline system.

The analysis subroutines, or modules, are written in standard Fortran77, except that inputs and outputs are passed to and from the central database using simple calls to executive subroutines. Details of these communication subroutines, and the structure of the database, are described in Appendix A. The central database holds the inputs and outputs of the analysis subroutines. It acts as a buffer between individual modules, so that changing a single subroutine does not require modification of routines with which it shares information. It also reduces the number of interfaces between program modules. For n analysis modules, there are n interfaces with the database (one for each module). If modules are connected directly, each might be connected to all others, so the number of interfaces increases as $n(n-1)/2$. The maintenance of large collections of complex analyses is greatly simplified by communicating through a central database.

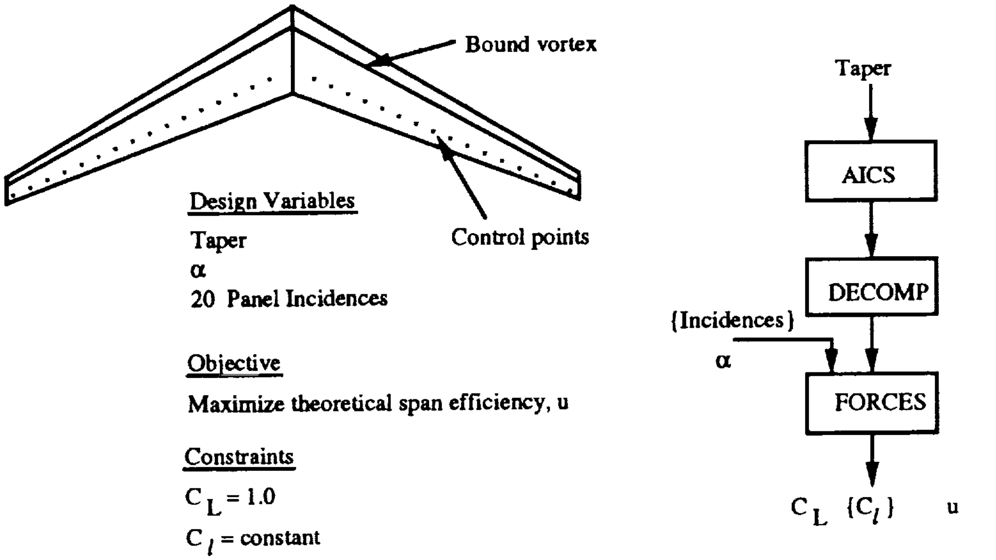
The executive controls interactions between analyses and the database, and monitors the validity of variables. A subroutine is only run when an output is required in a calculation, but the value in the database is not valid. At that point, the executive looks up the name of the analysis routine that produces the required output, and issues a command to run the routine. In contrast to a conventional program which runs top down, from all of the inputs to all of the outputs, a quasi-procedural program uses the database to build a sequence of routines that will produce the required outputs given the available inputs. In the event that no such path is available, the system will indicate the extra inputs that must be provided.

Figure 2.2(a) illustrates a simple wing design task. The goal is to maintain high theoretical span efficiency, u, at the desired lift coefficient, C_{Lreq}, while achieving a relatively uniform loading, C_l(y). The objective function is:

$$J = (-u) + (C_{Lreq} - C_L)^2 + [\Sigma(1 - \frac{C_{li}}{C_L})^2]^{\frac{1}{2}}$$

A vortex-lattice representation is used to describe the wing. Bound vortices are located at the quarter-chord line, and a control point lies at the semi-span and three-quarter-chord point of each panel.

Figure 2.2(b) indicates the computational path. The boxes represent analysis modules, with AICS calculating aerodynamic influence coefficients, DECOMP decomposing the system of linear equations, and FORCES calculating the aerodynamic forces produced by a specified incidence distribution. The computational path is indicated by the arrows that connect the boxes, and the design variables (*Taper*, *Incidence* and α [angle-of-attack]) are shown as inputs to the routines that they affect. The subroutines called by the quasi-procedural method depend on the validity of the different inputs. If a *Incidence* input is modified, only FORCES is called, but if *Taper* is modified, AICS and DECOMP must also be invoked.



(a)Vortex-lattice representation

(b) Computational path

Figure 2.2: A simple wing design problem.

This is convenient for coding, because the programmer is not responsible for calling subroutines in the correct sequence. The method is also efficient, because it updates only the outputs invalidated by changes to inputs. Any output that is unaffected by new input values is recognised as valid, so it is not recomputed. During preliminary design it is common to perform calculations repeatedly, with only small modifications to the input set for each new calculation. These operations are exactly the type most efficiently handled by the quasi-procedural system. Takai demonstrated the efficiency of the basic quasi-procedural executive for preliminary design calculations. The interaction between executive and optimizer is examined carefully here, to ensure that maximum efficiency is achieved. Calculus-based algorithms are surveyed in the next section, to identify the best available optimization method, and the chosen implementation is then modified to enhance its efficiency.

2.2 Calculus-Based Optimization

The next step in building the baseline system is the selection and incorporation of an appropriate optimizer. The mechanism of gradient-based search is described here, and the relative merits of several algorithms are discussed. The sequential quadratic programming method, which handles constraint gradient information directly, outperforms the variable metric method used by Takai, which uses penalty functions to handle constraint violations.

Numerical optimization proceeds by evaluating a sequence of points in design space, using information from previous evaluations to guide selection of new designs. Each candidate point is characterized by a number of design variables, which are selected from the input variables of the analysis system. The merit of each design is measured by an objective function and constraint functions, which are outputs of the analysis system.

Calculus-based optimizers have two distinct phases in each iteration: they select a search direction, and then perform a line search in that direction, from the current design to a new design with superior performance. This process is repeated until no direction of improvement can be found. The gradient information is used in the selection of the search direction.

A variable metric algorithm, which is recommended for general engineering design applications [19] was used previously in conjunction with the quasi-procedural architecture [16]. This method uses gradient information, and also builds an approximation to the Hessian (curvature information) as the search continues. The problem with the current implementation of this algorithm is that the constraints are handled by application of an exterior quadratic penalty function. This means that the descent function is:

$$J = J_{uncon} + PenWt \times C(x)^T C(x)$$

Here, J is the objective value, J_{uncon} is the unconstrained objective value, $PenWt$ is a user-specified weighting factor, and $C(x)$ is the constraint violation function. The efficiency of the algorithm depends strongly on the choice of the penalty weight [19]. The weight should be increased as optimization proceeds, but not too rapidly, because a current design that violates constraints may be trapped outside the feasible region, making convergence to the optimum impossible. Numerical difficulties may arise as the value of the penalty parameter increases, because the Hessian becomes ill-conditioned (the curvature near the constraint boundary becomes very large as the penalty parameter increases).

Interior penalty functions may be used to prevent the solution from being trapped in the infeasible region, but they require a feasible starting point, and do not permit the use of equality constraints. Extended interior penalty functions combine interior and exterior penalties to overcome some limitations of each penalty type. However, interior and extended interior methods require that the penalty being applied should increase as the optimization proceeds, so problems due to ill-conditioning of the Hessian arise for these methods, too [33].

The penalty method described above is an example of a transformation method, in which the constrained problem is converted to an unconstrained one. There are several standard algorithms that directly use information about the constraint gradients when calculating new search directions. These include sequential linear programming, sequential quadratic programming, feasible direction methods and reduced gradient methods. All of these methods are widely discussed in the literature [19, 35, 33, 34]. One of the most promising of these methods is sequential quadratic programming, which has been encoded in the NPSOL software package developed at Stanford Univerity's Optimization Laboratory [31]. This method searches for stationary points of a Lagrangian function.

The Lagrangian may be augmented by the addition of a penalty term, which ensures that the system searches for a minimum point. When accurate Lagrange multipliers are used, the penalty weight need not be large, which means that the problem can be better-conditioned than a formulation which relies purely on penalty functions [33].

As suggested by the name of the method, the search for the minimum of the Lagrangian involves a sequence of quadratic programming problems. Each subproblem is formed from a quadratic approximation to the cost function, and linear approximations to the constraints. This approximation is chosen because the Kuhn-Tucker first-order necessary conditions for the approximation are satisfied by the solution of a system of linear equations. The quadratic programming subproblem produces a new search direction and updated estimates for the Lagrange multipliers. A line search is performed in the direction suggested by the quadratic subproblem, and new approximations for the objective and constraints are made at the new design point. The process is continued until convergence is achieved, at which point the line search indicates that no move should be made.

The NPSOL optimizer, which implements the sequential quadratic programming algorithm, has been added to the quasi-procedural preliminary design system. It searches more efficiently than the variable metric method used previously. Convergence history for an aircraft synthesis task, with 10 variables and 5 constraints, is shown in Fig. 2.3.

NPSOL took 613 seconds, on a Macintosh IIci computer, to converge to the optimum, whereas the variable metric method required 3856 seconds. Both optimizers were started at the same point, and both converged to the same optimum. The two optimizers have different starting points on the chart because they penalize constraint violations in different ways. They reach the same optimum because there are no violations at the optimum, and hence no penalties. The sequential quadratic programming method is used for optimization in the remainder of this chapter.

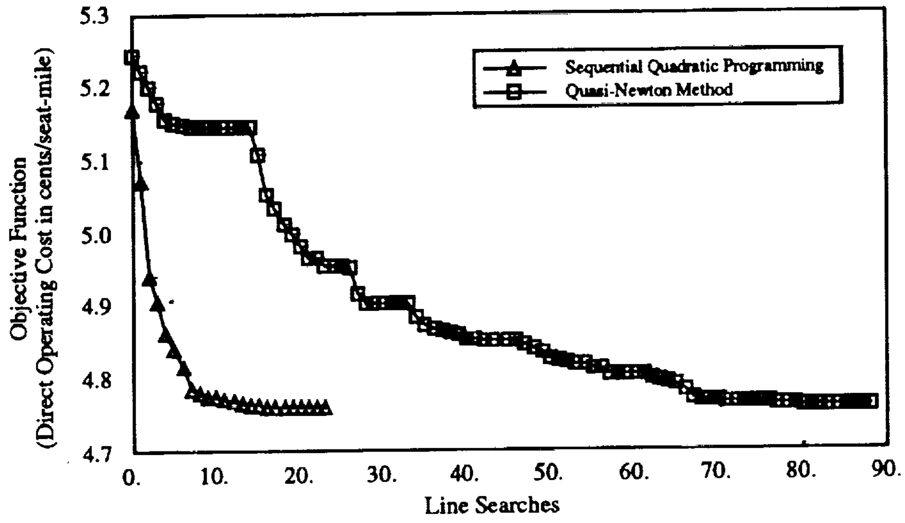


Figure 2.3: Convergence history for different calculus-based algorithms.

2.3 Controlling Calculus-Based Optimization with the Quasi-Procedural Method

The quasi-procedural method is advantageous in optimization, because the process involves repeated updates of the same output variables in response to modification of a subset of inputs. The aim here is to identify possibilities for manipulating the optimization process to exploit the quasi-procedural efficiency. Such manipulations will involve reductions in the number of inputs being modified at each step.

Recall that calculus-based optimizers have two distinct phases in each iteration: the gradient is computed to choose a direction of search, and then a line search is performed in that direction. In complex engineering problems, it is common to estimate gradient components using finite-difference approximations. Each variable is perturbed in turn, to isolate its influence on the objective and constraints. On the other hand, the line search usually requires all design variables to be modified at once. Thus, the quasi-procedural method has greater impact on the gradient calculations than on the line search. For most problems, this is the dominant computational expense.

Consider the wing design problem investigated by Takai, which was described in Section 2.1 and illustrated in Fig. 2.2. In the line search, the design variable *Taper* is modified at every function evaluation and so subroutines AICS, DECOMP, and FORCES are always called. In this phase of optimization, the quasi-procedural method does not avoid subroutine calls. However, many calls are avoided during the gradient estimation phase, because only one component of the Jacobian (the gradient with respect to *Taper*) requires the invocation of AICS and DECOMP.

The percentage of calculation time that is saved by the quasi-procedural method depends on the details of the optimization algorithm being used. Takai found a 73% improvement when using the variable metric algorithm, with central-difference estimation of the gradient. Using NPSOL [31], with forward-difference gradients, the quasi-procedural system achieved a saving of about 60%. The difference is due to a reduction in the expense of the gradient estimation, because forward-differencing is faster than central-differencing, while the cost of line-search is unchanged.

This shift in relative expense hints at an indirect influence that the quasi-procedural method exerts on the line-search. When gradient estimation is expensive, it is common to use an accurate line-search, so that fewer gradient estimations are performed. With the less expensive gradient estimation, it can be beneficial to reduce the accuracy of the line-search. The results presented in the next section indicate the effect of line-search accuracy on optimizer performance.

2.3.1 Efficient Gradient Approximation

The large benefit of the quasi-procedural method for finite-differencing motivates a closer look at how this operation is performed. The standard finite-differencing method requires two variables to be modified for the calculation of each column of the Jacobian. The new variable must be perturbed, and the variable that was previously perturbed must be restored to its original value. This restoration to the unperturbed value ensures that the change in objective value is due entirely to the modification of a single variable.

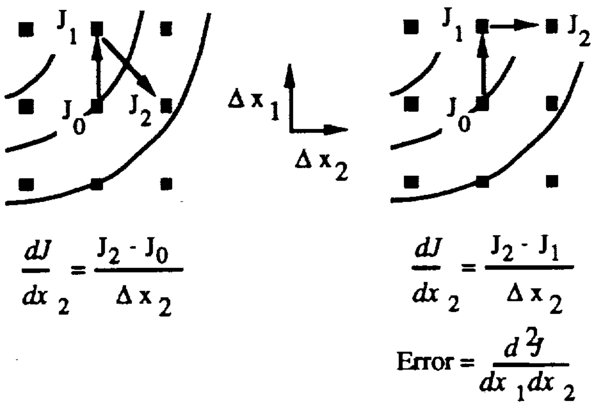

J_1
J_0
J_2
Δx_1
Δx_2
J_1
J_2
J_0
$\frac{dJ}{dx_2} = \frac{J_2 - J_0}{\Delta x_2}$
$\frac{dJ}{dx_2} = \frac{J_2 - J_1}{\Delta x_2}$
$\text{Error} = \frac{d^2J}{dx_1 dx_2}$

Figure 2.4: Standard and lazy finite-difference schemes for gradient estimation.

An alternative "lazy" method avoids the restoration of previously perturbed variables to their original value as illustrated in Fig. 2.4. The search direction calculated using this gradient estimation is not the same as the one given by the previous method, because the gradient differs by a second order term. The new method has produced successful convergence for all problems on which it has been tested, but it sometimes requires a different number of line searches to reach the optimum. The line-search accuracy should again be matched to the gradient estimation technique being used.

Figure 2.5 indicates the computations required for gradient estimation, for different finite-differencing schemes. A grid is constructed, with each column associated with a design variable, and each row associated with an analysis subroutine. The shading of each box in the grid indicates whether the analysis routine of that row needs to be executed when estimating the gradient for that column.

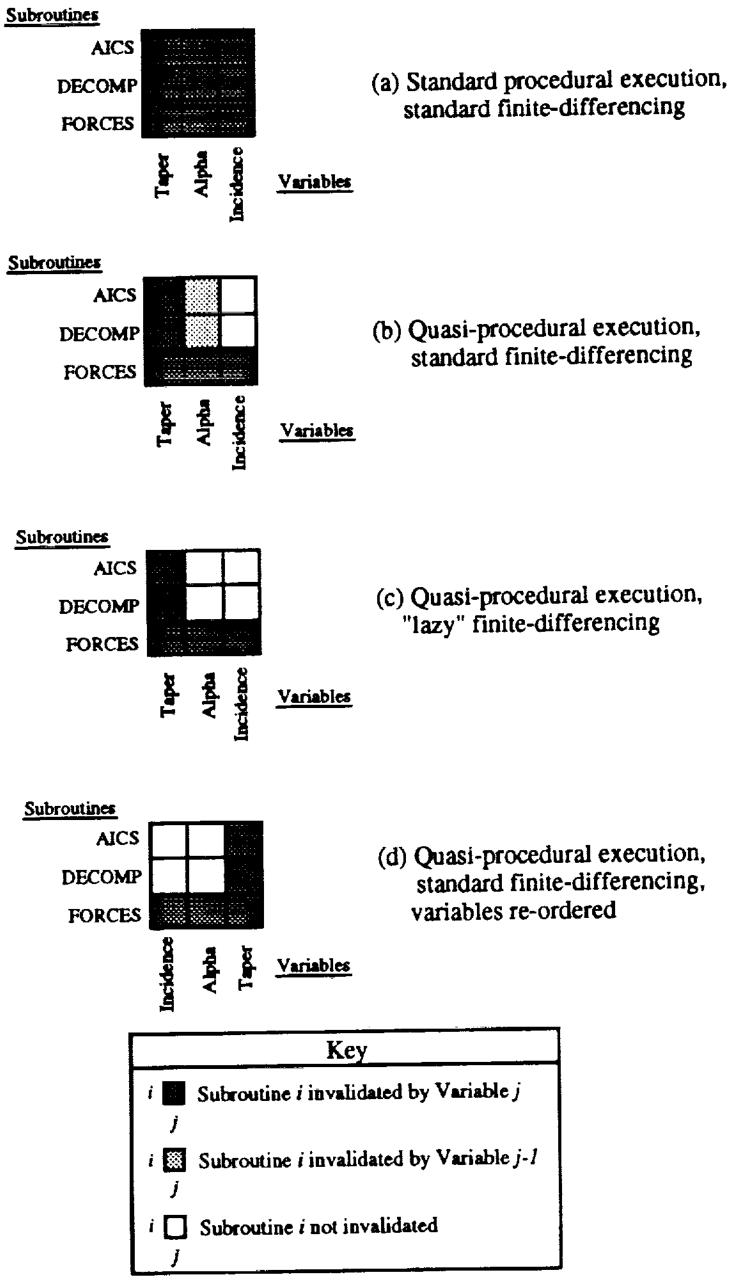
A standard procedural executive requires all analyses to be run whenever any design variable is modified. Hence, all boxes in the grid for Fig. 2.5(a), which shows the necessary computations for procedural execution, are shaded.

When the quasi-procedural executive is used, subroutines AICS and DECOMP are not run when the gradient with respect to *Incidence* is estimated. However, these subroutines are executed when the gradient with respect to α is computed, because the perturbed value of *Taper* is returned to its original value. Hence, Fig. 2.5(b) includes lightly shaded boxes to indicate that the need to run these subroutines is associated with restoration of a previously perturbed variable to its original value. Figure 2.5(c) shows that this work is avoided by the lazy finite-differencing scheme. Figure 2.5(d) demonstrates that it can alternatively be avoided by re-arranging the columns of the Jacobian. This re-ordering approach is described further in the next section.

The change in gradient estimation usually produces only a small increase in the savings provided by the quasi-procedural method, and this is true for the wing design problem. However, the effect is problem-dependent. If the incidence variables were excluded from the example, the original formulation would not be affected by the quasi-procedural method. The savings due to the new gradient estimation technique would then be a much higher percentage of the total computation. For the aircraft optimization problem considered by Takai, gradient estimation time was reduced by 10% relative to the standard estimation technique.

2.3.2 Ordering Design Variables

The lazy gradient estimation technique does not restore variables to their unperturbed values because it is costly to do so. The lazy finite-differencing scheme would not be attractive if there were no cost associated with the restoration. Consequently, it is useful to arrange the design variables to minimize the cost of restoration. This means that the previously perturbed variable should only invalidate a subset of the routines invalidated by the next variable. This invalidation information is available in the quasi-procedural method, so it is simple to sort the variables such that the set of invalidations caused by modification of the value for variable $j - 1$ is a subset of the invalidations caused by changing variable j. The result for the wing design example is shown in Fig. 2.5(d). The


Subroutines
AICS
DECOMP
FORCES
Taper
Alpha
Incidence
Variables
Key
i ■ Subroutine i invalidated by Variable j
j
i ▨ Subroutine i invalidated by Variable j-1
j
i □ Subroutine i not invalidated
j

(a) Standard procedural execution, standard finite-differencing

(b) Quasi-procedural execution, standard finite-differencing

(c) Quasi-procedural execution, "lazy" finite-differencing

(d) Quasi-procedural execution, standard finite-differencing, variables re-ordered

Figure 2.5: Quasi-procedural savings in gradient estimation for wing design task.

	# Line-Searches	Computer Time	% Saving
No Quasi-Procedural	15	18.97	-
Quasi-Procedural	15	8.62	55
Lazy Finite-Difference	17	8.39	56
Ordering of Variables	15	8.02	58

Table 2.1: Optimization efficiency with accurate line-search.

	# Line-Searches	Computer Time	% Saving
No Quasi-Procedural	18	21.12	-
Quasi-Procedural	18	8.14	61
Lazy Finite-Difference	19	7.63	64
Ordering of Variables	18	7.43	65

Table 2.2: Optimization efficiency with inaccurate line-search.

Taper variable is shifted to be modified last, because the invalidations due to modification of α are a subset of the invalidations caused by *Taper*.

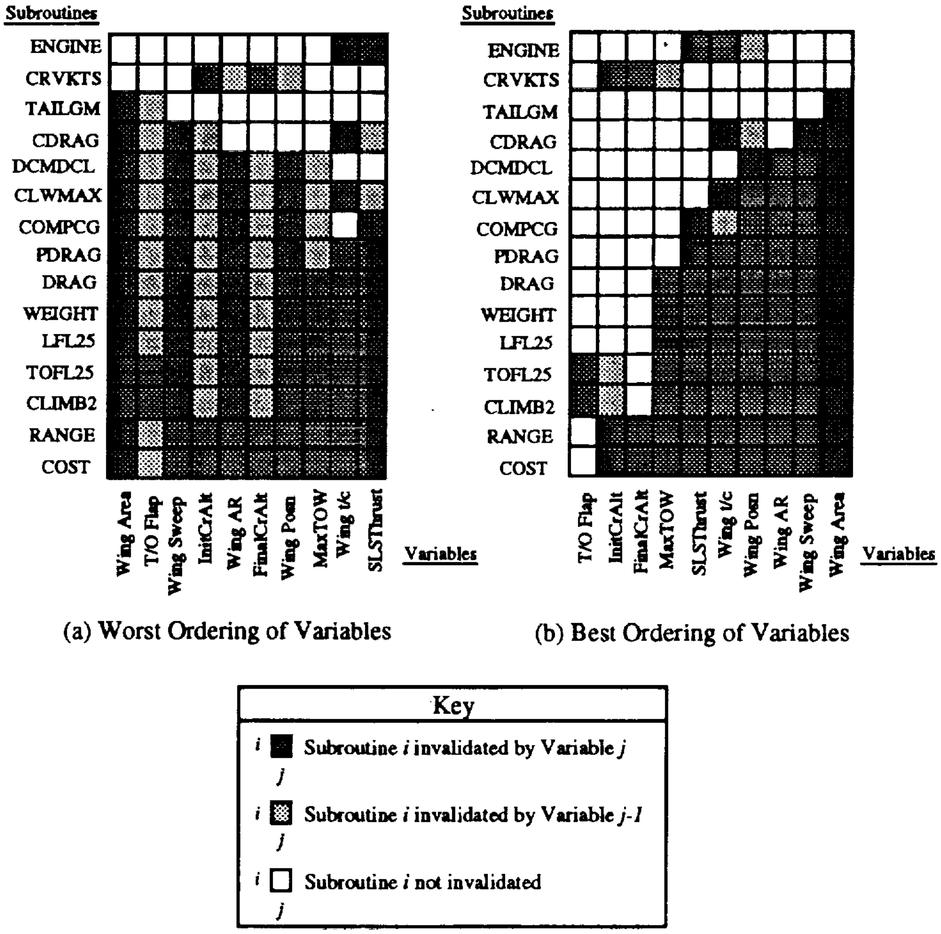
Computational savings achieved during gradient estimation are similar to those observed for the lazy finite-difference scheme. As shown in Table 2.1 and Table 2.2, overall optimizer performance for the wing design problem is greatest when this technique is employed, because the number of line-searches required to reach the optimum is not affected by the order in which the variables are arranged. The results also indicate that the accuracy of the line search should be reduced as the cost of each gradient estimation is reduced.

Figure 2.6 shows the computation required to estimate gradients for an aircraft design example. There are ten columns in the grid, each corresponding to a design variable. As in Fig. 2.5, the shading of each square indicates whether a particular subroutine must be executed to estimate gradients with respect to a given design variable. The worst ordering of design variables, shown in Fig. 2.6(a), produces the maximum number of invalidations caused by restoration of a previously perturbed variable to its original value. The best ordering, shown in Fig. 2.6(b), minimizes these invalidations. This example illustrates that the variables cannot always be arranged in a sequence that produces a strictly increasing number of invalidations for each design variable, but the number of unnecessary computations can be reduced significantly. Here, the number of subroutines executed due to restoration of previously perturbed variables is reduced from 38 to 6. There is a 10% difference in computation time for the worst-order case and the best-order case, which is the same as the saving associated with the lazy finite-difference scheme for this problem.

Variable ordering is important for optimization techniques that do not use gradient information. When using a response surface technique, the sample points should be evaluated in a sequence that minimises the number of variables modified between consecutive points. In a genetic algorithm, ordering the population so that individuals with similar characteristics are evaluated consecutively can reduce the time required for that evaluation. Ordering is less useful in this case, because it has to be repeated at each generation, as the population changes. The cost of ordering must be balanced against the cost of evaluating each individual, but when evaluation is expensive, optimizer efficiency improves. This issue is considered further in Chapter 7.

2.3.3 Removal of Iteration Loops in Analysis Subroutines

Many engineering analyses contain feedback loops, in which estimates of a certain parameter are successively refined through iteration until the value is converged to within a specified tolerance. A gradient-based optimizer requires smooth functions, so the tolerance must be small compared with the finite difference step size. This can require many iterations, particularly when convergence



(a) Worst Ordering of Variables

(b) Best Ordering of Variables

Figure 2.6: Quasi-procedural savings in gradient estimation for aircraft synthesis task.

is weak. The iteration may be replaced by an equality constraint, which must be satisfied at an optimum design point [32].

Such constraints have been implemented in the TASOP preliminary design code [12], and provided dramatic improvement in both reliability and speed of optimization, but the observed benefits for a wing optimization were problem-dependent [36]. The influence of the different program structures depends on the cost of the extra design variable compared with the savings of avoided iterations. The only iteration loop in the analysis used in the simple aircraft configuration problem is on zero fuel weight. It is strongly convergent, so only a few cheap iterations are required. Optimization with the extra variable and equality constraint is not significantly faster than optimization with the iteration loop (their relative performance varies depending on the point in design space from which the optimization is started). However, each iteration on zero-fuel-weight required substantial computation in the more complex problem described in the next section. The analysis that replaces the iteration with a design variable and constraint takes half the time of the original analysis to reach the optimum.

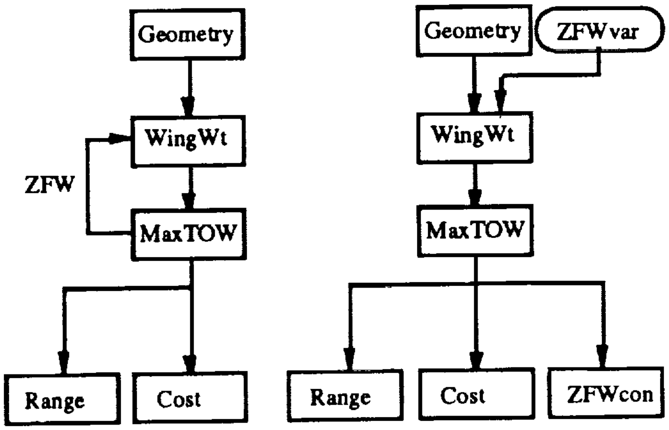


Figure 2.7: Replacement of iteration loop with design variable and constraint.

While the net benefit of loop removal depends on the sensitivity of the computed result to the estimated initial value, the role of the quasi-procedural method is clear. It minimises the work associated with the introduction of an extra design variable, and thus maximises any computational savings that might accrue.

2.4 Application of the Baseline System To a Complex Problem

The effectiveness of the quasi-procedural method was demonstrated in some simple applications by Takai [16, 17, 18]. A wing design problem (20 panel incidence variables, 1 angle of attack, 1 taper ratio) produced a 73% reduction in computation time. A complete aircraft synthesis (10 design variables, 5 constraints) required 22% less computation time than the conventional method. Preliminary studies for aircraft design may, however, involve much more complex analyses, and much larger sets of data, than these sample problems. The full-mission optimization of a transport aircraft is considered here, to check that the quasi-procedural method continues to perform well when optimization runs take minutes on a supercomputer rather than minutes on a personal computer, and the central database has thousands rather than hundreds of entries.

To handle problems of this magnitude, several small changes have been made to the quasi-procedural system. The executive routines are now written in machine-independent Fortran, which has allowed them to be run on the Cray Y-MP and several workstations. Additional routines have been introduced to control the transfer of vector and array variables between the analyses and the database. The method for assembling a computational path has been modified to improve the extensibility of systems involving many analysis routines. Previously, data files associated with each analysis routine were required for building the path, but now the user need only supply a single project file. It is now possible to provide more than one analysis to compute the same output variable, and the appropriate method for a given situation can be selected by the user. This last modification allows the optimizer to use approximate analyses along with detailed analyses, which has been shown to be advantageous in situations where the detailed analysis is computationally expensive or non-smooth [28].

2.4.1 A Sample Problem

The problem chosen for consideration is the full-mission optimization of a transport aircraft, using the method developed by Gallman and Kroo [30]. Aerodynamic loads are calculated using a vortex-lattice model. Induced drag is found by integration in the Trefftz-plane. The aircraft model includes fuselage, wing (two spanwise elements, so inboard twist may differ from outboard twist), engines and pylons, vertical tail and horizontal tail. It also captures the effects on load distribution of flap and elevator deflection. Wing and tail structural weights are computed using beam theory applied to a structural box that is sized to resist the maximum applied loads from five different flight conditions. An iteration loop is used in this sizing, because the loads and the weights depend on each other. At each iteration, the lift coefficient is calculated using an assumed weight, and the aircraft is trimmed to ensure that the load distribution is realistic. Iteration continues until the calculated weight matches the assumed weight. The twelve design variables, nine constraints and the objective function for the chosen problem are shown in Fig. 2.8. Previous optimization using these analyses has required about 200 seconds CPU on a Cray Y-MP.

2.4.2 Modification of Analyses for Quasi-Procedural Execution

While the central motivation for this work is to check the performance of the quasi-procedural system, it is also important to deal with the modification of existing procedural analyses. Difficulties in this area would have ramifications for the usefulness of the system in terms of extensibility, because new analyses must be able to interact with existing routines. Although integration of simple analyses with the quasi-procedural executive is trivial, experience with this larger code, which has 8000 executable lines and uses 20000 variables, suggests that the modification process should be automated as a precompilation step. The key issues encountered during modification are discussed below.

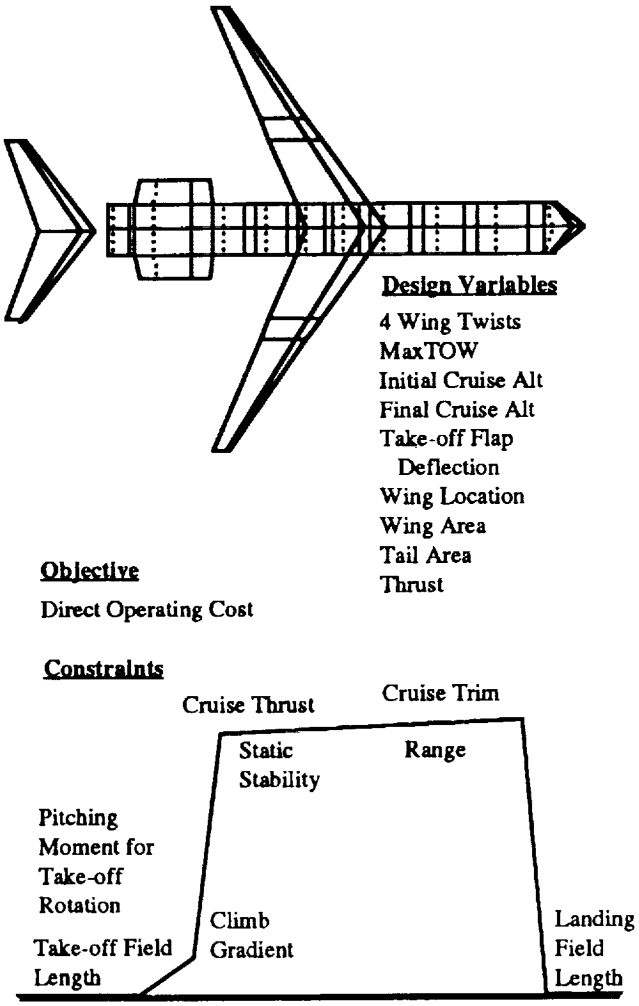
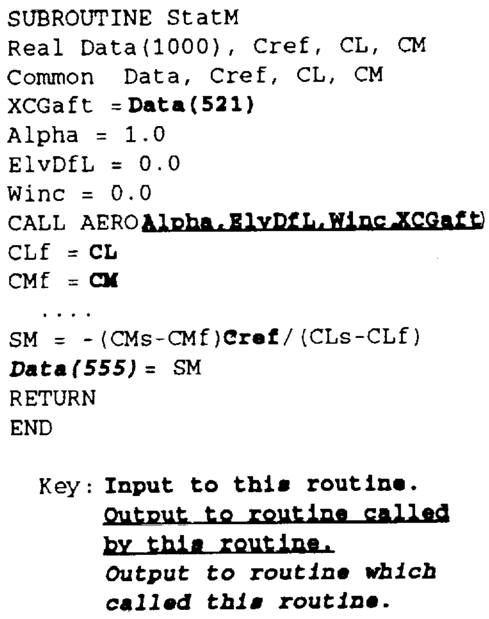


Figure 2.8: Optimization of mid-sized transport aircraft.

It is important to note that the difficulties with the original code are restricted to the passing of variables between routines. Hence, the first job is simply to identify all the inputs and outputs of each subroutine. The second task is to replace the commands that send them to (or get them from) other routines with commands that send them to (or get them from) the central database.

Programming tools can be developed to identify the inputs and outputs of analysis routines. All variables appearing in common blocks and calling statements may be communicated between subroutines. Inputs are distinguished from outputs by the way they are used in the current routine. The highlighted variables in Fig. 2.9 provide examples of the three classes of variables that must be distinguished.



```
SUBROUTINE StatM
Real Data(1000), Cref, CL, CM
Common  Data, Cref, CL, CM
XCGaft =Data(521)
Alpha = 1.0
ElvDfL = 0.0
Winc = 0.0
CALL AEROAlpha,ElvDfL,Winc,XCGaft)
CLf = CL
CMf = CM
  ....
SM = -(CMs-CMf)Cref/(CLs-CLf)
Data(555)= SM
RETURN
END

   Key: Input to this routine.
        Output to routine called
        by this routine.
        Output to routine which
        called this routine.
```

Figure 2.9: Identification of inputs and outputs.

It is often necessary to introduce new variable names for inputs or outputs, because a single name was used for several variables in the original code. This difficulty arose in the joined-wing code because a common block used in aerodynamics routines contained variable names that also appeared in a common block used in structural routines, but the names referred to different variables. The problem also occurred because angle-of-attack was both an input and an output for the subroutine used to trim the aircraft, and because elevator deflection was output by several routines. This requirement for unique names may seem to complicate the modification process, but they can easily be automatically generated. In the example of code modification presented in Fig. 2.9 and Fig. 2.10, the variable *CL*, originally in a common block, is changed to *CL-AEROo* in the database. The suffix indicates that it is an output of the AERO routine, and thereby distinguishes it from *CL* calculated in any other routine. The new names make it easier to trace each variable in the source code, which enhances the extensibility and understandability of the system.

The code that transfers information between routines is replaced by calls to quasi-procedural executive routines, with a different routine being used for each of the three different classes of variables being communicated. For inputs, a call to subroutine GET is inserted prior to the line in which the variable is first used. For outputs that are used by a routine being called by the current routine, a call to subroutine PUSH replaces the regular call. All other outputs are sent to the database by introducing a PUT command at the end of the current routine. (The distinction between PUSH and PUT is that PUSH temporarily assigns a value in the database, while PUT makes a permanent assignment.) These modifications are illustrated in Fig. 2.10. All communications to and from the analysis routine appear explicitly as calls to these executive routines, aiding the programmer in understanding the flow of information.

A project file that lists all database entries can be written while the inputs and outputs are being identified. Whenever a new variable is found, it should be added to the file. If it is output to be used by a routine that called the current routine, the current routine should be listed as the analysis routine. In Fig. 2.11, each variable carries the default value of 9999., because the source code being modified does not provide the actual value.

Hence, an automated precompilation procedure should generate source code and an input file that are suitable for use with the quasi-procedural executive

```
SUBROUTINE StatM
call GET( XCGaft ,'XaftCG' )
Alpha = 1.0
ElvDfL = 0.0
Winc = 0.0
call PUSH(alpha , 'alpha' )
call PUSH(elvdfl, 'elvdfl' )
call PUSH(winc  , 'winc-delf' )
call PUSH(xcgaft, 'cgposition )
call GET(CLf, 'CL-AEROo' )
call GET(CMf, 'CM-AEROo' )
  ....
call GET( Cref ,'Cref' )
SM = -(CMs-CMf)*Cref/(CLs-CLf)
call PUT( SM ,'StaticMargin' )
RETURN
END
```

Figure 2.10: Insertion of calls to communicate with the central database.

```
_XaftCG             9999.
_alpha              9999.
_elvdfl             9999.
_winc-delf          9999.
_cgposition         9999.
_CL-AEROo           9999.
  ANALYSIS: AERO
_CM-AEROo           9999.
  ANALYSIS: AERO
_Cref               9999.
_StaticMargin       9999.
  ANALYSIS: STATM
```

Figure 2.11: Generation of project file.

routines. It should be noted that automatically generated code will not always fully exploit the quasi-procedural method, because it is sometimes beneficial to break existing routines into smaller units. Sometimes logical tests have been included in the original code to avoid some unnecessary calls. These will not be removed automatically, and can expand the database unnecessarily.

2.4.3 Performance of the Baseline System

The performance of the quasi-procedural method has been assessed by direct comparison with the system used by Gallman and Kroo for the optimization of transport aircraft. The computations performed in each subroutine were not modified, to ensure that any differences in performance would be entirely attributable to the quasi-procedural control of subroutine execution. In some cases, however, large subroutines were split into smaller routines. This allowed the system to avoid calculation of outputs that were not specifically requested. Both systems used NPSOL for optimization, with the same settings of optional parameters and the same scaling of variables. Tests were performed on an IBM RS6000 workstation, and on a Cray Y-MP supercomputer.

The impact of variable order and finite-difference scheme on optimizer performance is similar to that quoted for the simple aircraft optimization discussed earlier. The results presented in this section for the new system are for the best ordering of design variables, and for the standard finite-differencing scheme, because best performance is achieved with these choices.

The baseline system successfully handled this optimization task, reaching the same optimum point as the standard analysis system. This solution was consistently produced for optimization runs from several different initial design points. These results confirm the effectiveness of the new executive routines, and they verify the accuracy of the modified analysis routines.

Optimization with the quasi-procedural system reduced the time spent on the analysis routines. The reduction in analysis subroutine calls for this optimization problem is important, because care had been taken in the original routines to prevent unnecessary computation. For example, design variables that affect the expensive AICS subroutine were identified in an input file, so that it would not be called during the calculation of all gradient components. The quasi-procedural system handled this automatically, but also identified unnecessary calls to inexpensive routines that were repeated thousands of times. The impact of these routines on total computation time is more difficult to handle by handcrafting, but it is also much more significant.

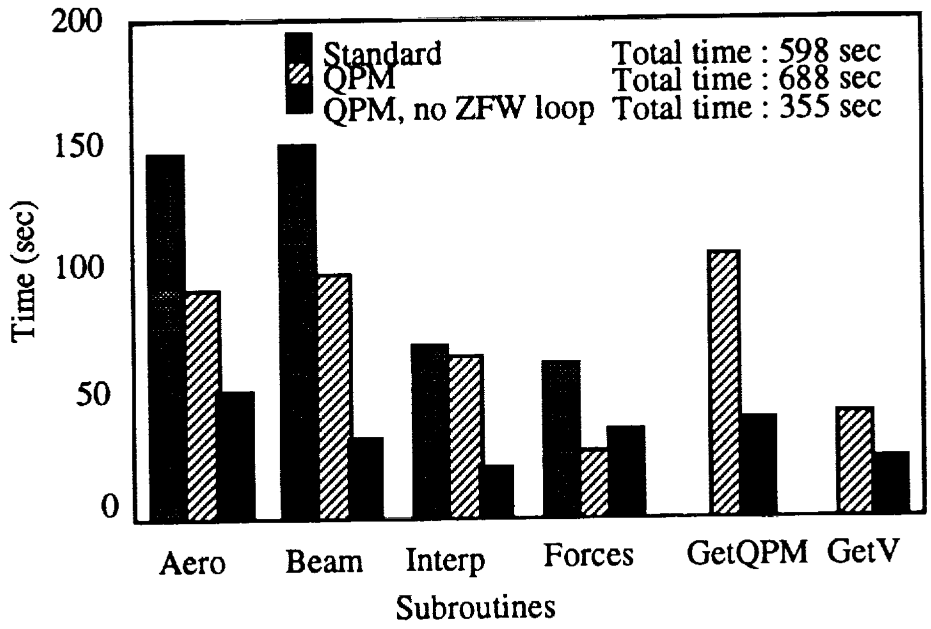


Figure 2.12: Relative computation times for complete optimization.

The overhead associated with the quasi-procedural control of the analyses increased the total time of optimization, so that it required slightly more computation than the standard system. Examination of the most expensive executive routines reveals the aspects of the method that are expensive.

GetQPM is the subroutine that generates and checks the computational path for the system. Almost all of the time spent in this routine is used in the first pass through the analyses, when the path is built. Figure 2.13 shows that GetQPM is less costly in later iterations. As the number of iterations is increased, GetQPM takes a smaller fraction of the total time. The expense of path generation is not really part of the optimization, because it could be built and saved prior to beginning the optimization run. When the same analyses are used for many optimization problems, it is possible to re-use the dependency information, in a manner similar to the re-use of Hessian information when warm-starting an optimizer. Consequently, it is reasonable to account for the bulk of the time spent in GetQPM as development time, because the programmer is relieved of the burden of explicitly coding the subroutine calls.

Much of the overhead time is devoted to writing vectors to and from the database, as local variables are kept separate from global variables. This cost is less significant on the multi-processor Cray Y-MP than on the workstation, but programmers should be aware that it is expensive to transfer vectors to subroutines that are called often. In the example, the AERO routine is called 60,000 times during an optimization, and it uses GETV 28 times at each call, which accounts for 70% of the cost of getting vectors from the database. Alteration of the structure of this single routine, to limit the number of vectors being transferred, leads to a substantial reduction in the cost of optimization.

The remaining overhead is chiefly devoted to identifying which database variables are to be associated with local variables, and to tracking the validity of database variables as inputs are modified. Both of these tasks are strongly affected by the structure of the analyses, because loops require modification of inputs at every iteration. A method for replacing an iteration on zero-fuel-weight with an extra design variable and a constraint was described earlier, and it was noted that the total optimization time was reduced by 48%. Analysis architecture has a dramatic influence on the overhead cost of the quasi-procedural method, with the simpler structure reducing path generation time by 75%. This suggests that further effort should be devoted to developing more general methods for handling non-linear program architectures.

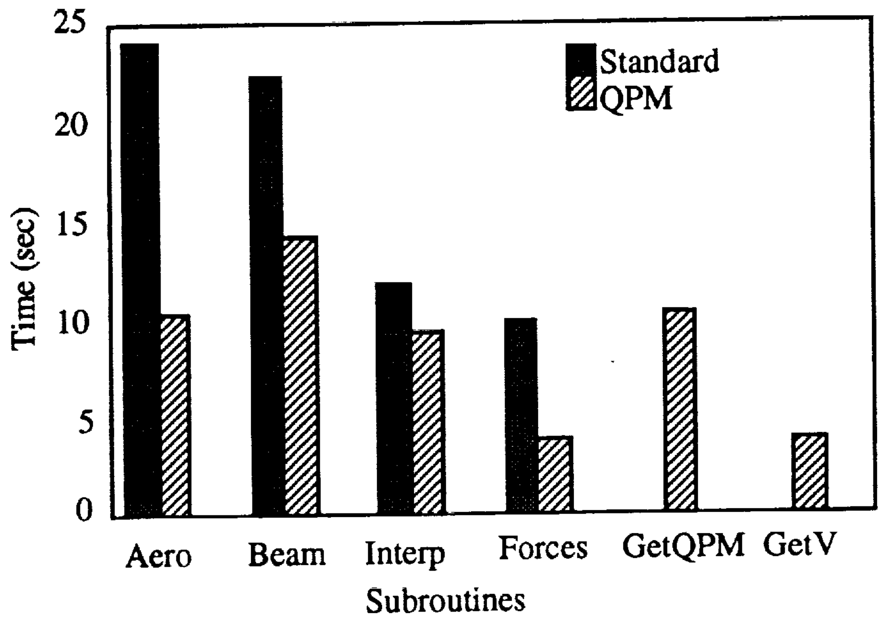


Figure 2.13: Relative computation times for one iteration.

2.5 Summary

This chapter has described the development and validation of a baseline design system that combines a quasi-procedural executive with a sequential quadratic programming optimizer. Standard Fortran subroutines have been modified so that they can be controlled by this system, and a technique for automating the conversion process has been outlined. The system has performed successfully on a computer-intensive optimization task. Its flexibility and extensibility become more apparent as the complexity of the analysis increases. The suitability of this architecture for controlling aerospace optimization systems has been confirmed. It is now being used for preliminary design studies at the Boeing Aircraft Company [37], where an automated software conversion tool is also being developed [38].

Investigation of the role of the quasi-procedural method in calculus-based optimization reveals that the influence of the system is greatest when a small number of inputs are modified at each pass through the computation. Consequently, it is significant when gradients are estimated using finite-differences, but it is less important during the line-search. An investigation of the impact of the method when automatic differentiation [39, 40] is used to compute gradients is in progress, but preliminary results indicate that it provides significant benefits [41].

The best formulation of the optimization task is affected by the interaction between the executive and the optimizer. The quasi-procedural method has been shown to reduce the cost associated with extra design variables and constraints that are introduced to replace iteration loops. Coupling variables and constraints can be used, in similar fashion, to split a large problem into components that execute in parallel. (When an output parameter from one routine is an input to a second routine, the input variable can be made a design variable so the routines can execute independently. The dependence is captured by a constraint requiring the output of the first routine to match the design variable input to the second routine at the completion of optimization). Work on this decomposition method, and on parallel optimization with the quasi-procedural architecture, is being conducted by other members of the aircraft design group at Stanford [42].

Chapter 3

A Simple Genetic Algorithm for Aerospace Design

The range of problems that can be considered in the baseline system is limited by the calculus-based optimizer's need for gradient information, which is not available in many aerospace design tasks. A more general automated search capability can be attained through integration of optimization algorithms which do not use gradients. One such method, a genetic algorithm, is studied in this chapter.

Figure 3.1, taken from Ref. [43], displays several complex and highly-refined flying creatures that have evolved naturally. These biological designs are compared with man-made flying machines with similar features, to illustrate that human designers have often been inspired by biological precedent. Useful analogies are not limited to the designs themselves, but can extend to the process by which they were developed. Genetic algorithms belong to a class of optimization methods known as evolution strategies, which use operators similar to those of natural evolution to guide their search for improved performance. The basic search mechanism of genetic optimizers is described in this chapter.

Genetic optimization occupies a gap in the range of available techniques, lying between gradient-based methods and random search [22]. It can be used in multi-modal domains, or when the search space is discontinuous or noisy.

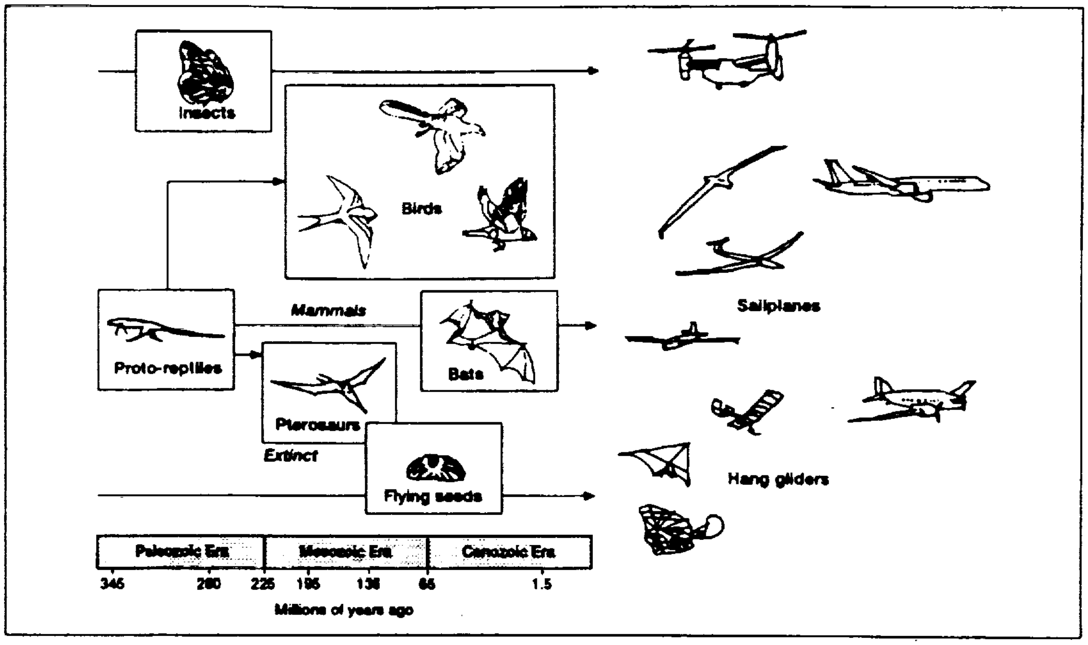


Figure 3.1: The natural evolution of flying devices inspires the development of a genetic algorithm.

Successful applications of genetic algorithms have been reported in many disciplines, including pattern recognition [44], layout [45], scheduling [46], and partitioning [47]. These achievements suggest that the genetic algorithm provides a robust, general purpose search capability, but their use is not universally appropriate. The optimizer must be able to exploit some structure in the search space to guide the generation of improved designs. Expert knowledge can be used to shape a domain so that it possesses this necessary structure. Optimizer parameters can be selected so that the genetic algorithm can detect it.

In this chapter, several features that can limit optimizer performance are discussed. Applications in spacecraft trajectory design demonstrate that implementation details strongly influence optimizer behavior. They also show that the genetic algorithm provides new search efficiency in this design domain.

3.1 The Genetic Search Mechanism

All optimizers search for improvement of an objective function. Calculus-based methods use gradient information as their guide, as described in Section 2.2, but evolution strategies mimic the natural evolutionary process. Genetic algorithms form one class of evolution scheme, and are distinguished by the basic set of genetic operators they use to seek improvement. A number of evolutionary schemes are described here, and the essential features of genetic algorithms are identified.

Natural selection is sometimes simplistically described as "survival of the fittest", but it is more accurate to say "survival of the genetic code of the fittest". Similarly, in a genetic optimizer it is an encoding of the relatively fit candidate that survives. A parametric description of the design is developed, and the optimizer is used to identify appropriate values for the parameters. Each individual is represented by a genetic string, which is a concatenation of the values of the design variables. The entire string is analogous to a chromosome, with genes for the different features (or variables).

The simplest search process is purely random. A single starting point is chosen, and a sequence of undirected mutations, or modifications, is made. The performance of each design is recorded, but information about previously evaluated designs is not used to guide the development of new designs. Convergence to an optimum is only guaranteed if the design space is exhaustively evaluated.

The earliest artificial evolution scheme was developed by Rechenberg in 1964 [48]. It adds selection to the purely random search. When a single design is randomly mutated, or modified, to generate a new design, the two designs are compared. The design with higher performance is selected, and used as the starting point for further modification. This sequence of random modification and selection is repeated for many iterations. Improvement is achieved whenever the mutation produces a design with superior performance, but the new design is rejected when its performance is inferior. Although progress is not guaranteed at every step, it can be accumulated after many iterations.

This scheme out-performs random search when it is better to make random modifications from one point rather than another. This is true when the search domain is regular, because higher performance is expected in the neighborhood of the design with superior fitness. A maximum mutation step should be specified, so that the mutation operation can only reach points in the neighborhood of the current design. The requirement for regularity restricts the range of applications for which this search is appropriate, but it is not unreasonable for an evolutionary algorithm. Dawkins [49] stresses that complex biological designs are the result of accumulating a large number of small changes, with improvement in performance at each change. (It is worth noting, in passing, that simulated annealing algorithms [50] are similar to this mutation-selection search scheme, except in the details of selection, and in the criterion used to size the maximum mutation step.)

The simple mutation-selection evolutionary algorithm performs a series of local searches, which are similar to, but less efficient than, the iterated line searches of gradient-based optimization methods. In multi-modal doamins, these searches can terminate at a locally optimal point. Parallel searches from several points are more likely to identify the global optimum. Consequently, a population of candidate designs which are considered simultaneously has more robust performance than a search from a single point.

When a population of candidate designs is available, the optimizer does not need to conduct a number of independent local searches. Different population members can share information to improve search efficiency. Evolutionary search from a single point must be asexual, and is limited to a mutation operation. Genetic search from a population of points permits sexual reproduction, meaning that offspring can be formed by recombining elements from two parents. This is done in a crossover operation, which takes different pieces of the genetic string from different parents, and recombines them to form viable offspring.

A crossover operator is much more powerful than a random mutation operation when the search domain has a structure that provides correlations between parts of the genetic string (genotype) and the performance of the individual it represents (phenotype). Substrings, or building blocks, which appear in the description of above-average phenotypes are likely to survive into the next generation, even if the genotype is broken up by the action of crossover and mutation. Short, low-order building blocks are retained and combined to form higher-order building blocks, with the process repeating over many generations until the best design is found.

Figure 3.2 illustrates the essential features of a genetic algorithm. A population of candidate designs is generated. Each design is described by a set of variables, which are encoded in a genetic string. The performance of each member is evaluated by computing values for the objective and constraint functions. Designs with high performance are selected to participate in reproduction. The crossover operator recombines elements of the genetic encoding of each parent, to produce a new encoding for a new design that inherits features from each parent. The mutation operator may also modify elements of the new individual, so that new features not present in either parent can be introduced. Thus, the basic operators of selection, crossover and mutation permit both exploitation of the best features in the current population, and exploration for features that are not currently represented.

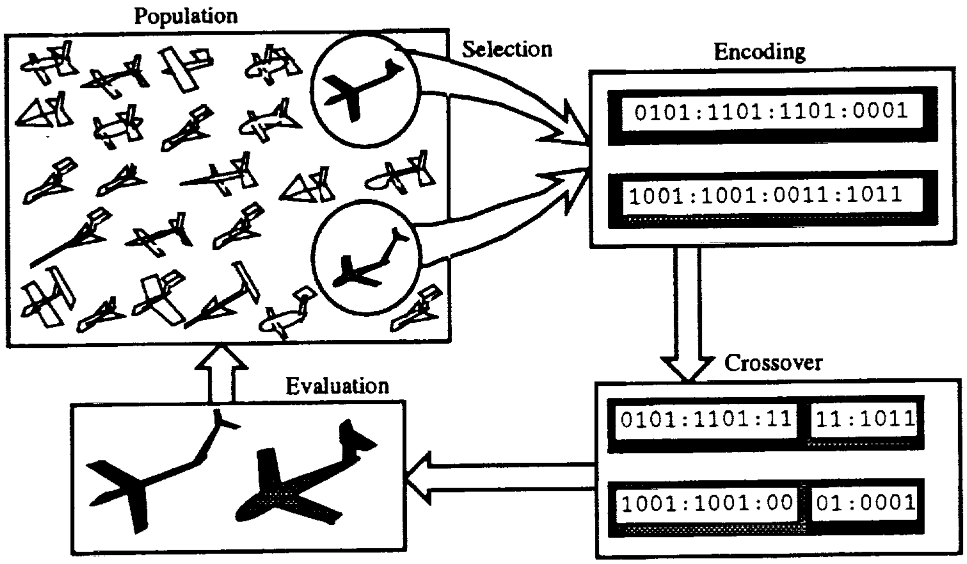


Figure 3.2: A standard genetic algorithm, illustrating the roles of selection, reproduction and crossover.

The crossover operator distinguishes genetic algorithms within the range of evolution schemes. This operator increases search efficiency, but requires a search domain which contains building blocks that can be recognized and exploited through recombination. Although formal gradients are not required, some trend information should be available. These limitations of genetic search are discussed further in the next section.

3.2 Limitations of Genetic Algorithms

The genetic algorithm is unlikely to be successful in domains where low-order building blocks do not combine to form superior higher-order blocks. This may occur when the different variables are decoupled, so that the optimal value of each variable is independent of the values of other variables. Davidor [51] discusses the importance of epistasis (coupling of design variables) on the performance of genetic algorithms. More significantly, above-average low-order building blocks may combine to form below-average higher-order blocks [22]. When this occurs, the domain is called deceptive, because the attractive building blocks are misleading.

Difficulties can arise even in domains that are not deceptive, if the better building blocks are not recognised, or they are not retained. Liepins and Vose [52] note that genetic algorithms may fail if the chosen embedding (representation of the variables in the genetic string) is bad, if sampling error gives unreliable estimates of the relative utility of building blocks, or if crossover breaks up building blocks of high utility. These potential problems are described below.

Encodings that allow description of infeasible solutions increase the work for a genetic algorithm, because they increase the size of the total search space, and reduce the proportion of useful building blocks in the population. Use of a precedence matrix in a sequencing task allows the description of candidate orderings that are logically inconsistent (eg A precedes B, B precedes C, C precedes A) [53]. With only 8 items to be placed in sequence, 99.98% of all possible strings describe impossible orderings, and a randomly-generated initial population is unlikely to contain any feasible candidates. Permutation encodings used in conjunction with re-ordering operators are much more successful in problems of this type [42].

The population is a small sample of all possible designs in the domain. An ideal sample includes all the important features of the domain, but if some helpful building blocks are not present the representation is not accurate, and the sample has some error. Population size has a strong effect on sampling error, with the error reduced in large populations where building blocks are more likely to be represented. Goldberg has presented theoretical results for optimal population size for serial genetic algorithms which suggest that total computation can vary anywhere from logarithmically to exponentially with problem size, depending on the choice of population size [54]. As the size of the problem increases, the population should also increase, but the required size will be domain-dependent.

Crossover is likely to disrupt useful building blocks when the components of those blocks lie far apart in the string, because the probability of the crossover point falling between the components is high. Expert knowledge of the domain can help to prevent crossover disruption. In aircraft design, a good encoding would place wing sweep and thickness-to-chord ratio close together in the genetic string. They are very tightly coupled, so they will form a useful building block which is unlikely to be broken up by crossover if they are consecutive entries in the string.

In function optimization by genetic methods, the standard method for handling constraints is the application of penalty functions to the objective [22]. It is desirable to use graded penalties that reflect the extent of constraint violation, rather than applying a harsh penalty in an attempt to avoid the infeasible region entirely [55, 56], but it is difficult to grade these penalties when evaluation of the performance is impossible. If a fixed penalty is applied, the genetic optimizer can converge prematurely, to a point that is simply feasible rather than optimal. When many members of the population are unable to be analyzed, and consequently share a very poor rating, it is difficult to correlate the building blocks with performance, and sampling error is increased. An alternative approach for handling constraint violations is to perform some sort of repair to correct the infeasibility [57], and to evaluate the performance of the repaired design.

3.3 A Simple Genetic Algorithm

The genetic algorithm used in this research is based on Goldberg's SGA (Simple Genetic Algorithm), which is listed in Ref. [22]. The operators are slightly modified to handle real-valued or binary encodings. Roulette-wheel selection is replaced by tournament selection, and advanced operators for species formation and mating restriction are introduced. The software developed for this thesis is described in Appendix B. The general features of the algorithm are described here.

An example of a string to be used in genetic optimization of a wing is presented in Fig. 3.3, along with the candidate design that it represents. Each feature (dihedral angle) is given a binary representation in the example, but integer or real representations are also possible [58]. The form of the encoding distinguishes genetic algorithms from genetic programming, where the string contains the parse tree of a program [59].

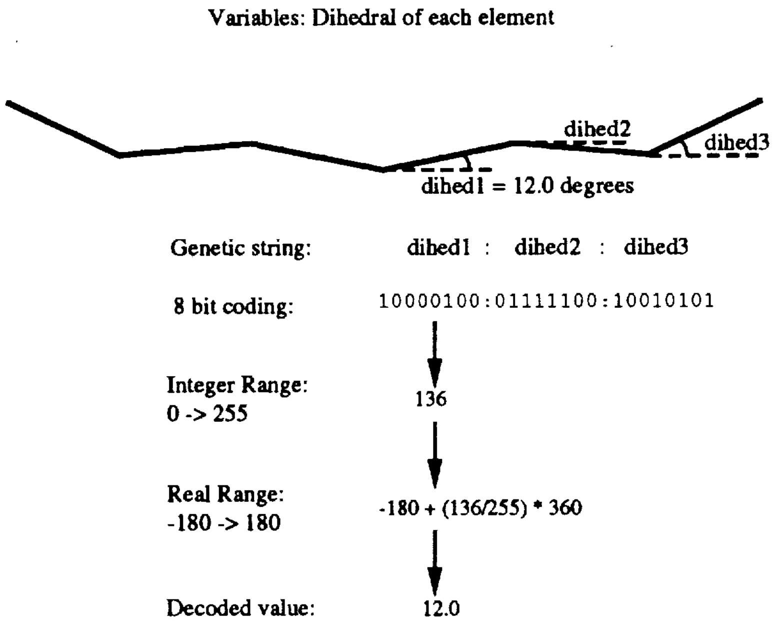


Figure 3.3: Decoding a genetic string.

Selection is the essential element of an evolutionary scheme. Some performance metric must be chosen to rate the relative fitness of different population members. This fitness measure is used to select high performance individuals to participate in reproduction. The idea of fitness is natural in optimization: any algorithm compares the performance of different candidates, and chooses higher performance. The standard statement of an optimization task requires minimization of an objective function, whereas the genetic algorithm favors maximum fitness. Hence, an appropriate fitness function will be inversely related to the regular objective.

$$f_i = \frac{1}{1 + J_i}$$

where f denotes fitness and J represents the objective value. For J $\geq$ 0, this guarantees fitness values in the range [0,1].

Selection of individuals to participate in reproduction is performed using a tournament scheme. Each time a parent is needed, k members of the current population are selected at random, where k is the tournament size. Their fitness is compared, and the highest fitness individual becomes the parent. With this scheme, it is expected that the best individual will be a parent k times per generation (it will participate in k tournaments and win them all), with linear decline in expectation of reproduction to the worst individual, which cannot win a tournament. A tournament size of two ($k = 2$) is used in this research. This method is introduced to replace roulette-wheel selection, which is susceptible to premature convergence when a poorly scaled fitness domain allows one individual to dominate reproduction. Ranking schemes prevent such dominance, because they are not affected by the margin of superiority of higher-fitness individuals. Tournament schemes perform a local ranking at each selection operation, without ever requiring the entire population to be sorted.

Raw selection only produces clones of the current best in the population. Improvement requires modification of their genetic code. The genetic algorithm needs reproduction operators, such as crossover and mutation, that alter the selected encodings to create offspring of higher fitness.

In crossover, two individuals swap part of the string, so two new individuals are formed as combinations of parts of the old strings. An example of the effect of crossover is shown in Fig. 3.4. A crossover point is randomly chosen along the string, and is the same for both parents. When one parent has above-average building blocks in one part of the string, and the other has good building blocks in another part of the string, the offspring receives useful building blocks in both sections, and should have fitness superior to either parent. The operation shown is a single-point crossover, meaning that each parent string is broken at only one point. It is used in preference to multi-point crossover, which has been found to be effective in some applications [60], but can be excessively disruptive when building blocks are large.

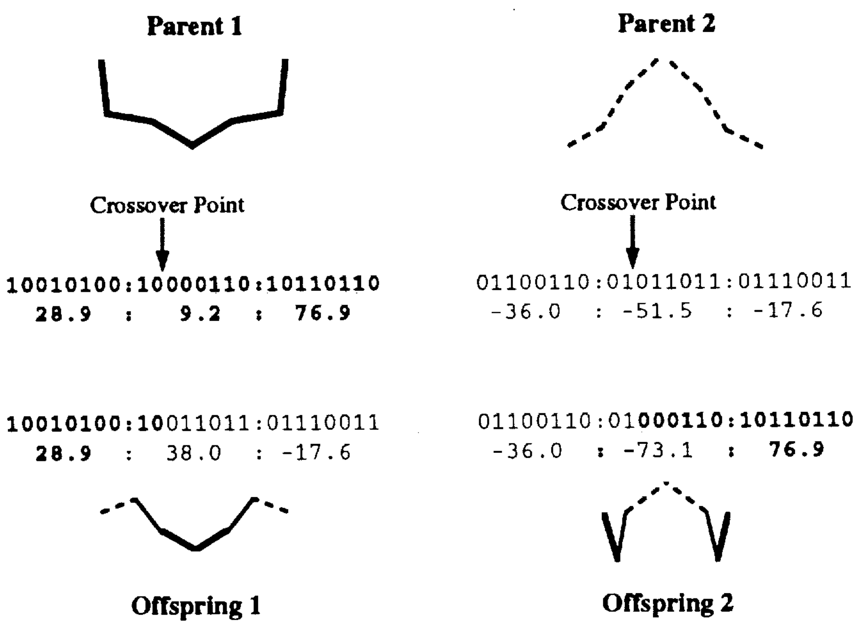


Figure 3.4: Crossover.

If a real-valued encoding is used, crossover occurs at a variable rather than between two binary bits. The variables in the genetic string are copied from one parent before the crossover point, and the other parent after the crossover point. The value of the crossover variable is mutated:

$$Val_{Off} = Val_{P1} + Rand \times (Val_{P1} - Val_{P2})$$

Here, *Off* denotes offspring, while *P1* and *P2* refer to Parents 1 and 2 respectively. *Rand* is a random number in the range [-1,1]. This mimics the effect of the operator for the binary string, where one variable generally changes value (because the crossover point lies withing the substring representing that variable).

Crossover can disrupt existing building blocks in the quest for new combinations of features. The crossover probability should be chosen to balance the number of new designs introduced each generation against the possibility of losing high-fitness individuals from previous generations. Although Grefenstette recommends crossover probabilities of 0.6 - 0.8 [61], the accuracy of the tournament selection scheme reduces the likelihood of losing high-performance designs, and a crossover rate of 0.9 is used in the applications described later.

A mutation operation allows modification of elements of the new individual, so that new features that were not present in either parent can be introduced. Pointwise mutation is a very simple operator that allows individual bits of the string to be changed. This produces corresponding changes to a design variable, as shown in Fig. 3.5. Mutation rate is generally quite low, so that random new features are introduced in only a few members of the population. If a mutation is advantageous, it can be exploited in subsequent generations through the action of selection and crossover.

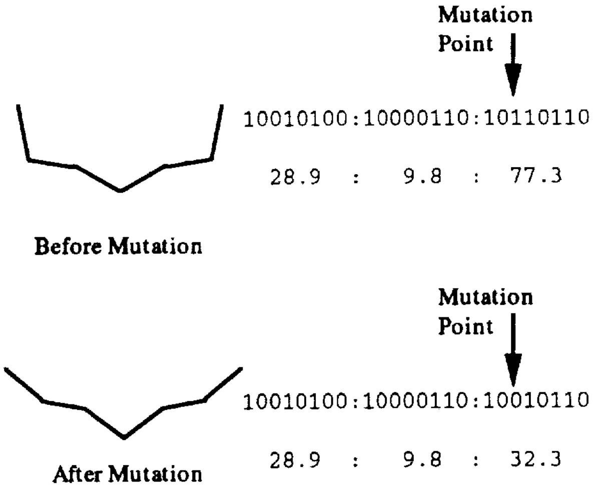


Figure 3.5: Mutation.

A basic genetic algorithm is unable to maintain sub-populations at several local minima without the introduction of some mechanism to induce diversity. A sharing function, added to a standard genetic algorithm, can help to identify several local minima by allowing the formation of several 'species'. Information about several near-optimal alternatives is useful at the preliminary design stage. A final choice between them can be made after more detailed analysis, or by considering factors not modelled in the optimization task.

Goldberg and Richardson have reviewed several proposed schemes [62], and conclude that a sharing function is effective. This sharing function has been used successfully by KrishnaKumar et al to identify multiple near-optimal solutions for a structural control problem [23]. The function simply penalises population members that are close together in the search space, by degrading fitness according to the distance between them. The distance between ith and jth population members is

$$d_{ij} = \sqrt{\sum_{k=1}^{n} \left(\frac{x_{k,i} - x_{k,j}}{x_{k,max} - x_{k,min}} \right)^2}$$

where x is the vector of design variables, and k denotes each element of the vector (from 1 to n, where n is the number of design variables). The sharing function is

$$S(d_{ij}) = \begin{cases} K \times (1 - \frac{d}{\sigma}) & \text{if } d \leq \sigma \\ 0 & \text{otherwise} \end{cases}$$

The sharing strength, K, is a parameter provided by the user. The maximum sharing distance, σ, depends on the dimension of the search space and the number of assumed minima, q, which is another user-selected parameter.

$$\sigma = \sqrt[n]{\frac{1}{q}}$$

The adjusted fitness is then given by

$$F_i = \frac{f_i}{\sum_{j=1}^{N} S(d_{ij})}$$

Note that the denominator is always ≥ 1, because $S(d_{ii}) = 1$, so fitness can only be degraded by sharing.

When several sub-populations are clustering around different local optima, crossover of two parents from different clusters is unlikely to produce offspring of higher performance, because they are likely to be far from either local optimum.

Deb and Goldberg have investigated several schemes for restricting mating between sub-populations, or species [63]. They report that it can be helpful to allow mating only between parents with separation $\leq \sigma$.

Genetic algorithms do not have convergence criteria analogous to the Karush-Kuhn-Tucker conditions exploited by gradient-based methods, so the number of function evaluations required to find a solution depends on the user-selected termination criteria. For the results in this chapter, a specified number of generations is used as the only termination criterion. For each optimization task, the algorithm is run several times to check that results are consistent, because performance can vary due to the probabilistic nature of the genetic operators.

3.4 Application to Spacecraft Trajectory Design

Although genetic algorithms can be applied to a wide variety of problems, satisfactory performance often requires very careful implementation. Selection of constraint handling technique can be critical, because the choice exerts a strong influence on the topology of the design space. The importance of these issues is demonstrated in the applications that are described in the remainder of this chapter.

3.4.1 Background

The investigation of interplanetary trajectories is a typical design problem. A large number of potential solutions are initially considered, with relatively simple analysis techniques. A few of the most promising alternatives are then chosen as starting points for more detailed design. Effective preliminary studies must identify the best alternatives rapidly and accurately.

Initial analysis of potential interplanetary trajectories is often performed using patched-conic techniques, in which the spacecraft is assumed to be solely influenced by the gravitational attraction of the Sun during its heliocentric transfer. Interplanetary transfer legs are specified by (1) the departure date and body, and (2) the arrival date and body. Complete trajectories are described by a sequence of interplanetary legs. In the preliminary design phase, a variety of mission types may be investigated (composed of different interplanetary legs) and a range of departure and arrival dates must be assessed for each leg.

The interplanetary trajectory design space is typically characterized by numerous minima separated by infeasible regions. Consequently optimization by standard methods is difficult. Some patched-conic analysis programs include a gradient-based optimization method [64, 65, 66], but only the local minimum closest to the starting point is discovered by these methods. Programs that do include a gradient-based optimizer typically have a grid search option and suggest its use during initial mission studies. In the grid search, the program steps through the departure and arrival date ranges for each leg of the mission, and every potential trajectory is simulated [64, 67]. The grid must be fine enough to detect local minima. Hence, for missions with many planetary encounters or large ranges of dates, the number of function evaluations grows very quickly.

For this application, a standard genetic algorithm has been added to the IPREP (Interplanetary PREProcessor) program, which is part of the IPOST (Interplanetary Program to Optimize Simulated Trajectories) package [64]. This allows the genetic algorithm performance to be compared directly with the grid search method already available in IPREP. The sharing function and mating restriction scheme described in the previous section are available as optional extensions to the basic algorithm, so that their effect on optimizer performance can also be evaluated.

The IPREP grid search option allows the user to specify a range of starting dates for a mission, and a range of duration times for each interplanetary leg. The genetic algorithm uses those dates and times as design variables, with the same minimum and maximum values. The genetic string is simply a concatenation of these floating point variables. Although schemata theory suggests that binary strings are more efficient, floating point codings often work well in practice [68]. They are preferred here because the encoding is more natural than a binary representation.

3.4.2 One-Way Direct Earth to Mars

The first problem used to verify the effectiveness of the genetic algorithm is a one-way direct mission from Earth to Mars (Fig. 3.6). A 6000 kilogram payload must be delivered to a 1 Sol parking orbit (period of 24.6 hours) about Mars. The objective is to minimize the initial mass in low-Earth orbit.

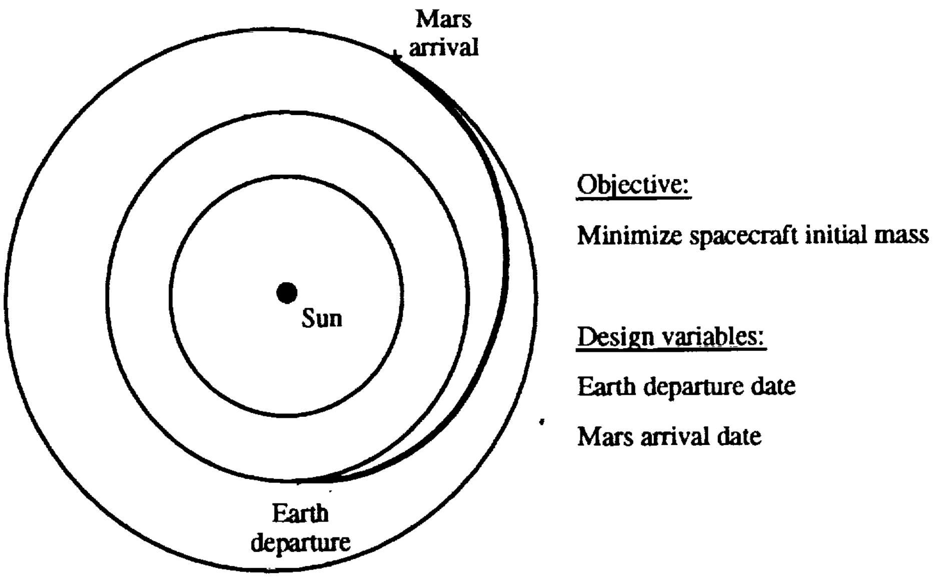
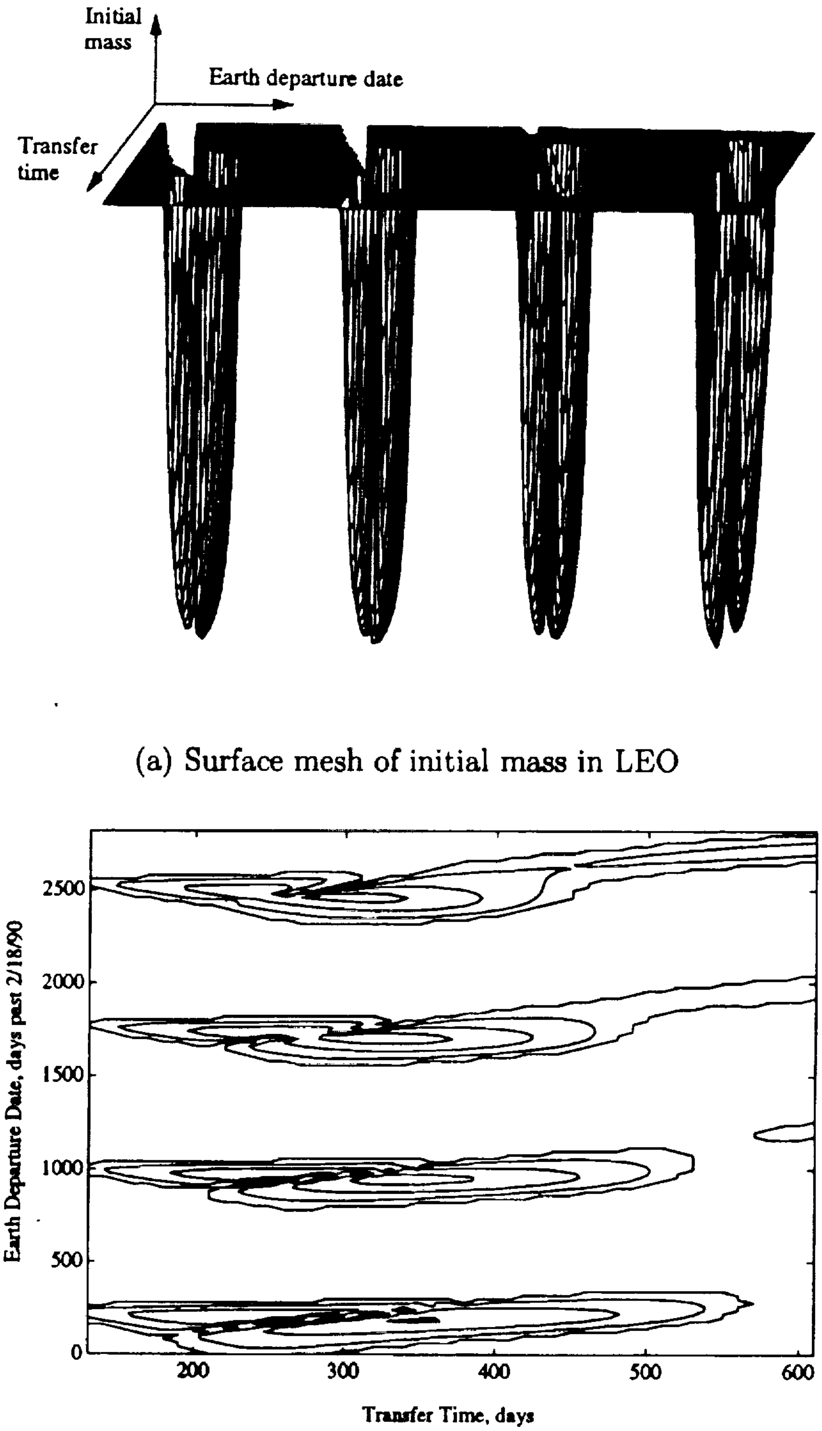


Figure 3.6: Earth to Mars mission.

Earth departure date and Mars arrival date are the only two variables, so the complete design space can be represented graphically (Fig. 3.7). There are eight local optima (located at the bottom of the valleys). Four of them have objective values within 5% of the global optimum (17700 lb). Infeasible points are assigned a nominal large initial mass (100000 lb), so infeasible regions appear as a plateau in the mesh plot.



(a) Surface mesh of initial mass in LEO

(b) Contours of initial mass in LEO. Levels are 20000, 30000, 60000, 100000 lb.

Figure 3.7: Design space for Earth to Mars mission.

Search Method	Function Evaluations
Genetic Algorithm	3000
Coarse grid followed by local fine grid (1-day step)	13564
Fine grid (1-day step)	1346400

Table 3.1: Function evaluations for different search methods.

The genetic algorithm readily solves this problem to within a day of the exact optimum, in fewer than 3000 function evaluations (30 generations, each with population size of 100). Further refinement of the solution is inappropriate for these mission feasibility studies, as it exceeds the accuracy of patched-conic theory. The efficiency of the genetic algorithm compares favorably with grid search, as shown in 3.1. The step size for the grid search has to be chosen carefully, because too large a step may jump over the true optimum. Usually a coarse grid is used to locate promising regions, and subsequently a fine grid is used in those regions. For this problem the initial grid has a 5-day step size for launch date and a 20-day step size for duration of the interplanetary leg. The local grid has a step size of 1 day, to match the accuracy attained by the genetic algorithm.

For the genetic algorithm, the distribution of the population in the design space evolves over the generations, as indicated in Figs. 3.8, 3.9, 3.10, 3.11. The location of each population member is marked by a cross on the contour plot of the design space. The members of the first generation are randomly chosen by the optimizer. The original population is largely infeasible, but there are members in each of the feasible regions. Most of the population in Generation 10 is feasible. By Generation 20 the population is clustering around the two best local optima. At Generation 30 the entire population is clustered around the global optimum.

When most of the design space is infeasible, and all infeasible points are assigned the same fixed value of the objective function, it is difficult to correlate parts of the genetic string with performance. Most strings produce identical

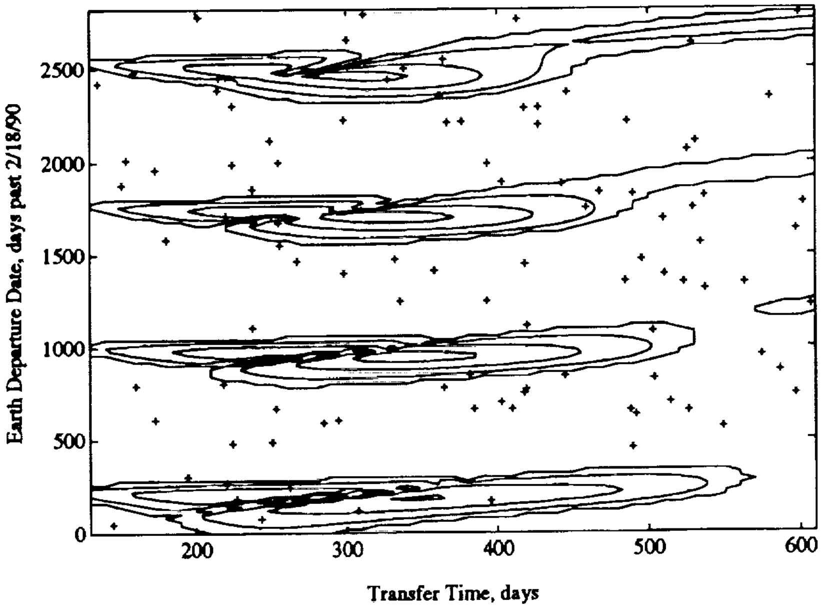


Figure 3.8: Population distribution, Generation 1

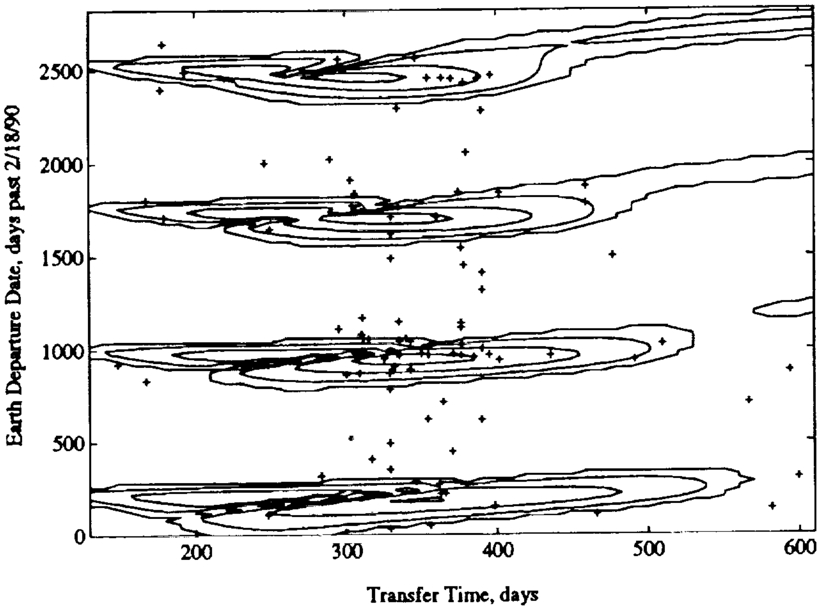


Figure 3.9: Population distribution, Generation 10

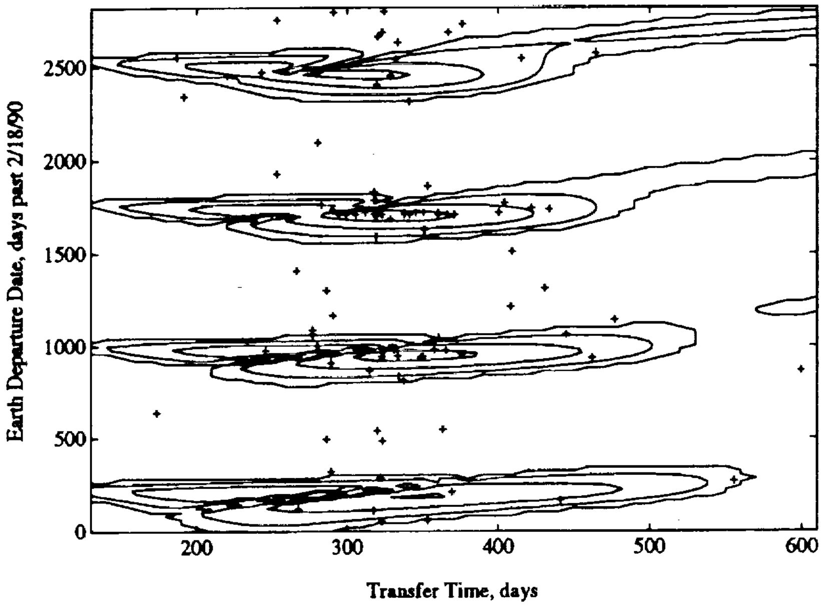


Figure 3.10: Population distribution, Generation 20

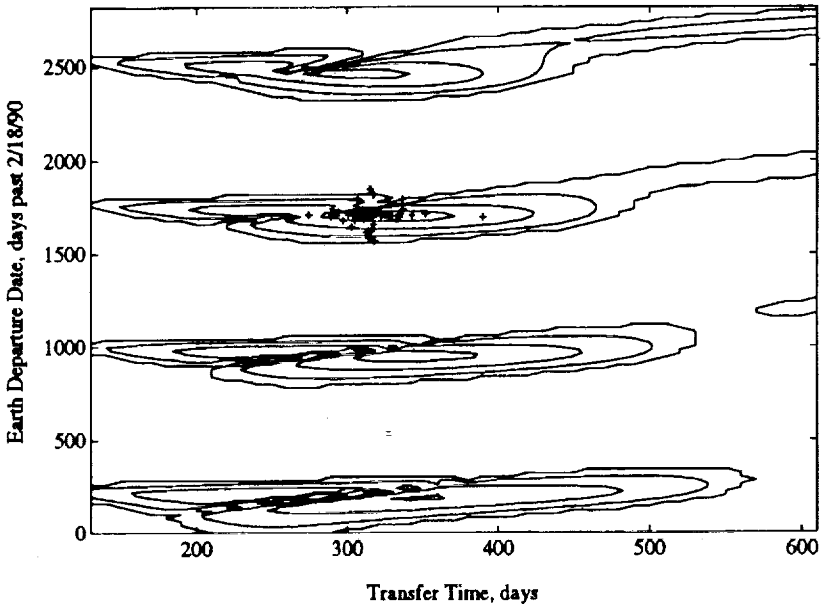


Figure 3.11: Population distribution, Generation 30.

objective values, so selection pressure is very low. Search is worse than random until feasible points are found, because infeasible points are chosen for reproduction instead of new random points being tried.

The effect of large infeasible regions can be demonstrated in this domain by modifying the objective function, so that any trajectory with initial mass larger than 20000 lb is considered infeasible. This shifts the infeasible plateau, shown in the surface mesh plots, from 100000 lb (Fig. 3.7) to 20000 lb (Fig. 3.12). The new design space is 95% infeasible.

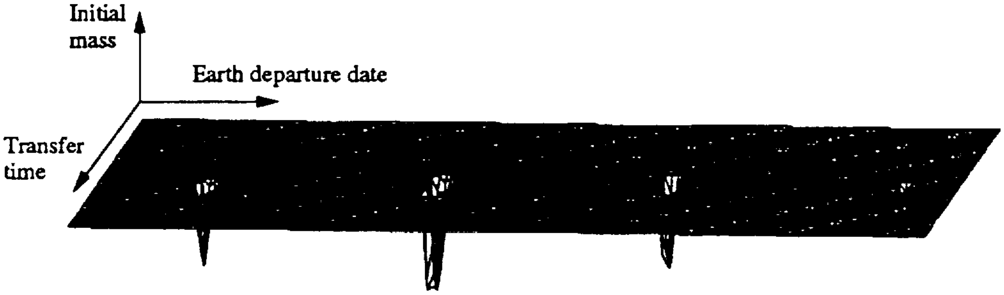


Figure 3.12: Surface mesh of initial mass in LEO, 95% infeasible.

Figure 3.13 indicates that this new space is indeed more difficult for the genetic algorithm. After 30 generations, by which time the previous example had fully converged, most of the population is still infeasible. A few members have located a local minimum. Members from this small feasible region quickly dominate the reproductive process, and by Generation 50 (Fig. 3.14) the entire population has shifted to this minimum. The other feasible regions were never located, and a global minimum in one of those regions would be missed.

This example confirms that large flat areas in the design space should be avoided. The infeasible region should be graded so that points that are nearly feasible are more likely to reproduce than points that are far from feasible. This is not always possible, but one example is provided later in this chapter, where a quadratic penalty is applied to violations of the constraint on total trip duration. The extent of infeasibility is of no interest in grid search, because information

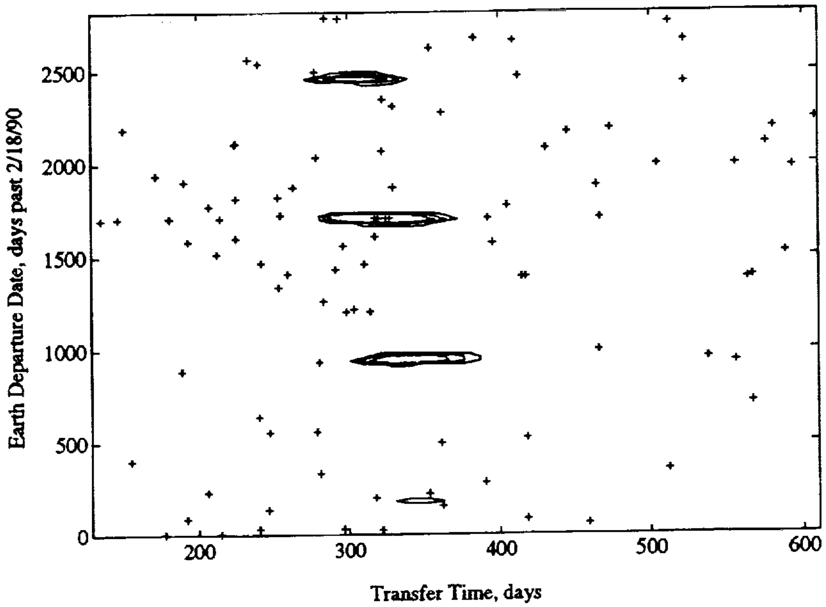


Figure 3.13: Population distribution, Generation 30, 95% infeasible.

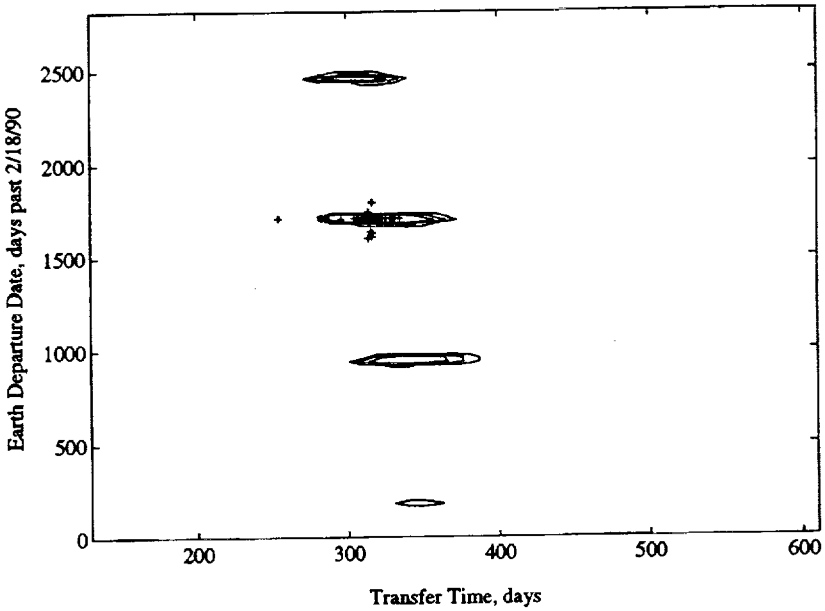


Figure 3.14: Population distribution, Generation 50, 95% infeasible.

from current points does not guide the selection of future points. This distinction should be noted when developing a problem description, and the solution method should influence the design of the search space topology. Genetic algorithms can handle large regions of constant objective value, but when they occupy more than 90% of the design space, grid search may be competitive.

The history of population distribution presented in Fig. 3.8 to Fig. 3.11 showed that the population in a basic genetic algorithm ultimately clusters around a single optimum point. It would be preferable to preserve information about several local minima, so that the misson planner can select one of them for reasons not included in the statement of the optimization problem. A sharing function that can preserve subpopulations at several local minima was described earlier in this chapter. This function is added to the basic genetic algorithm, and its influence on optimizer behavior is now investigated. The number of expected minima, q, is set to be 5. This is not the actual number of minima present in this space. The resulting performance is representative of behavior in a design space of unknown topology.

When a sharing function is applied without a mating restriction, any population member can mate with any other. The effect on convergence is shown in Fig. 3.15. At Generation 30, when the standard algorithm had fully converged, there are population members retained at each local minimum, but there are still many infeasible designs.

Offspring can take some features from a parent in one feasible region, and different features from a parent in another feasible region, and end up with a combination that is infeasible. The poor performance of these offspring makes them unlikely to reproduce, but they are replaced each generation by new offspring produced in a similar manner. Without a mating restriction, there will always be a significant number of infeasible designs in the population.

The introduction of a mating restriction successfully resolves this difficulty with the sharing function. A restriction suggested by Deb and Goldberg [63] limits crossover between separate feasible regions. Figure 3.16 indicates that the population is well distributed in each feasible region. The value for expected

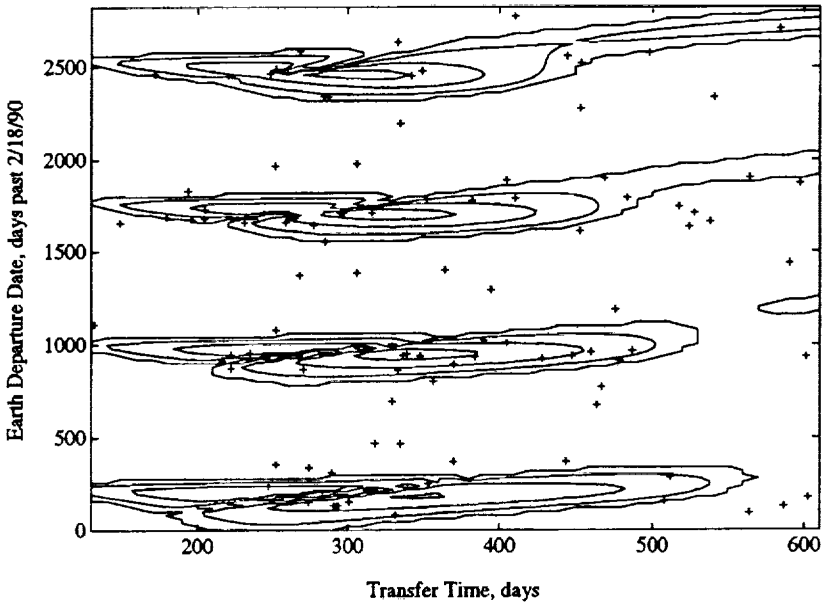


Figure 3.15: Sharing without mating restriction. Generation 30. K = 1.0

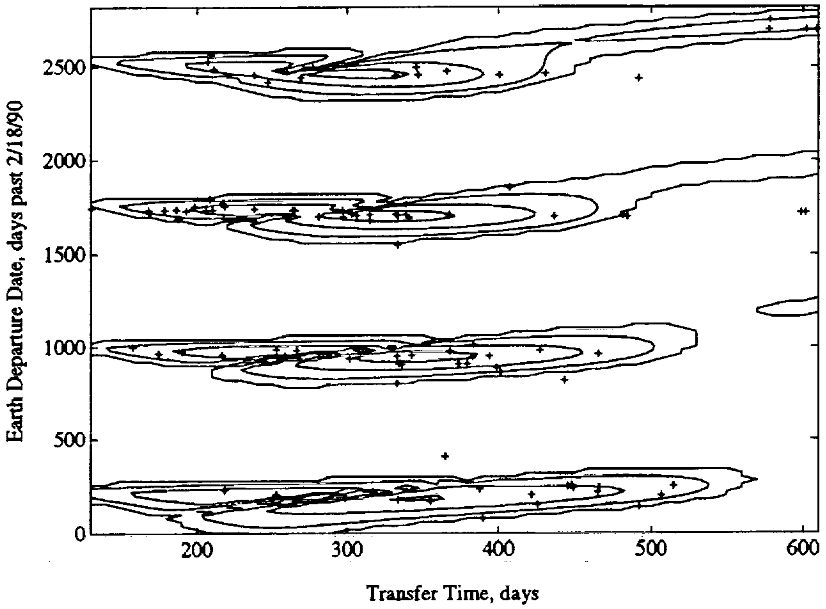


Figure 3.16: Sharing with mating restriction. Generation 30. K = 1.0

number of minima was not critical to the success of the sharing function, because the algorithm is not forced to find exactly that number of minima.

The influence of the sharing strength parameter, K, can be observed by comparing Fig. 3.16 with Fig. 3.17. Lower sharing strength allows tighter clusters to form. Thus, Fig. 3.17 shows sub-populations around the four minima that are within 5% of the global minimum, and a few members near three other local minima. In general, it is useful to reduce the sharing strength as population size increases. Choosing $K = 0$ eliminates the effect of the sharing function.

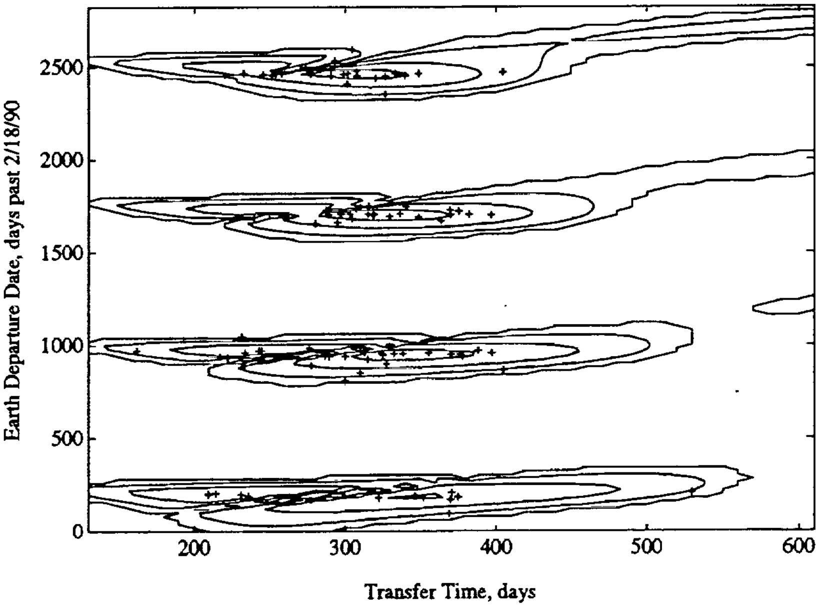


Figure 3.17: Sharing with mating restriction. Generation 30. K = 0.025

The introduction of a sharing function and a mating restriction allows the genetic algorithm to retain more information about the design space. This is achieved without significant increase in the function evaluations required to reach the optimum. Consequently these features are retained in the genetic algorithm, and are used in the more difficult optimization tasks that are described next.

3.4.3 Roundtrip Earth to Mars, with Optional Venus Swingby

The performance and behavior of genetic algorithms has been established for a two-variable problem. Now several potential opposition-class roundtrip manned missions to Mars are investigated. They include direct and Venus-swingby (inbound, outbound and double) interplanetary transfers. The number of design variables varies from 3 to 5, depending on the number of Venus-swingby legs in the mission. These tasks allow comparison of genetic algorithm and grid search performance for a range of problem sizes.

Total trip time is limited to 2 years with a 60-day stay at Mars. Earth departure in the 2010-2025 time span is considered because this design space has been previously investigated using grid search techniques [69, 70, 71, 72]. For this study, missions employing nuclear thermal propulsion (Isp = 925 sec) are evaluated. All missions begin and return to a 500 km circular, 28.5 degree inclination, Earth orbit. Upon Mars arrival, the interplanetary spacecraft inserts into a 1 Sol parking orbit with a periapsis of 500 km. Earth-return is accomplished with minimal propulsion using a reentry capsule. All vehicle mass estimates used in this analysis are chosen to be consistent with References [73, 74, 75, 76].

The summary of results presented in Table 3.2 clearly indicates that the genetic algorithm is more efficient than grid search for all problem sizes considered. This superiority is more marked for the larger problems. The population size must be sufficiently large to accurately sample the search space. As the total search space increases, population size is also increased.

The difference in performance for the outbound swingby and inbound swingby cases, both 4-variable problems, is due to the different topology of the search spaces. The importance of representation of infeasible regions, which was discussed earlier in this chapter, is demonstrated by the relative performance of two strategies used in the solution of the double swingby case. This representation issue is particularly important for the double swingby case because it has the largest infeasible space. The total duration of the return mission to Mars is limited to two years. Satisfaction of the constraint on total duration of the

Mission Type	Design Variables	GA Function Evals (N × Gens)	Grid Function Evals
Direct	3	4000 200 × 20	8000
Inbound swingby	4	12000 300 × 40	64000
Outbound swingby	4	18000 300 × 60	64000
Double swingby, fixed penalty	5	120000 1500 × 80	512000
Double swingby, quadratic penalty	5	50000 500 × 100	512000

Table 3.2: Function evaluations for different problem sizes.

return mission to Mars becomes more difficult as the total number of legs is increased. For the double swingby mission, with the chosen ranges of allowable duration for each leg, only 25% of the missions that can be described have a total duration of less than two years. Many of these missions are infeasible for other reasons, so just 2% of the entire space is feasible.

The standard method for representing infeasible points in IPREP is to assign a fixed performance value. This produces an infeasible region with a flat topology, that gives no information about the proximity of infeasible points to feasible regions. A simple alteration to this representation allows the true performance of the mission to be calculated, and the constraint violation is handled by adding a quadratic penalty that grows as the extent of infeasibilty increases.

$$J = J_{uncon} + PenWt \times (t - t_{max})^2$$

Here, J is the objective value, J_{uncon} is the unconstrained objective value, $PenWt$ is a user-specified weighting factor, t is the trip duration, and t_{max} is the maximum allowable trip time. The use of this method for penalizing excessive trip duration does not reduce the number of infeasible points, but it reflects the extent of infeasibility. The smoother search space can be accurately modelled with a much smaller population, and total work for the genetic algorithm is reduced by almost 60%.

3.4.4 Simultaneous Investigation of Different Mission Types

The various optimization tasks considered in the previous sections are really all sub-tasks in a larger optimization problem. The ultimate goal is to find the best way to get to Mars and return safely. According to standard practice, the best example of each mission type is found independently, and then these examples are compared to find the best mission. The genetic algorithm offers the opportunity to solve the entire problem in a single optimization run.

A description of the full problem must provide enough degrees of freedom to characterize a mission that includes all possible legs. Each leg is identified by the planetary encounter at its end point. Thus the general candidate for the Mars return problem allows legs for launch, outbound Venus swingby, Mars arrival, inbound Venus swingby, and Earth arrival. (Mars departure is not a variable, because it is fixed at 60 days after arrival.) Extra variables are added to this set, to indicate whether optional legs should be included in a particular candidate. These variables switch between 'Include' and 'Not include' values. When these variables have the 'Not include' value, the variable describing the duration of that leg is ignored. For the Mars mission, only the Venus swingby legs are optional. Hence, there are two switching variables, taking the total number of variables to 7.

It has been suggested that these extra variables might be avoided by simply adding a zero value to the set of allowable duration values for an interplanetary leg. However, the set of allowable values should be continuous between the lower and upper bounds. If a leg exists, the lower bound is always much larger than zero, because an interplanetary leg will never have a duration of only a

Start of Launch Window	Optimal Mission Type	GA Function Evaluations	Grid Search Function Evaluations
April 26 2015	Outbound Swingby	140000 2000 × 70	648000
Sept 28 2019	Double Swingby	160000 2000 × 80	648000

Table 3.3: Locating global optima of different type.

few days. Moving the lower bound to zero would cause description of many more infeasible designs. This has been shown to cause difficulty for the genetic algorithm. Hence, it is better to retain the lower bound, and introduce the separate variable.

The switching variables used here are not binary, because missions including and excluding optional legs should not be equally represented. Candidate mission types with more legs occupy a larger fraction of the total search space than missions with fewer legs. The randomly generated initial population should provide an accurate sample of the entire space, so most of its members should include optional legs. In these studies, the switching variables are weighted to give a 95% chance that each optional leg will be included in members of the initial population.

The duration of a good Mars arrival leg that originated on Earth is generally not appropriate for a leg that originated at a Venus swingby. Crossover between missions with different numbers of Venus swingby legs is therefore unlikely to produce offspring of improved performance. Consequently, the inclusion of the sharing function and mating restriction is very helpful in this domain.

Table 3.3 shows the results of running the complete optimization problem for two different launch opportunities, each of 100 days. These opportunities were chosen because the optimum missions were known to be different types [72, 73]. Identification of different mission types indicates that the success is not due to the method favoring a single type. In each case, the genetic algorithm located the global optimum more efficiently than a sequence of separate grid searches, with less intervention required of the user. The benefits in both efficiency and convenience are expected to be even more marked for larger problems.

3.5 Summary

Successful applications of a genetic algorithm to several optimization tasks, ranging in size from 2-variable to 7-variable problems, have been reported in this chapter. Genetic optimization is both more efficient and more convenient than the grid search technique commonly used for interplanetary trajectory design, particularly with the introduction of sharing and mating functions that allow simultaneous identification of several distinct local optima. These results confirm that genetic algorithms can play an important role in preliminary design, but they should be used judiciously.

The performance of the genetic algorithm is influenced by the topology of the search space. Consequently, the user should ensure that the objective function distinguishes the fitness of different designs wherever possible. The introduction of graded penalties that reflect the extent of infeasibility, rather than a fixed fitness value for any constraint violation, helps significantly in this regard. Modification of the size of the search space, through disqualification of strings that cannot be analyzed or through repair of infeasible candidates, can also increase efficiency of the genetic algorithm.

Special switching variables, which refer to the existence or absence of possible components of a candidate design, can be included in the genetic encoding. This allows simultaneous assessment of designs of varying complexity, because different population members can have different numbers of components. In the interplanetary trajectory application, automatic selection of mission type produces a significant reduction in the interface work required during optimization. The ability to explore a space that includes designs of varying complexity is explored further in the next chapter.

Chapter 4

A Variable-Complexity Genetic Algorithm

The search mechanism used by genetic algorithms relies on the identification of high-performance building blocks in candidate designs, which are then recombined so that promising features from different designs are collected in new candidates. Despite the suggestion of growth, the morphology of candidate designs remains static throughout the conventional optimization process. However, the building block description can be extended to allow the nature of candidates to change during optimization. In this chapter, a new genetic algorithm that provides this capability is introduced. Application to a simple block-stacking task demonstrates the fundamental advantage of the new algorithm.

4.1 Motivation for Variable-Complexity Optimization

The scope of the search capability provided by standard optimization algorithms is limited by the need for a fixed parameterization of the problem - one that must be specified *a priori*. The concept must be fully developed by the designer, and the algorithm simply finds the best values for the chosen variables. When several alternative configurations are being considered, each candidate must be described and optimized independently. Furthermore, re-optimization is required whenever the designer changes the description of a particular concept.

A more flexible algorithm would automatically alter the parameterization during optimization. Successful features can be identified in simple designs, and subsequently refined as the algorithm increases the complexity of the parameterization. This progression is familiar to aeronautical designers, who generally select approximate values for global features such as wing area and span, before concentrating on more specific aspects such as the airfoil sections at different stations along the wing. The ability to modify the parameterization can extend the role of optimization beyond automated analysis of prescribed configurations to automated design of new concepts.

Natural evolution does not limit improvement by operating on the values of a fixed number of parameters. Dawkins [49] defines a complex object as one which 'could not have come into existence in a single act of chance'. He acknowledges that it might be developed by an intentional designer (as aircraft are), but stresses that complexity can also result from cumulative selection (as it does in biological systems). Figure 4.1 suggests that the analogy with natural evolution should be extended to include the evolution of complexity.

The standard genetic algorithm possesses a limited capability for assessment of different concepts, because the encoding can include variables that describe the existence or absence of optional features. One such scheme, used to simultaneously examine different interplanetary trajectory concepts, was described in the previous chapter. Several researchers have used similar schemes for genetic solution of structural optimization tasks [78, 79, 80, 81, 82]. These encodings grow very rapidly as the number of possible elements is increased. The population size required to avoid a deceptive sample of the design space can become prohibitively large. The members of the initial, randomly-generated population will, on average, include half of all possible elements. When the best design uses only a small fraction of all possible elements, the genetic algorithm is forced to

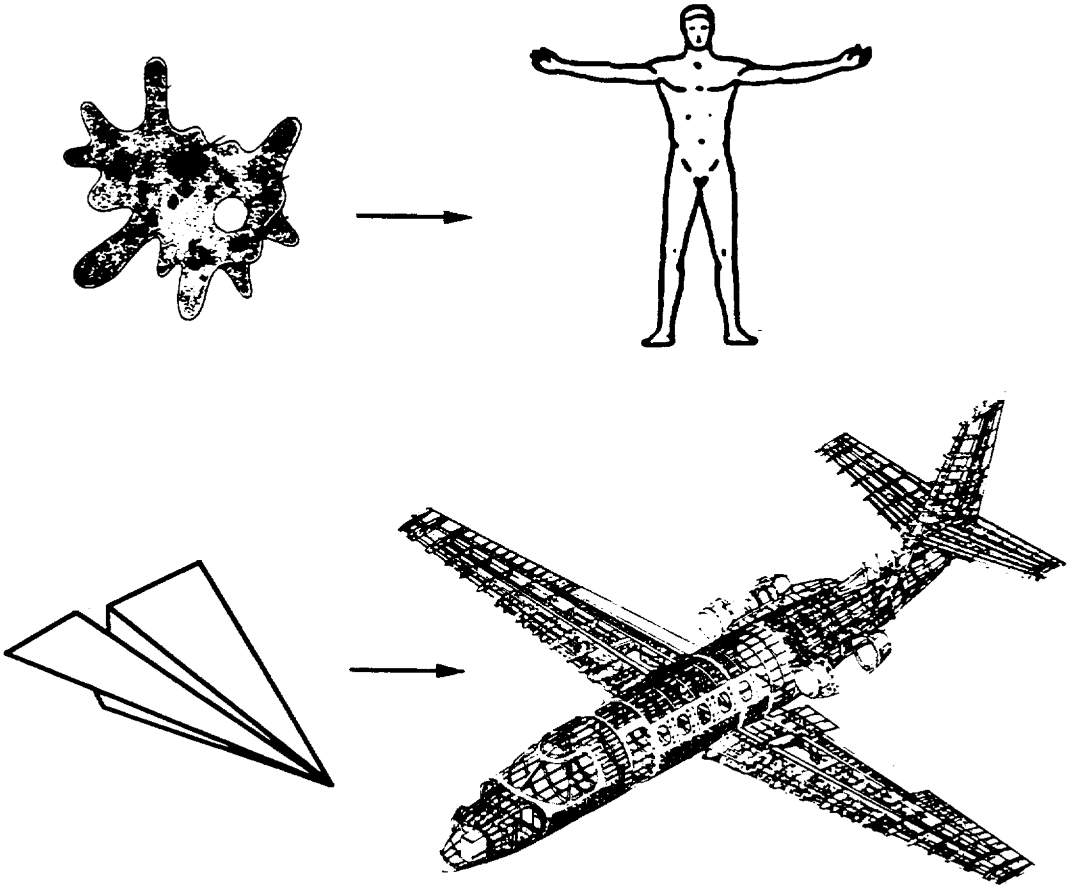


Figure 4.1: Natural evolution, like aerospace vehicle design, proceeds from simple descriptions to complex specialization.

consider designs of unnecessary complexity, which reduces the efficiency of the search.

The existence or absence of possible elements can be described implicitly, by using a genetic string that carries information only about elements that actually appear in the design. Elements in the string exist, elements not in the string are absent. This encoding requires a variable-length genetic string, because the length of the string reflects the complexity of the design. Although variable-length encodings are unusual in genetic search, several investigators have been attracted by their expressive power. They have been used for development of rule sets [83], for automatic generation of computer programs [59], and for function optimization [86, 87]. Important prior implementations of variable-length strings, and the current scheme, are briefly discussed in the next section. Modifications to the standard genetic operators are introduced, and then a simple application demonstrates the power of the variable-complexity algorithm.

4.2 Variable-Length Encodings

4.2.1 Prior Use

Smith developed a learning system, LS-1, that uses genetic operators to evolve high-performance rule sets of varying size [83]. Each individual in the population is a set of rules, and the recombination of rules from different parents generates new rule sets of superior performance. This representation contrasts with the classifier system developed by Holland et al., where individuals are isolated rules, and the entire population constitutes the rule set. The classifier system relies on competition between population members during performance evaluation to generate useful groups of rules, whereas LS-1 uses genetic search to identify both individual rules and groups of rules. (De Jong notes that Smith's formulation outperforms classifier systems when extensive exploration is permissible and radical changes are acceptable [85]. This observation indicates that the variable-length representation is appropriate for conceptual design, while fixed complexity classifier systems might be more appropriate for the detailed design phase.) The size of the best group of rules is not known *a priori*, so rule sets of varying size are permitted. For LS-1, individuals that contain different numbers of rules must use encodings of different length. To validate the use of genetic search with variable-length strings, Smith performed a hyperplane analysis. It shows that promising building blocks are appropriately retained and recombined, provided that a steady state average length is achieved. This analysis is an extension of Holland's schema theorem, which provides a theoretical basis for the genetic search mechanism applied to fixed-length strings [84].

In genetic programming, the individuals in the population are computer programs. The user specifies a set of terminals (parameters and constants) and a set of functions (e.g. arithmetic, mathematical, logical, domain-specific), and random combinations are chosen to compose the programs (of varying length) for the initial population. Thereafter, selection and crossover are used to recombine features from relatively fit programs to produce offspring of higher performance. Koza cites the empirical evidence of successful applications in a variety of fields as proof that genetic adaptation of variable-length strings is a valid search mechanism [59].

Messy genetic algorithms have been developed by Goldberg et al. [86, 87] for use in function optimization. Variable-length strings are introduced in an attempt to avoid deception that can arise when promising building blocks have a large defining length in the chosen encoding. By removing position-dependence from the encoding scheme, and allowing each variable to occur anywhere along the string, the genetic operators are able to discover strings that arrange the building blocks in a non-deceptive order. The variable-length strings may have several representations of some parameters, and no representation of other parameters. A simple conflict resolution scheme is used to choose a single value for overspecified parameters, and 'competitive templates' are used to provide values for underspecified variables. This means that the decoding scheme always provides values for a fixed set of parameters that describe all candidate solutions, and messy genetic algorithms do not provide the variable complexity that is sought for the preliminary design system.

4.2.2 An Encoding Scheme for Conceptual Design

The present scheme provides the functionality of genetic programming for function optimization, but the genetic string is a list of parameters, rather than a composition of functions and terminals. The initial population is constructed by selecting random combinations of parameters from a user-specified set, and the solution is a set of parameters rather than a computer program. In contrast to messy genetic algorithms, there is no template used during decoding to produce designs of standard form. Candidates of variable complexity are admitted.

There is position dependence in this encoding, but it is relative position rather than absolute position that is important. Each new parameter influences the design that has been constructed from previous parameters, either by adding elements to the design or modifying existing elements. The decoding of each parameter therefore depends on the decoding of parameters that occur earlier in the string. Similar position dependence occurs in the construction of computer programs, where the operation of each instruction is influenced by prior instructions.

Variable-length strings make it easier to recognize promising building blocks in the design problem. The initial population has random values for each variable, so the likelihood of a poor value for at least one variable increases as the number of variables increases. In problems where a single poor variable radically affects the performance of the entire design (as is likely in aircraft design, which is known to be very tightly coupled), random long strings are unlikely to look attractive. If the genetic algorithm starts with short strings, the population is more likely to include members that have reasonable values for all variables. Once these building blocks of short defining length have been identified, the string can be extended to describe more complex designs. This parallels the motivation for messy genetic algorithms, where deception is avoided by using short strings that omit variables that would otherwise extend the defining length of promising building blocks. The competitive template used with the messy genetic algorithm is not required here, because strings of any length describe complete individuals that can be evaluated.

4.2.3 An Extended Encoding Scheme for Varying Constraint Activity

When individuals of different complexity are allowed in the population, and there is a trend toward increasing complexity, it is possible for constraints that are initially inactive for the most promising individuals to become active in later generations. If variables have been selected without regard to a constraint that has been inactive, they may have converged to values that are inappropriate when the constraint is active. Without diversity, the population is unable to adapt to the changed conditions.

Biological precedent suggests an approach by which adaptation can be encouraged [77]. Only a small fraction of the genetic material in biological systems is used to construct the organism. The locations along the string where decoding starts and stops are controlled by regulatory genes, which respond to environmental influences when determining whether their piece of the genetic string should be decoded or not. A switch in activity of even a single regulatory gene can have profound influence on the final organism.

There are two important aspects of this system to be incorporated in the decoding of variable-length genetic strings: unexpressed sections, and a switching mechanism affected by the environment to control the expression and suppression of different sections. In the simplest form of this scheme, the genetic string is extended to carry two values for each design variable. Selection of the copy to be expressed is controlled by constraint activity. Initially, the first value is expressed, and selection of promising building blocks proceeds as usual. There is no selection pressure on the unexpressed values, so they remain randomly distributed. When expression of the first value causes a constraint to become active, the value is ignored, and selection operates on the second value. The first value is retained, because it can be expressed in future generations if it is read when the constraint is not active. The extended genotype is able to produce different phenotypes in different environmental circumstances. The structure built from early sections of the string causes environmental changes that affect the activity of later sections of the string. This influence of environment on expression of each part of the genetic string is a special case of position dependence.

4.3 New Genetic Operators

The modified genetic algorithm, which permits the use of variable-length encodings, is illustrated in Fig. 4.2. The basic operators for selection and mutation that are used in the standard genetic algorithm do not need to be changed for application to variable-length strings. The selection operator is not directly related to the string, because it works with the fitness value, which is calculated after decoding. Tournament selection is used, as described in Chapter 3. The mutation operator is applied at every location along the string, with a constant small chance of causing a change at each point. Consequently, the number of points in the string does not affect the mutation operation, but it does change the likelihood of a modification occurring somewhere along the string (with longer strings being more likely to be modified).

The key change is the introduction of a new crossover operator, which is modelled on the one used in genetic programming [59]. It is similar to single point crossover, because each parent is broken at only one point. The important distinction is that the crossover point may be different for each structure, so two parents of equal length may produce offspring that are longer or shorter. This allows the individuals to grow and shrink. If a new string produced by unequal crossover exceeds the maximum allowed length, it is truncated at the maximum length.

A little care is required to ensure that offspring produced by unequal crossover are viable, particularly for a binary representation. The crossover point in the second parent is not completely free. It may occur in any substring, but within the substring it must match the location of the first crossover point. This restriction ensures that offspring contain regular substrings that can be decoded correctly. Thus, in the example shown in Fig. 4.3, the second crossover point

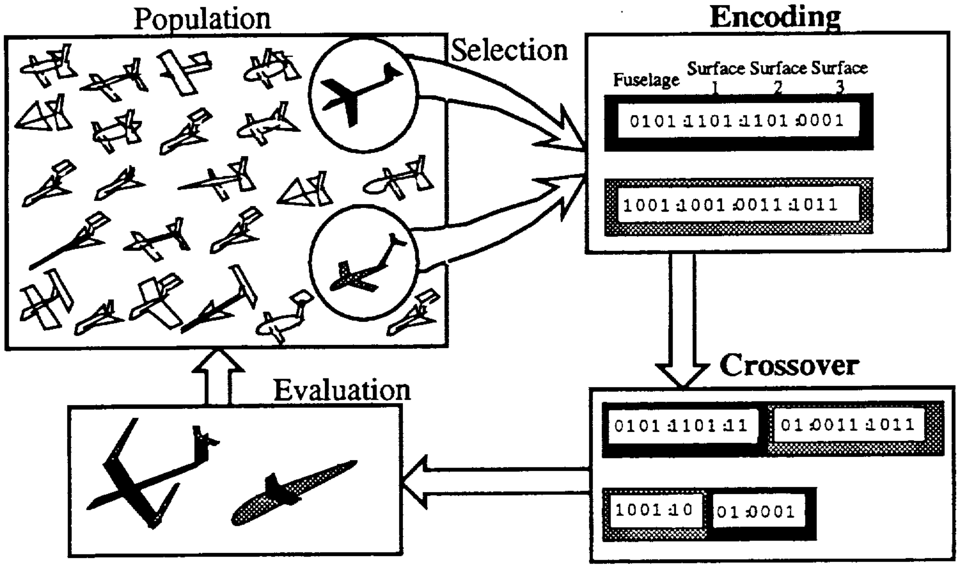


Figure 4.2: A variable-complexity genetic algorithm

can occur in the first, second, or third substring, but it must occur after the second bit of the substring, so that the substrings in the offspring all have eight bits.

The sharing and mating restriction operators, which were described in Chapter 3, use a distance measure between pairs of individuals in the population to determine their mutual influence. With variable-length strings, the distance is not defined when strings do not contain matching sets of variables. The metric for determining mutual influence must be modified if a sharing function is to be used in the new algorithm. Further investigation of these advanced operators is not included in this thesis. A genetic algorithm with only the fundamental operators of selection, crossover and mutation is used in conjunction with variable-length encodings.

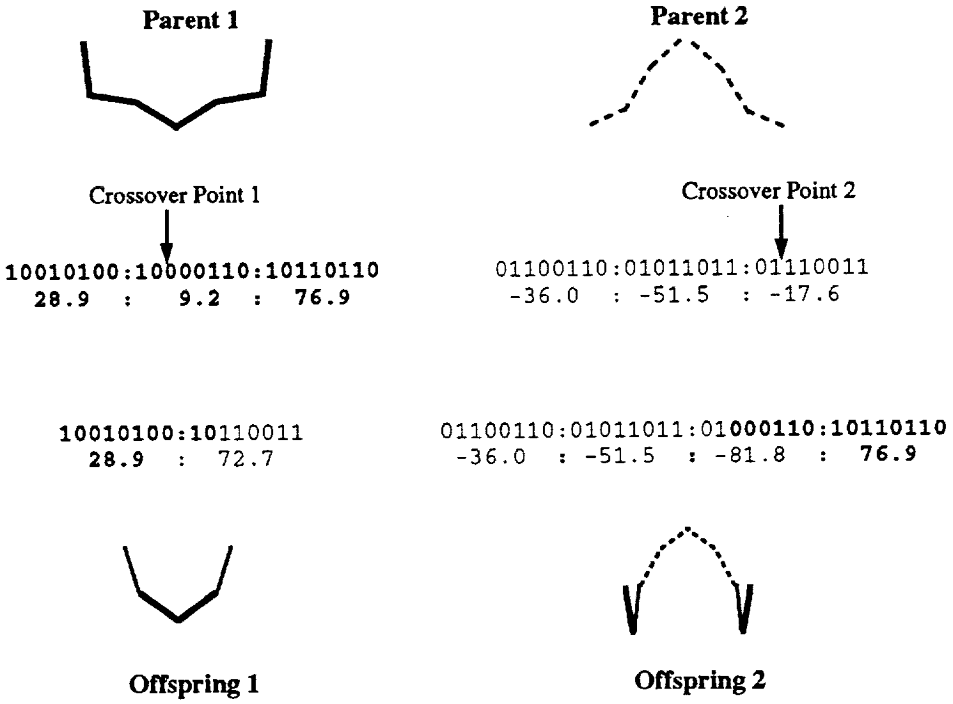


Figure 4.3: Modified crossover operator

4.4 Application to a Block-Stacking Task

The new genetic algorithm is first tested in a block-stacking problem. This example is selected because the candidate designs are constructed from many similar elements, so it is convenient to vary the complexity of the candidate designs simply by varying the number of elements. Function evaluations are inexpensive, so the efficiency of the method can be quickly evaluated.

The aim in this problem is to maximize the horizontal overhang x of a stack of blocks of height y, as shown in Fig. 4.4. The tower can collapse if the upper blocks are located too far beyond their supports, and performance is assessed for the stack that remains after toppling occurs. This objective is chosen because it makes the problem difficult for gradient-based methods by making the design space discontinuous. There are sudden jumps in objective value when incremental movement of blocks causes toppling.

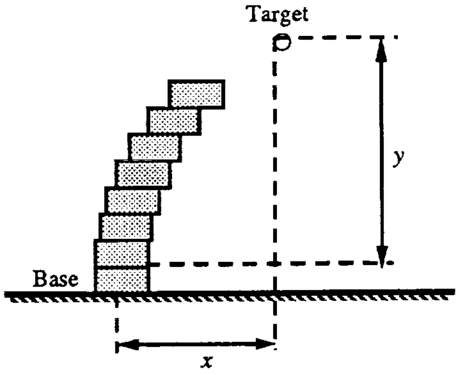
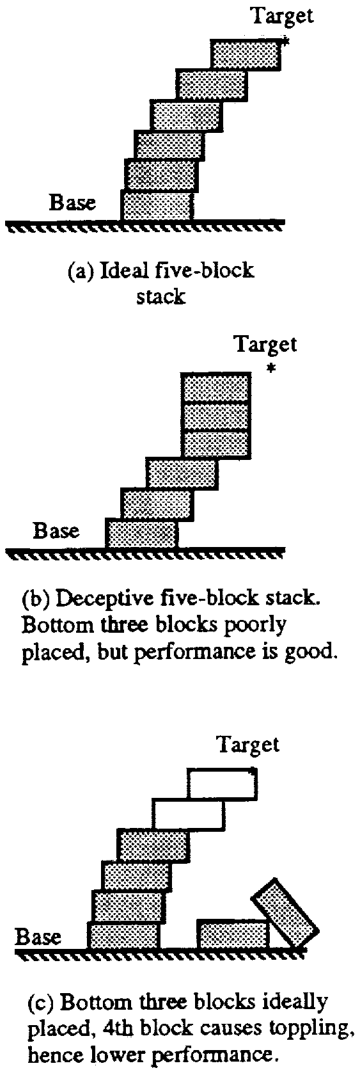


Figure 4.4: The block-stacking problem.

The block-stacking problem is difficult for a standard genetic algorithm also, as Fig. 4.5 illustrates. The stack that achieves significant overhang near the base is difficult to improve, because any shift toward the target will cause toppling. Its relatively high performance is deceptive, because it contains low-order building blocks that will combine to produce higher-order building blocks of low merit.

The modified genetic algorithm is able to counteract this effect, because the candidate individuals are not required to use all the blocks. Elements of the



(a) Ideal five-block stack

(b) Deceptive five-block stack. Bottom three blocks poorly placed, but performance is good.

(c) Bottom three blocks ideally placed, 4th block causes toppling, hence lower performance.

Figure 4.5: Deception makes poorly placed lower blocks appear to provide a good foundation for further stacking.

best solution may be developed in different short stacks, which can combine to form complete stacks later in the optimization. The new algorithm did not have any great advantage when the optimal stack was short, because there was little scope for the upper blocks to cause deception. The results presented here are for stacks that have a maximum height of 40 blocks, and the variable-length stacks are clearly superior. Figure 4.6 shows three histories for both algorithms, each starting from a different randomly generated population.

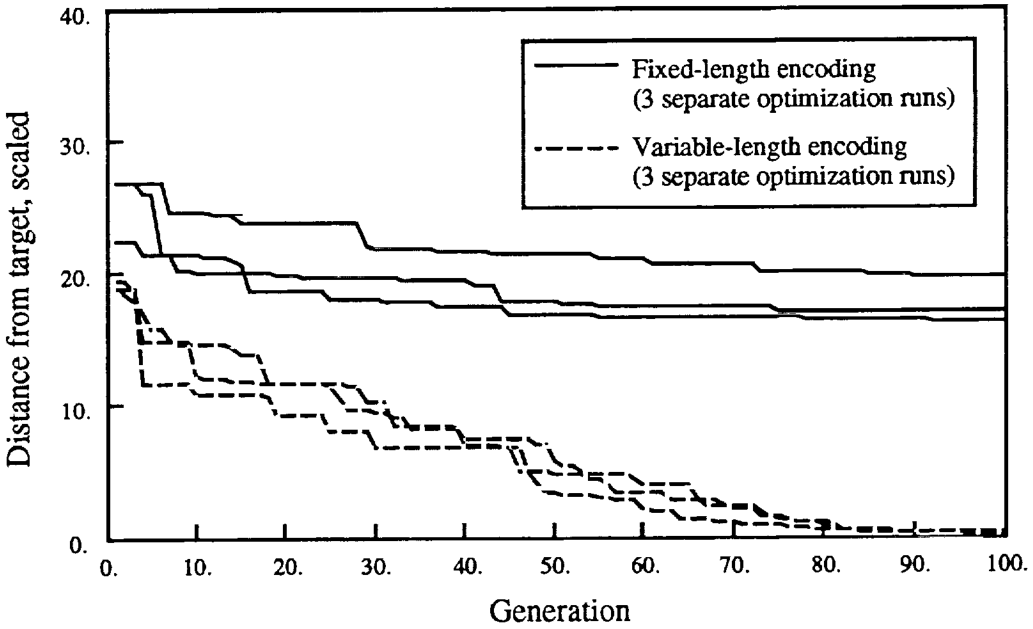


Figure 4.6: Fitness of best individual in population.

The algorithm with variable-length strings processes the building blocks of this problem more efficiently than with fixed length strings. A history of the growth of the best individual shows a strong trend towards a monotonic increase in size. Improvement occurs when stacks of intermediate fitness crossover unevenly to generate a longer string.

C-2.

A large part of the success in this particular problem arises from the capability of the system to shift building blocks to different locations in the string. In the exact solution, the top of the stack always looks the same, because toppling is only influenced by what lies above a particular block. The crossover mechanism allows the top of a small stack to end up on top of a larger stack. Figure 4.6 indicates that the performance of the variable-length stacks is superior even in the initial random population. A random short stack is less likely to topple than a random long stack, and the performance of untoppled short stacks exceeds the performance of toppled longer stacks. Figure 4.7 shows that rapid progress is possible once stacks exceed a height of about fifteen blocks. The base blocks for stacks of this height are close to vertical, with most of the horizontal distance being covered in the top few blocks. Unequal crossover that occurs close to the base can increase the height of the stack without being likely to cause toppling.

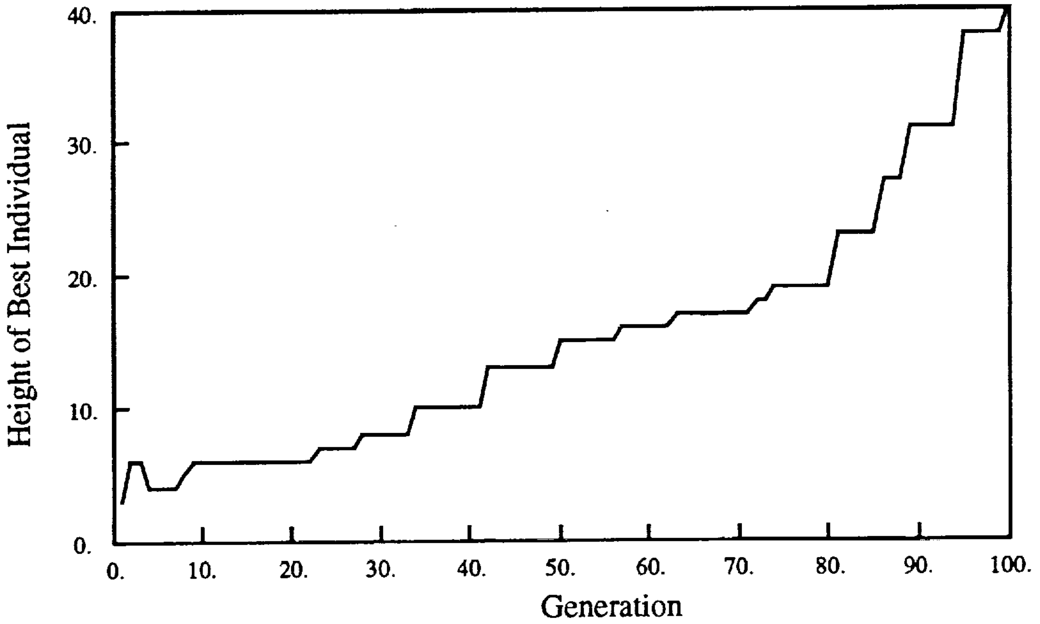


Figure 4.7: Growth in height of best individual. (40-block maximum height)

This growth mechanism indicates that this stacking problem is ideally suited to solution using a crossover operator that shifts the location of building blocks within the genetic string. The operator should prove useful in other applications in which complexity arises from the combination of many similar elements. In the next chapter, it is applied to the optimization of structural trusses, which are large assemblies of simple bars. In Chapter 6, the variable-complexity algorithm is used to minimize the drag of a wing that is modelled by a set of lifting elements.

Chapter 5

Topological Design of Structural Trusses for Minimum Weight

The variable-complexity genetic algorithm is now applied to the topological design of structural trusses. Other researchers have previously used standard genetic algorithms in this domain, so the performance of the new algorithm is directly compared with their results. A variable-length encoding that includes expert knowledge of the domain restricts the search space to the feasible region, so near-optimal truss topologies are efficiently discovered. A hybrid search scheme, with a calculus-based method to search the smooth subspaces, further improves the efficiency of optimization.

5.1 Introduction to Structural Optimization

The structural optimization literature identifies three kinds of structural design problems: sizing, shape and topological optimization [88]. At the simplest level, a sizing problem varies only the cross-sectional areas of a fixed number of components. A shape problem may add variables describing the location of components, so the proportions of each component in the fixed set may be altered. A topological problem must also include variables that refer to the existence or absence of components, so that candidate structures use different subsets of the set of all possible components.

For sizing and shape optimization, gradient-based techniques are often appropriate. It is possible to treat topological design as a sizing optimization task by starting with a ground structure that includes all available elements, and allowing some of them to vanish [89, 90]. For complex tasks, the computational burden of sizing large numbers of elements soon becomes unacceptable. Furthermore, this formulation generally produces a design space that includes many local minima, so convergence to the global optimum is problematic. Standard genetic algorithms have been used to address both of these difficulties, but the variable-complexity algorithm can further reduce the computational burden associated with unnecessarily complex candidate designs.

Although the variable-complexity method can be applied to optimization of general structures, attention is restricted here to trusses, because the analysis of these assemblies of pin-jointed bar elements is relatively straightforward. A plane truss structure is typically described by a set of nodes connected by pin-jointed bars that can be loaded only in tension or compression. Some nodes are prescribed to have zero displacement while others are acted upon by specified external loads. An optimum truss has minimum total weight, while satisfying stress constraints on the members, and displacement constraints on the nodes.

A finite-element method is used to compute the stresses in the bars, and the displacements of the nodes. The method is described in most modern textbooks on structural analysis (e.g. [91]). A global stiffness matrix is assembled using the local stiffness matrix of each element, and specified external loads and node constraints are included in the relevant vectors. The force displacement relation is used to solve for unknown loads and displacements. The weight of the structure is found by summing the volume of material of all elements.

5.2 Standard Genetic Optimization of Structural Trusses

The earliest application of genetic techniques to truss design was performed by Goldberg and Samtani [92]. They optimized the areas of a ten-member truss, with stress constraints for each member applied as quadratic exterior penalty functions. Their solutions on three separate runs, each performing 6400 function evaluations and starting from a different randomly-generated initial population, were within 2% of the optimum attained by a gradient method. Although the genetic algorithm was effective for this sizing task, it was not efficient, and the gradient method is really more appropriate.

More recently, Sakamoto and Oda [78] introduced a hybrid technique for truss design, with a genetic algorithm used for layout design and a simple gradient method used for sizing the cross-sectional areas. They found that the hybrid method had greater practical reliability than a member elimination strategy, where they defined practical reliability as the likelihood of finding a design with performance within 20% of the global optimum. The member elimination strategy often became stuck at local optima that were unnecessarily complex.

The genetic string used by Sakamoto and Oda for topological design is a concatenation of single bits, each representing the existence or absence of a possible element. Several other researchers have used similar schemes for genetic solution of structural optimization tasks [79, 80, 81, 82]. This means that the string length grows very rapidly as the number of nodes (and hence the number of possible elements) is increased. The population size required to avoid a deceptive sample of the design space can become prohibitively large.

Candidate designs generated by a genetic algorithm may be incomplete structures or mechanisms. Sakamoto and Oda identified these candidates and gave them fitness equal to the minimum in the population. This makes it difficult to identify useful building blocks, because it creates large regions with no gradation in fitness [55, 56, 57]. It is preferable to avoid description of mechanisms by employing an encoding scheme that generates only feasible candidates. Such a scheme is clearly biased, because it precludes description of many instances in the feature space. However, Mitchell [93] claims that biases that include factual knowledge of the domain (that candidates should not be mechanisms), and biases that favor simplicity, are useful aids to the learning process.

5.3 Variable-Complexity Genetic Optimization of Structural Trusses

The variable-complexity genetic algorithm provides an alternative approach for topological design. The genetic string need only carry information about members that actually appear in the design. They are described as general truss members with particular endpoints, rather than particular truss members with known endpoints. This approach is more flexible and efficient than exhaustive representation of all possible members when a large number of nodes is available.

In the encoding scheme chosen for truss design tasks, candidate solutions start from a baseline design and the genetic string describes modifications to be made. The baseline is a non-mechanism (a structure that does not collapse under the applied load) that uses the minimum possible number of members to attach all loaded nodes. Where several alternatives use the same number of members, the description with minimum total length of members is selected.

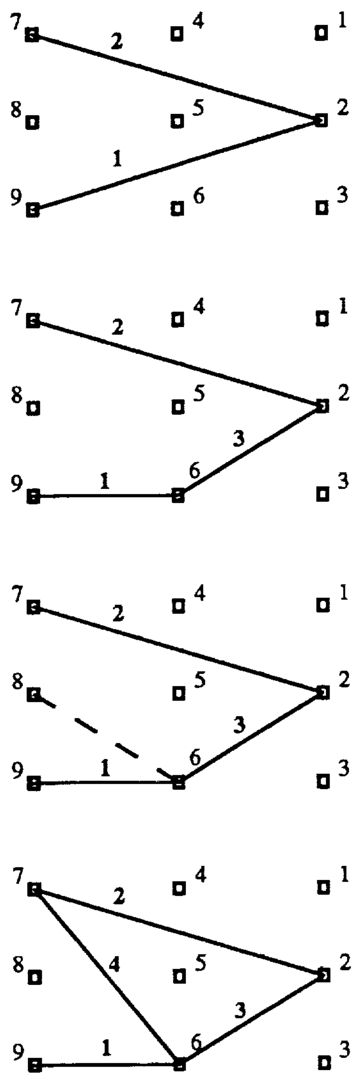
Each modification to the existing structure is described by a set of three variables. The first variable identifies an existing member that is to be split into two members. The endpoints of this member are retained as endpoints of the two new members, but a new endpoint must also be specified. The second variable identifies this new endpoint. When an existing member is replaced by two members in this manner, it is possible that the new structure is a mechanism, and therefore unable to support the applied load. A mechanism can be avoided by adding a new member to brace the structure, connecting the new node with a second node that is already connected to the structure. The third variable identifies a second node. If this node is not appropriately connected, a repair scheme shifts it to the nearest attached point.

The genetic string that encodes the series of changes to the baseline structure is a concatenation of sets of three variables, with each triplet describing a potential modification. Complexity of the structure increases as the length of the genetic string is increased, because each triplet splits an existing member into two, and adds an additional member. The decoding operation is shown in Fig. 5.1, which shows the effect of a single triplet on the baseline design. The number of nodes is fixed ($TotalNodes = 9$ in this example, represented by small squares in the figure). The number of elements ($Nelem$), or truss members, changes as decoding proceeds. The first variable in the triplet has a value between 0 and 1, and it is multiplied by $Nelem$ to find which member should be modified. The second and third variables have values between 1 and $TotalNodes$, and they refer directly to particular nodes in the grid.

Figure 5.1(a) shows the baseline structure to support a single load at $Node2$, with $Node7$ and $Node9$ as constrained supports. $Variable1$ is decoded to show that $Member1$ is to be modified. Decoding $Variable2$, as shown in Fig. 5.1(b), indicates that the new endpoint for the two members which replace $Member1$ is $Node6$. The member connecting $Node9$ and $Node6$ is now $Member1$, while the connection between $Node6$ and $Node2$ is $Member3$. An extra member must still be introduced to ensure that the structure will not collapse. Figure 5.1(c) shows that $Variable3$ is decoded to suggest $Node8$ as the new endpoint, but this node is not connected to the structure. The effect of the repair scheme is illustrated in Fig. 5.1(b), with the new endpoint shifted to the nearest attached point ($Node7$) to form a viable structure. The connection between $Node6$ and $Node7$ is labelled as $Member4$.

This decoding scheme is an example of a 'shape grammar'. Shape grammars provide a formal method for generating topologies. A particular grammar is defined by a set of shapes (straight truss members), labels (each member is numbered as it is added to the structure), shape rules (the repair scheme avoids mechanisms) and an initial shape(the simplest shape that supports the specified load). Reddy and Cagan [94] have used a different shape grammar in conjunction with a simulated annealing algorithm to develop complex truss topologies.

Triplet {Member : Node 1 : Node 2}
{ 0.4 : 6.3 : 8.7 }



a) **Member**
= int(Var1* Nelem)+1
= int(0.4 * 2) + 1
= 1

b) **Node1 = int(Var2)**
= 6

c) **Node2 = int(Var3)**
= 8

d) **Repair Node2**
Shift to nearest
attached node = 7

Figure 5.1: Decoding a triplet of the genetic string.

When the topology of the candidate structure has been determined by decoding the string, a gradient-based optimizer is used to size the members. Stress constraints for the members and displacement constraints for the nodes are handled efficiently by a sequential quadratic programming algorithm. In a hybrid scheme, the cost of each function evaluation for the genetic algorithm depends directly on the time spent on optimization in the smooth subspace. The work required for sizing increases linearly with the number of members (quadratically if finite-differences are used to produce gradient information). The variable-complexity algorithm typically generates candidates with a small number of members, so the time spent sizing them is greatly reduced.

This genetic scheme has been applied to a series of truss design tasks. The first two are used to provide direct comparison with the hybrid method employed by Sakamoto and Oda. The third allows comparison with an analytically-determined optimum for a Michell truss.

5.4 Applications

5.4.1 Nine nodes, one load point.

The first optimization task is to find a minimum weight truss to support an end load of 1000 Newtons, with a vertical deflection of 0.015 mm at the load point. A 3 x 3 grid of nodes is provided, with horizontal spacing of 100 mm and vertical spacing of 50 mm. The number of possible elements is 36 (for n nodes, the number of possible elements is $n[n-1]/2$), and Young's Modulus is prescribed to be 200 GPa. The density used to calculate truss weight is 0.0079 g/mm^3.

This task is quite simple for the variable-complexity algorithm. Only two triplets in the genetic string are required to shift from the baseline, which has two members connecting the load point and fixed nodes, to the optimal configuration, which has six members. The results of 5 trials, from different random starting populations, are shown in Fig. 5.2.

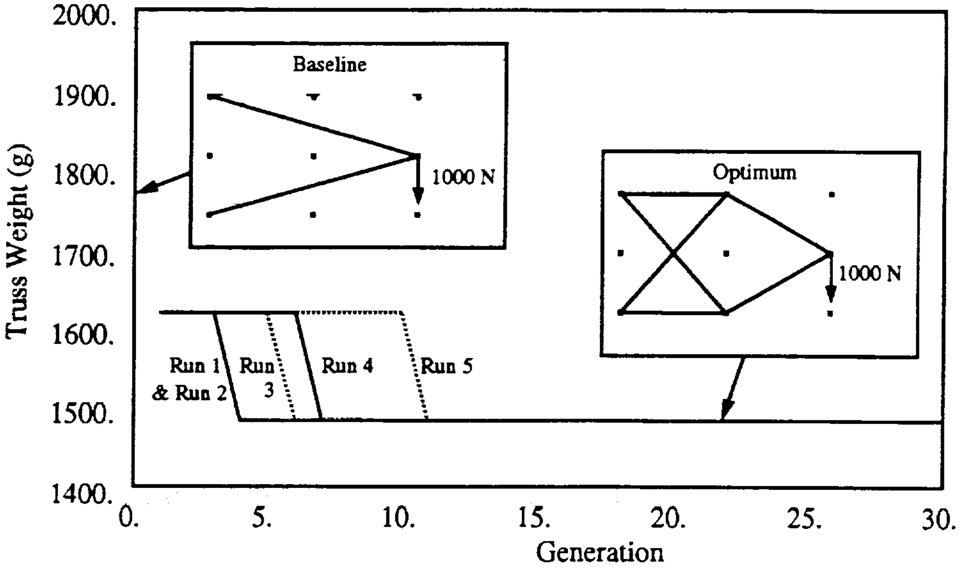


Figure 5.2: Optimization histories for single end load.

A population size of 50 is used, and the genetic algorithm is allowed to run for 30 generations. The true optimum is found within 500 function evaluations. In every trial using the variable-complexity algorithm, some members of the initial random population have one useful triplet. The optimum is reached as soon as a mating with unequal crossover is performed between two parents with different useful triplets, producing offspring that combine the best features of both parents.

Sakamoto and Oda reached a practical optimum in 96% of their trials, but even the two-member design used as a starting point for the variable- complexity algorithm satisfies their criterion for practical optimality. They are essentially reporting a success rate for finding a feasible design, whereas the encoding used here reduces the search space by guaranteeing feasibility. The average candidate design in their starting population has 18 members (half the bits in an average random string of length 36 will be ‘1’). Consequently, each function evaluation for their genetic algorithm is more expensive, because there are more members to be sized. The variable-complexity algorithm provides superior performance at lower computational expense.

5.4.2 Nine nodes, two load points.

In the second example, the 3 x 3 grid of nodes and the material properties of the members are retained. Two nodes are loaded, as shown in Figure 5.3. Vertical deflection at the load points is again constrained to a maximum of 0.015 *mm*. The baseline truss now needs four members to connect the loaded nodes to the supports.

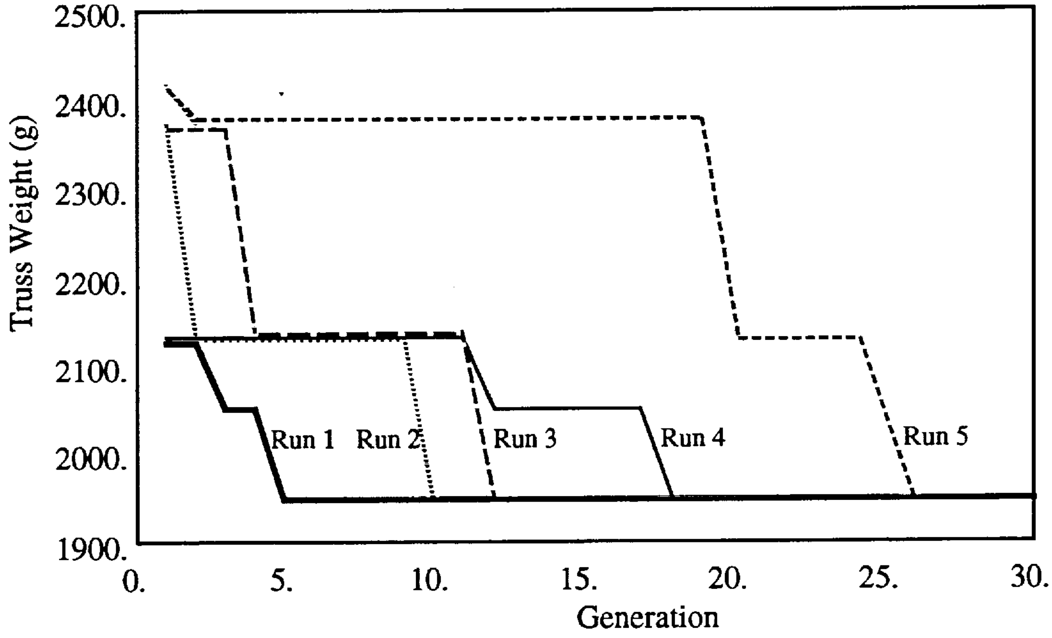


Figure 5.3: Optimization histories for two load points.

As in the first example, only two triplets are required to move from the baseline to the optimum, but there are more baseline members that can be modifed. The variable-complexity algorithm requires more function evaluations to locate the optimum in this problem, but still finds it within 2500 function evaluations for all trials. The hybrid method of Sakamoto and Oda has more difficulty in this problem, finding a practical optimum in only 80% of trials.

As before, candidate designs have extra members, so the sizing portion of the hybrid algorithm must do more work than is needed for the variable-complexity algorithm.

A history for the best population member for a sequence of generations shows the variety of designs encountered during optimization (Fig. 5.4). The trusses up to Generation 20 are built with a single modification of the baseline structure, while the trusses in later generations are built using two triplets in the genetic string. The optimum combines the modifications of Generation 2 and Generation 20. Once again the variable-complexity algorithm combines building blocks from simple parents to produce more complex offspring that have higher performance.

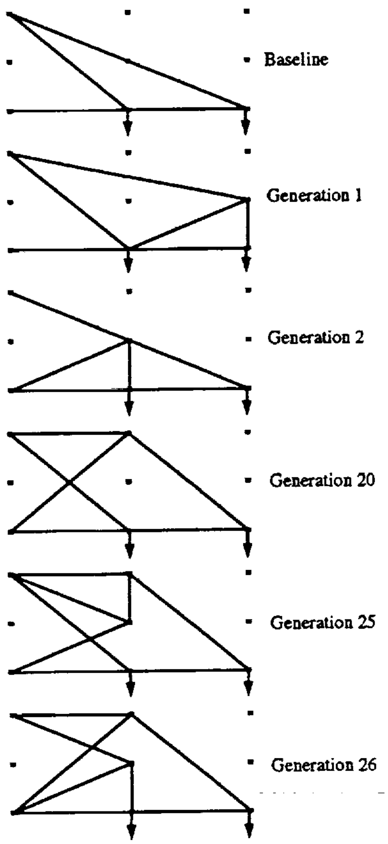


Figure 5.4: History of best individual in population.

5.4.3 Michell truss.

For a single vertical end-load that produces a force and a couple resisted by a supporting circle, the optimal truss topology is known. First described by Michell [95], the members lie along lines of principal strain, forming a series of spirals that intersect orthogonally, as shown in Fig. 5.5. The optimal members are curved, while straight bars are used in practice.

The optimal solution can be approached by providing a grid of reasonable density, so that several straight bars approximate the theoretical spirals. A 7 x 5 grid of nodes is used in this example, as indicated in Fig. 5.6. The base circle is approximated by prescribing zero displacement for 3 nodes. (The base circle is represented in Fig. 5.6 for completeness, but it is not directly modelled in the analysis).

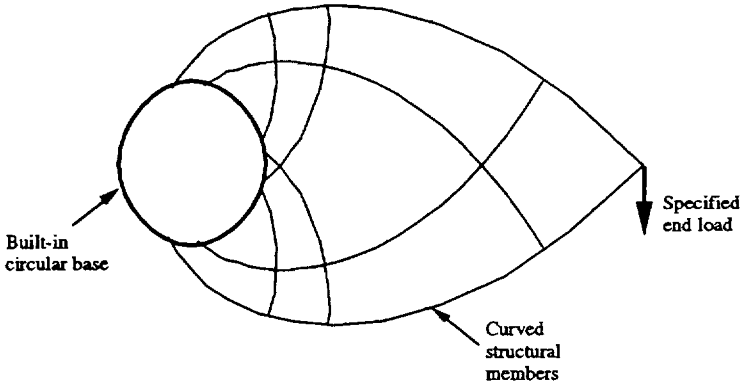
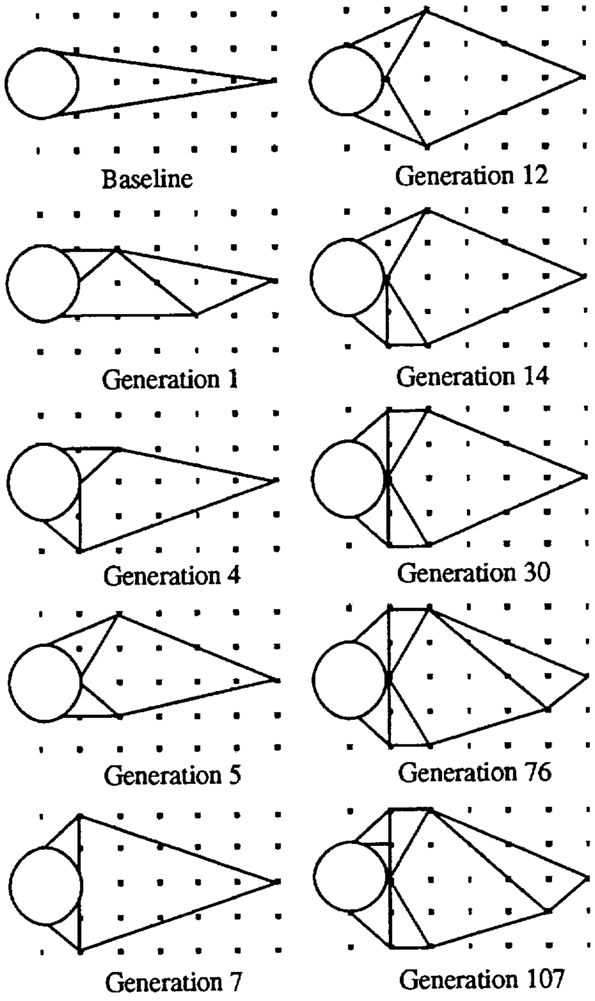


Figure 5.5: Michell truss for single end load.

The variable-complexity algorithm starts with a baseline structure that has two members, connecting the upper and lower supports to the loadpoint. Stress constraints are now imposed on all members, instead of displacement constraints on the load point, to allow direct comparison with Michell's theoretical solution.

The outer shape of the truss exerts a strong influence on the total weight of the truss, and a history of the best member in the population shows that trusses of maximum height are quickly found. Details of the internal structure are not completely determined when the algorithm is stopped after 150 generations. The final designs from five different runs are shown in Fig. 5.7. The result from Run


Baseline

Generation 12

Generation 1

Generation 14

Generation 4

Generation 30

Generation 5

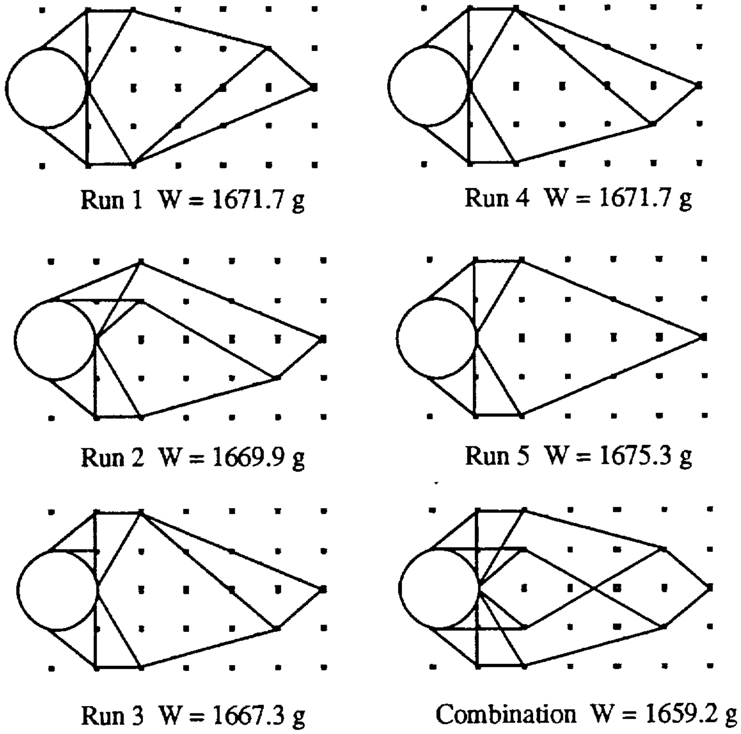
Generation 76

Generation 7

Generation 107

Figure 5.6: History of best individual in population.

5 is a simple, near-optimal truss that uses only ten bars. It is observed during every run. The first four runs produce designs that include up to four additional elements, which improve performance by up to 0.5%. No single design produced by the genetic algorithm includes all the features that appear in different runs. The designer is able to combine these features to seek superior performance. The last truss shown in Fig. 5.7 is generated in this fashion, and it yields a further 0.5% performance improvement. It uses eighteen bars, and closely approximates the shape of the curved optimal truss. Its weight is within 15% of Michell's solution.



Run 1 W = 1671.7 g

Run 4 W = 1671.7 g

Run 2 W = 1669.9 g

Run 5 W = 1675.3 g

Run 3 W = 1667.3 g

Combination W = 1659.2 g

Figure 5.7: Final designs from different runs.

A problem of this magnitude has not been solved using a standard encoding with a fixed length genetic algorithm. With 595 bits required to represent all possible members, a population of several thousand candidates would be needed to provide an adequate sample of the search space. Average candidates in the first generation would include 297 bars, an order of magnitude more complex than anything considered by the variable-complexity algorithm in the entire optimization procedure. The algorithm described here is far better suited to tasks of genuine engineering interest.

5.5 Summary

Genetic algorithms are slow to converge to exact optima. Solutions generated by the variable-complexity algorithm are often slightly simpler than the true optimum, but they capture its essential nature. Results of genetic optimization are most useful when several near-optimal designs are produced. The designer can combine features from different designs, or may prefer a slightly sub-optimal design due to considerations not modelled in the problem description. Gradient-based optimizers can be used in conjunction with a genetic algorithm, to achieve tighter convergence in smooth subspaces.

The encoding language used by a genetic algorithm can strongly influence its performance. Biases that reflect factual knowledge of the domain can prevent description of infeasible designs, and thereby restrict the search space to manageable size. Encodings of variable length, which are available to the variable-complexity algorithm, allow a bias towards simplicity. This generally reduces the cost of each function evaluation. The smaller search space and less expensive function evaluations both significantly improve the efficiency of genetic optimization, so that previously intractable tasks can be managed by the new variable-complexity genetic algorithm. Future work will include implementation of the shape grammar for truss development defined by Reddy and Cagan [94].

Chapter 6

Wing Topology Optimization for Minimum Drag

Several applications of genetic optimization to aerodynamic design of wings are presented in this chapter. These examples demonstrate that careful constraint-handling is critically important for location of the optimum. Penalty methods, repair methods, and analytic satisfaction methods are all used in this domain. When geometric constraints are imposed on the span and height of a non-planar wing, fixed-complexity representations converge to suboptimal designs. The geometric constraints are satisfied by folding the structure into the allowable space, and the details of the folding are strongly dependent on the initial topology. Successful identification of the optimum by the variable-complexity algorithm requires an extended encoding of the variables in the genetic string. This encoding allows adaptation in response to changes in constraint activity, as explained in Chapter 4. Although calculus-based methods do not produce optimal topologies when used alone, they can be helpful for final refinement of designs produced by the genetic optimizer.

6.1 Motivation for Use of Genetic Optimization

Systems for preliminary synthesis studies have traditionally used algebraic relationships and heuristic rules to size aircraft components [3, 5, 6, 7]. With steady increases in the computational power available to designers, it has been possible to introduce panel methods for more accurate aerodynamic estimates at the preliminary design stage [4, 28, 96, 97]. It has been observed that the drag predicted by these methods is strongly influenced by panel geometry. If continuous variation of the width of each panel is permitted, the aerodynamic response can be noisy, and estimation of gradients for calculus-based optimization becomes difficult.

Gallman [4] smooths the drag prediction by constraining all panels to have the same width. This is effective, but it limits the choice of variables, because parameters such as fuselage width and wing span are no longer independent. Unger and Hall [99] have attempted to smooth the gradient estimation by using automatic differentiation, but found that convergence difficulties were not completely resolved. Giunta et al [98] have introduced response surface approximations for the same purpose, but note that the number of function evaluations required to provide data for curve-fitting can be prohibitive for large problems. No such efforts to eradicate noise from the aerodynamic analysis are required when genetic optimization is employed, because gradient information is not required.

In recent years there has been significant interest in highly nonplanar geometries for very large aircraft. When the total arc-length of the wing is greater than the maximum allowable span, the wing must be folded to satisfy the geometric constraints. If the arc-length is fixed throughout optimization, the folding is generally not optimal. If the arc-length can be increased during optimization, folding occurs only when the wing encounters a constraint, and is therefore more likely to be appropriate. The variable-complexity genetic algorithm provides the flexibility to alter the arc-length in this manner.

6.2 Aerodynamic Analysis of Lifting Surfaces

A vortex-lattice code [100] is used to calculate the loads on the wing, with drag being computed by integration in the Trefftz plane. In this code, the wing is represented by a number of panels, with a bound vortex located at the quarter-chord of each panel. The system solves for the vortex strengths that produce zero normal flow at control points, located at the three-quarter-chord. The panels are grouped into *elements*, with all panels in an element sharing the same incidence and dihedral. Figure 6.1 shows a wing of eight rectangular elements, with the control point of each panel shown at the three-quarter chord point, and the bound vortices lying along the quarter-chord line.

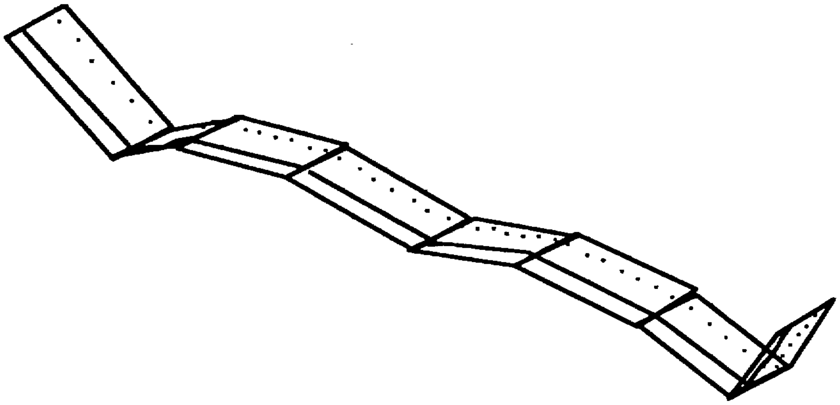


Figure 6.1: Panel representation of lifting surface.

The aerodynamic model solves the linear system of equations:

$$[AICS]\{\Gamma\} = U_\infty\{\theta\}$$

The aerodynamic influence coefficients [AICS] are computed using the Biot-Savart law, and represent the strength of the downwash at control point i due to the existence of a unit strength vortex at panel j.

All the design tasks that are explored in this chapter require the calculation of both lift and drag, but no consideration is given to wing weight. The objective is always to minimize drag while generating specified lift. Later examples also have geometric constraints on span and height of the non-planar wing. The design variables all relate to the geometry of the lifting surfaces. Dihedral is used in all problems. Incidence is also a variable, unless the constraint-handling scheme solves for it directly (as explained in the next section). Number of panels per element is used when a span constraint is introduced. The chord of each element becomes a design variable when parasite drag is included in the objective function. The size of the optimization task scales with the number of elements used to describe the wing.

6.3 Genetic Encoding

The genetic string is a concatenation of sets of variables, with each set describing a lifting element. *Dihedral* is always included in the set, while *Incidence*, *Number of panels* and *Chord* are added as required. To ensure viable decoding when unequal crossover is performed, the crossover point in the second parent must lie at the same point within an element description as the crossover point in the first parent. Thus, when the first crossover occurs at a *Dihedral* variable, the second must occur at *Dihedral* rather than, say, *Number of panels*.

The decoding of the genetic string assumes that the elements are attached end-to-end. The position of each new element is determined by the position and orientation of elements decoded before it. The assumption of connectivity reduces the chance of non-viable candidates being described. The search space is thereby reduced, so the population size required to provide a useful sample of the domain is also reduced.

Each set of variables in the genetic string can be decoded to modify an existing element or to add a new element. Modification of existing structure is permitted so that near-optimal topologies described by a few large elements can be refined by providing more detailed description of each element. When parasite drag is important, for example, performance improvement can be achieved by splitting a single element with constant chord into two elements with differing chord. A negative value for the *Number of panels* variable indicates that old panels are to be adjusted, while a positive value causes new panels to be added.

An extended genetic string is used for problems that include constraints on span and height. It carries three alternative values for each parameter, but only one is expressed. The motivation for a string with unexpressed sections was discussed in Chapter 4. The mechanism for controlling expression of alternative values is described in the next section of this chapter.

6.4 Constraint-Handling

6.4.1 Geometric Constraints to Permit Analysis

Some arrangements of lifting surfaces are difficult to analyze using panel methods. When vortices approach control points too closely, the matrix of influence coefficients can become ill-conditioned. If two panels lie directly on top of each other (as happens, for example, when the dihedral of consecutive elements is different by 180 degrees) the system is no longer independent, and there is no unique solution. These limitations require the introduction of extra constraints on the geometry of the candidate planform. Prior to aerodynamic analysis, the geometry is checked to ensure that vortices and control points are well-separated, that panels do not lie on top of each other, and that panels do not cross over each other. Candidate designs that fail these checks cannot be analyzed by the vortex-lattice code, so an alternative method must be used to assess their fitness. It is possible to assign a fixed value to all non-analyzable designs, but correlation of strings with performance becomes difficult if much of the population shares the same fitness. In this situation, the designs that cannot be analyzed are discarded from the population, and reproduction is continued until the population is filled with candidates that have been analyzed.

6.4.2 Lift Constraint

The requirement for fixed lift cannot be handled by excluding from consideration those designs which violate it. It is extremely unlikely that a wing with randomly chosen incidence values would satisfy this constraint, so the search for feasible members of the initial population would be extremely inefficient. Exclusion from the population is not required, because techniques are available for analyzing the infeasible candidates, and then either supplying a performance measure that reflects the extent of infeasibility, or repairing the design to make it feasible. Several possibilities are considered in this section.

Penalty Method

The standard method for handling constraint violations is to modify the objective function by appending a penalty that grows as the extent of infeasibilty increases.

$$J = J_{uncon} + PenWt \times (L - L_{req})^2$$

Here, J is the objective value, J_{uncon} is the unconstrained objective value, $PenWt$ is a user-specified weighting factor, L is the lift generated by the candidate design, and L_{req} is the required lift.

Numerical investigations show that when penalties are used to enforce the lift constraint, the strength of the applied penalty has a significant influence on the optimization history. Typical histories are plotted in Fig. 6.2, for a wing described by two elements and for a population size of 50. The number of panels per element is fixed, so there are four design variables: dihedral and incidence of each element.

A penalty weight of 2 is just sufficient to offset the induced drag benefit of violating the constraint. Stronger penalties make it more difficult for the genetic algorithm to find the optimum. Building blocks that help to satisfy constraints are selected over those that reduce the objective, so that the best building blocks may disappear from the population. Increasing the population size reduces this sampling error, but greatly increases the number of function evaluations required. Clearly, minimum penalty strengths are to be preferred.

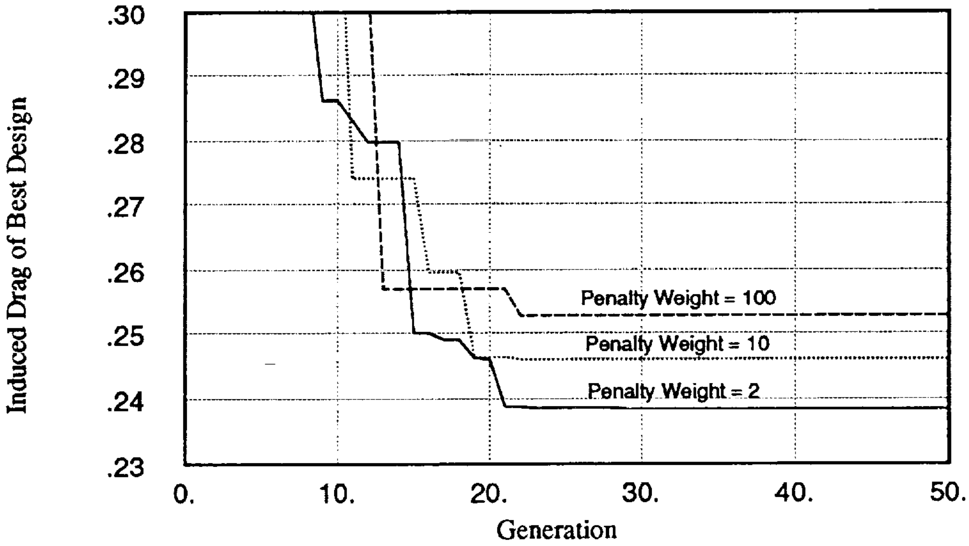


Figure 6.2: Influence of penalty weight

Repair Method

The repair scheme used to satisfy the lift constraint is essentially a sub-optimization in one variable. The incidence of each element is referenced to angle-of-attack (previously fixed, implicitly, at zero degrees). The candidate design is repaired by adjusting angle-of-attack to produce the correct lift.

Table 6.1 indicates the number of function evaluations requiredto find the optimum wing (which has maximum span, and elliptic spanwise lift distribution). The genetic algorithm was run three times on each problem, with a different randomly-generated starting population for each run, and the maximum number of function evaluations is listed in the table. Because there is no convergence criterion for the genetic algorithm that is equivalent to the Karush-Kuhn-Tucker conditions used by gradient methods, it is usually run for a fixed number of iterations, or until there has been no significant improvement for many generations. For these tests, the runs continued until the known optimum was found, but in practice the genetic algorithm is more likely to be stopped when the best design is not quite optimal.

	Genetic Algorithm				Sequential Quadratic Programming	
# Vars	C_L by Penalty		C_L by Repair		C_L by Constraint	C_L by Repair
	Pop	Evals	Pop	Evals	Evals	Evals
2	50	750	50	162	22	30
4	100	3636	50	939	86	92
8	200	14248	50	2065	303	350
16	500	64127	50	6935	819	582

Table 6.1: Function evaluations for different search methods.

These results make it clear that the repair scheme for the lift constraint is much more efficient than the penalty scheme, particularly as the number of design variables is increased. Problem size is changed by increasing the number of elements used to describe the wing, and reducing the panels per element so that the total number of panels is constant. The incidence and dihedral of each element are design variables. As the problem size grows, it becomes increasingly difficult for the penalty method to find a combination of variables to satisfy the lift constraint, and the required population size must be increased to accommodate this search. When the repair scheme is used, the entire population always satisfies the lift constraint, and a small population still provides an adequate sample for the larger problems.

Table 6.1 also presents the number of function evaluations required by a sequential quadratic programming algorithm. The genetic algorithm uses an order of magnitude more function evaluations for all problem sizes considered, even with the best constraint-handling. (Note that the sequential quadratic programming method is almost unaffected by the representation of the lift constraint. In fact, explicit constraints are usually more efficient for that optimizer [32].) There is no indication of a scaling advantage for the genetic algorithm. When gradient methods can be used, they provide better performance than genetic methods.

Analytic Solution of Optimal Incidence Distribution

The concept of using sub-optimization to satisfy the lift constraint can be extended, to include all incidence variables. This scheme has the further advantage that it minimizes total drag for a specified arrangement of panels. Performance is evaluated using the MULTOP program [101], in which linear equations representing the relations between vortex strength and parasite drag, induced drag and lift are constructed:

$$D_p = D_0 + \{D_1\} \cdot \{\gamma\} + [D_2]\{\gamma\} \cdot \{\gamma\}$$

$$D_{ind} = [DIC]\{\gamma\} \cdot \{\gamma\}$$

$$L = \{LIC\} \cdot \{\gamma\}$$

where

γ = vector of spanwise circulation strengths

D_p = parasite drag

D_{ind} = induced drag

DIC = array of drag influence coefficients

LIC = vector of lift influence coefficients

The objective for the sub-optimization problem is to minimize total drag, while constraining lift to a specified value, L_{req}. Using a Lagrange multiplier to handle the lift constraint, and defining $[DIC'] = [DIC] + [D_2]$, the objective is given by:

$$J = [DIC']\{\gamma\} \cdot \{\gamma\} + \{D_1\} \cdot \{\gamma\} + D_0 + \lambda_L([LIC] \cdot \{\gamma\} - L_{req})$$

The optimum occurs when the gradient of the objective function is zero. Differentiation produces the following system of linear equations:

$$\begin{bmatrix} \left[\, DIC' + DIC'^T \,\right] & \{LIC\} \\ \{LIC\}^T & 0 \end{bmatrix} \begin{Bmatrix} \{\gamma\} \\ \lambda_L \end{Bmatrix} = \begin{Bmatrix} \{-D_1\} \\ L_{req} \end{Bmatrix}$$

Solution of this system yields the vortex strengths that produce minimum total drag subject to the lift constraint. Search by the genetic algorithm is limited to dihedral and number of panels for each element. This implementation of the lift constraint improves the efficiency of genetic search, so it is used in the optimization studies that are reported in the remainder of this chapter.

6.4.3 Span and Height Constraints

Without the imposition of geometric or structural weight constraints, the optimal wing has very large span, and an elliptic chord distribution that produces the optimal local lift coefficient everywhere. Therefore, span and height constraints are introduced to force the wing to lie within a box as shown in Fig. 6.3, so that results will be of practical interest. These constraints are enforced using a penalty method, with the penalty chosen to just offset the drag benefit of violating the constraint.

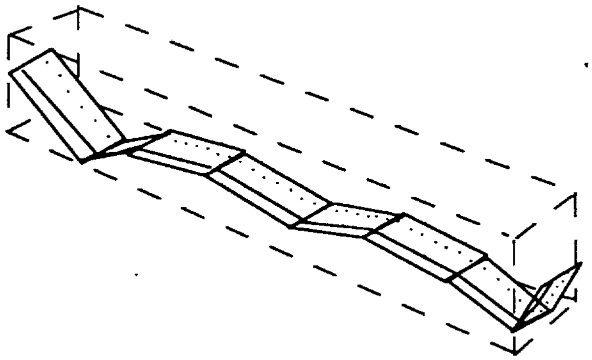


Figure 6.3: Wing topology design with span and height constraints.

When the variable-complexity algorithm is used, these new geometric constraints are not always active. If the population stabilizes at a particular design concept while they are inactive, it may lack the diversity needed to successfully respond to a changed environment when the constraints become active. For this wing design problem, elements with low dihedral, that combine to approximate a planar wing, are initially favored. When the span constraint is encountered, low-dihedral elements are no longer helpful. If they have dominated the population to the exclusion of all alternative dihedral values, the genetic algorithm must rely on mutation to identify further improvement.

To ensure that there is sufficient diversity to handle activation of the span and height constraints, the genetic string is extended to include three copies of each design variable. Figure 6.4 shows how the string is decoded to construct a candidate wing design.

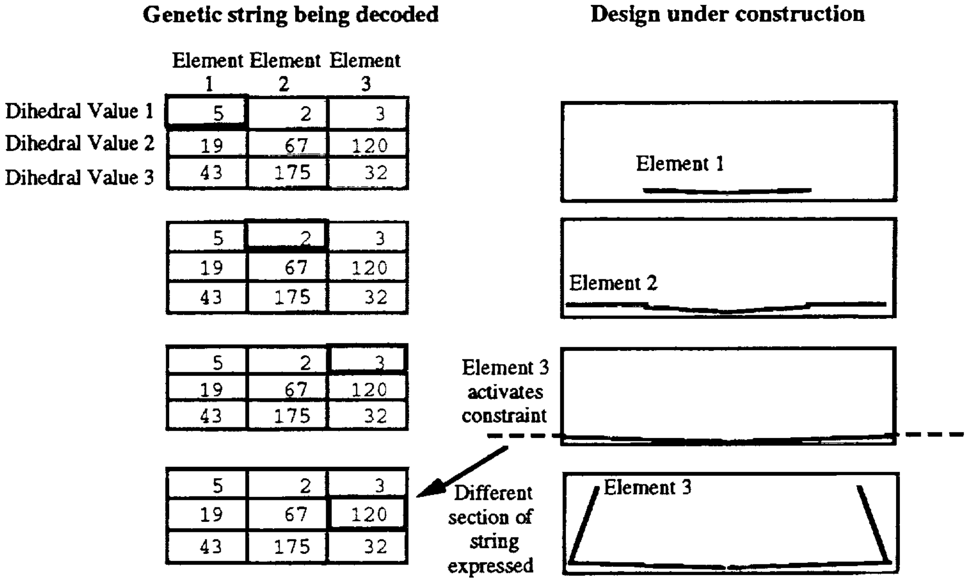

Genetic string being decoded
Design under construction
Element 1
Element 2
Element 3
Dihedral Value 1
Dihedral Value 2
Dihedral Value 3
5
2
3
19
67
120
43
175
32
Element 1
Element 2
Element 3 activates constraint
Different section of string expressed
Element 3

Figure 6.4: Decoding the extended genetic string.

The left side of the figure shows the encoding, with the variable currently being decoded indicated by a bold border. On the right is a front view of the wing being constructed, with the span and height constraints marked by the exterior box. Wing elements are shown in bold, and labelled as they are added. When no geometric constraint is active, the first copy of each variable is expressed. When the span constraint is encountered, the second copy is used. If the height limit is reached, the third copy is decoded. This arrangement allows different selection pressures to operate in different environments. The importance of this scheme is illustrated by the comparison of results for standard and extended encodings, included in the next section.

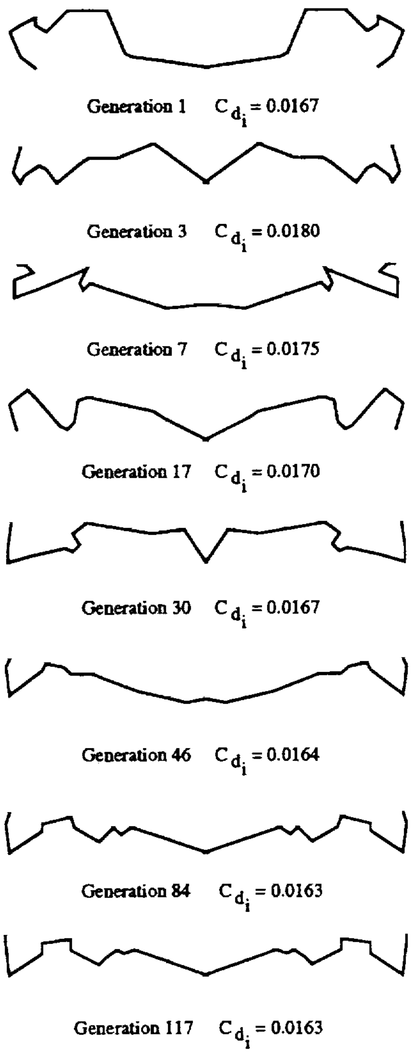
6.5 Optimization Results

6.5.1 Minimum Induced Drag

Initially, only induced drag is considered, because the theoretical optimum is known to be a box wing [102]. For the chosen height-to-span ratio of 0.1, the induced drag for the optimal wing is 21% lower than the drag for an elliptically-loaded planar wing. Chord does not affect induced drag, so it is removed from the set of design variables for this case.

A standard genetic algorithm operating on a fixed number of elements has difficulty finding any wing that can be analyzed. Each wing has 12 elements, so the combination of random dihedral values is very likely to position vortices from one element close to control panels from another. Even when wings can be analyzed, the algorithm struggles to find any designs that fit inside the box. The average element in the initial population has 4 panels, so the average wing has 48 panels to fit inside a box of semi-span 20 panels and height 4 panels. Elements of low span and moderate dihedral are favored. The selection pressure to satisfy the geometric constraints opposes the pressure to minimize drag. When the constraints have been satisfied using all available elements, it is difficult to allow further extension of the wing. Any extension can only be achieved by increasing the span of an existing element, and this is likely to produce constraint violation. This process is illustrated in Fig. 6.5, which gives a history of the best individual design in the population at various stages in the optimization run.

For the variable-complexity algorithm, the average element in the initial population also has 4 panels. However the average wing has only 6 elements, and a total of 24 panels to fit within the box. It is much easier to satisfy the geometric constraints, and consequently there is more freedom to work on drag minimization. If diversity is lost from the population too early, though, there is no dihedral value that can cope with the height constraint, and improvement



Generation 1 $C_{d_i} = 0.0167$

Generation 3 $C_{d_i} = 0.0180$

Generation 7 $C_{d_i} = 0.0175$

Generation 17 $C_{d_i} = 0.0170$

Generation 30 $C_{d_i} = 0.0167$

Generation 46 $C_{d_i} = 0.0164$

Generation 84 $C_{d_i} = 0.0163$

Generation 117 $C_{d_i} = 0.0163$

Figure 6.5: History of best individual in population for standard genetic algorithm.

stalls with the constraints active (Fig. 6.6). Winglets are quickly discovered, and are retained throughout the optimization history. There is never strong preference for flat elements, but selection pressure does favour dihedral between 0 and 90 degrees. Winglet extensions are never discovered.

When there are redundant copies of each variable, constraint activity can cause expression of different copies. The third copy of the dihedral variables experiences no selection pressure until the height constraint becomes active, so it still has random dihedral values available. It is possible to add more elements and discover more complex designs. Results for this case are shown in Fig. 6.7.

In the original population, the best design has reasonably high span and is relatively flat. Almost immediately, wings of the maximum span are developed, and vertical elements are present at the tips. The main wing is then refined to be closer to horizontal, and the first nearly horizontal winglet extensions are seen. Finally, when the main wing and winglet are close to fully refined, further improvement is achieved by increasing the span of the horizontal winglet extensions.

These general trends are also reflected in the whole population, as shown by the superposition of the right half-wings of the entire population, in Fig. 6.8. In the first generation, which is randomly generated, there are designs with extreme dihedral, but no individual with the maximum possible span. By generation 50, almost all designs lie within the box, and have large span. At generation 100, most main wings are close to flat, there are many vertical winglets and a couple of nearly horizontal winglet extensions. In the final generation, the main wings are even closer to horizontal, and many more "C-wings" are present.

There is significant diversity in the population even after 150 generations. The premature convergence that often limits the performance of standard genetic algorithms is not evident here. This is due to the novel crossover operation, which can produce new designs as offspring even if both parents are identical, by acting at a different location in the genetic string of each parent.

The best designs produced after 150 generations of population size 500 are shown in Fig. 6.9. Rather than concentrating resources in a single long run,

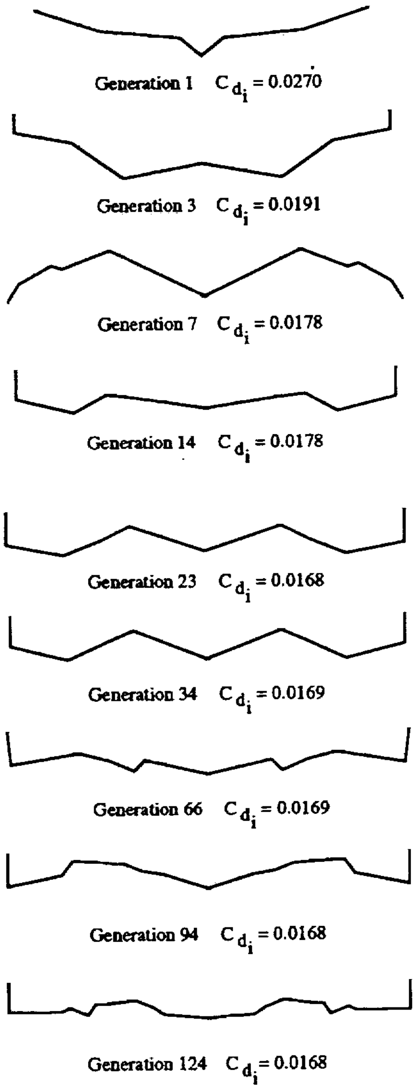

Generation 1 $C_{d_i} = 0.0270$
Generation 3 $C_{d_i} = 0.0191$
Generation 7 $C_{d_i} = 0.0178$
Generation 14 $C_{d_i} = 0.0178$
Generation 23 $C_{d_i} = 0.0168$
Generation 34 $C_{d_i} = 0.0169$
Generation 66 $C_{d_i} = 0.0169$
Generation 94 $C_{d_i} = 0.0168$
Generation 124 $C_{d_i} = 0.0168$

Figure 6.6: History of best individual in population for variable-complexity algorithm with regular encoding.

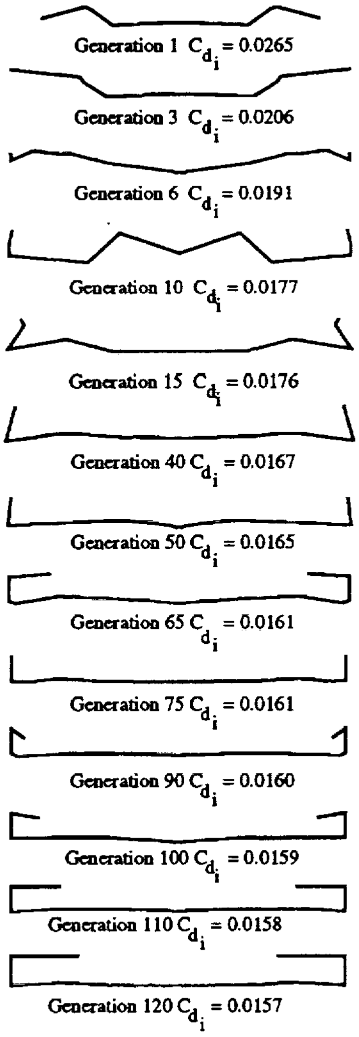
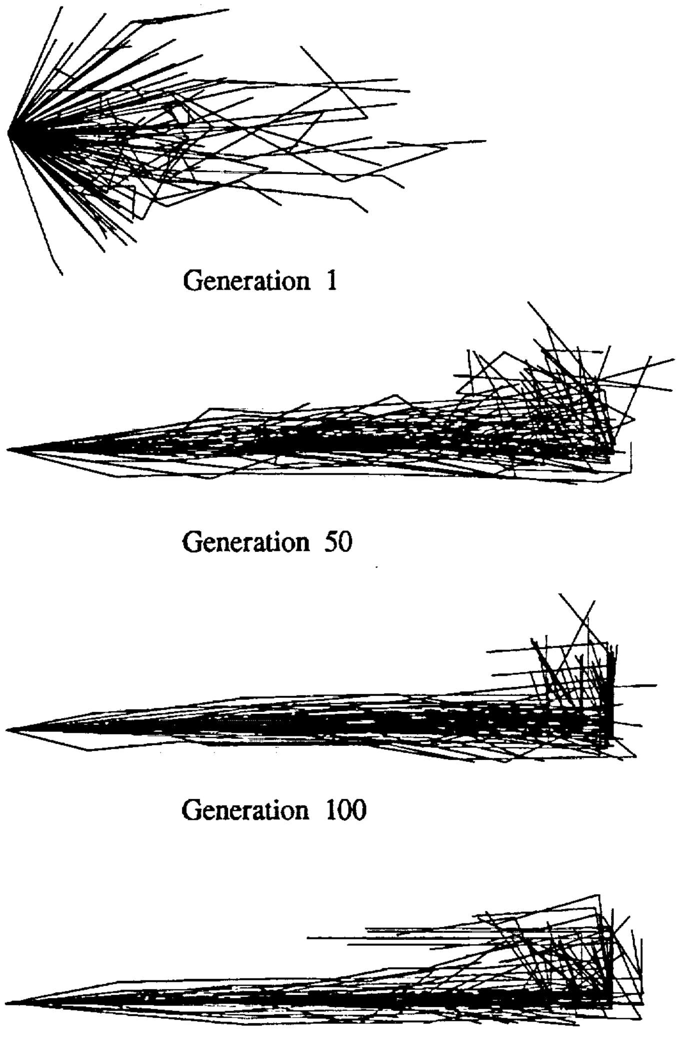

Generation 1 C_{d_i} = 0.0265
Generation 3 C_{d_i} = 0.0206
Generation 6 C_{d_i} = 0.0191
Generation 10 C_{d_i} = 0.0177
Generation 15 C_{d_i} = 0.0176
Generation 40 C_{d_i} = 0.0167
Generation 50 C_{d_i} = 0.0165
Generation 65 C_{d_i} = 0.0161
Generation 75 C_{d_i} = 0.0161
Generation 90 C_{d_i} = 0.0160
Generation 100 C_{d_i} = 0.0159
Generation 110 C_{d_i} = 0.0158
Generation 120 C_{d_i} = 0.0157

Figure 6.7: History of best individual in population for variable-complexity algorithm with extended encoding.



Generation 1

Generation 50

Generation 100

Generation 150

Figure 6.8: Superposition of all right half-wings in population.

computer time is split between several shorter runs. Results for four different randomly-generated starting populations are included. The final geometries are quite different in each case, although performance is quite similar (and always far superior to the performance of the best planar wing). The first two runs produce good designs that differ markedly from the true optimum. These indicate design opportunities that might be preferred for reasons not modelled in the optimization problem, that would be missed by focussing too early on global optimality.

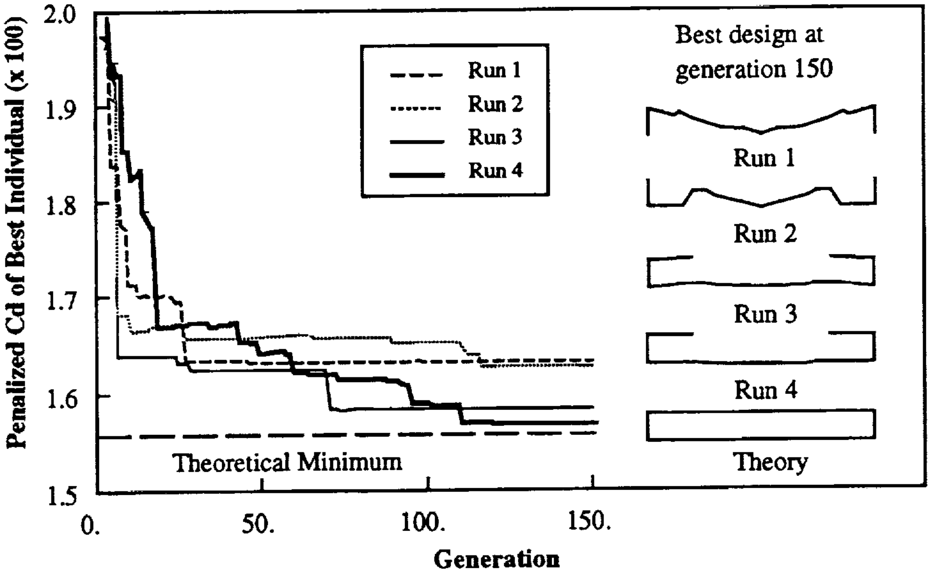


Figure 6.9: Optimization histories for minimum induced drag. $C_{d_p} = 0.000$

In the last two runs, the final result closely approaches the theoretically optimal box shape. Only very slight potential for performance improvement remains, and the selection pressure to further extend the upper wing is very low. Although the exact optimum is not located, the nature of the best solution is clearly identified.

Case	C_{d_0}	C_{d_1}	C_{d_2}
1	0.002	0.0	0.002
2	0.005	0.0	0.005

Table 6.2: Levels of parasite drag.

6.5.2 Minimum Parasite Drag

Consideration of parasite drag complicates the optimization task for non-planar topologies. The best load distribution for induced drag is not optimal for total drag. The total vertical load is constant (fixed total lift) but the best distribution of vertical load depends on the loading of non-planar elements. Load on these non-planar elements can reduce the induced drag of the planar section by redistributing the load in that region, but at the cost of increasing parasite drag locally. Non-planar elements are expected to be smaller and carry less load as parasite drag becomes an increasingly important component of total drag, but a theoretical solution for the optimal topology is not available.

The parasite drag coefficient is assumed to vary quadratically with section lift coefficient, C_l:

$$C_{d_p} = C_{d_0} + C_{d_1}C_l + C_{d_2}C_l^2$$

The relative importance of parasite drag can be modified by varying the coefficients C_{d_0}, C_{d_1} and C_{d_2}. The zero parasite drag case has already been discussed. In this section, two further cases are considered, as indicated in Table 6.2.

The genetic algorithm now uses three variables to describe each wing element, with chord being introduced to reflect the influence of element area on parasite drag. Population size is increased to 800, and number of generations to 250, to handle this larger design space.

The convergence histories and elevations of final designs for Case 1, shown in Fig. 6.10, are similar to those for the case of zero parasite drag. The chord of the winglet extension is very small. For the prescribed total lift coefficient of 1.0, parasite drag is only 20% of the total, but the benefit of the extensions for induced drag is already almost completely offset by their contribution to parasite drag. The combined area of the winglet and extension is 9.6% of the planar area.

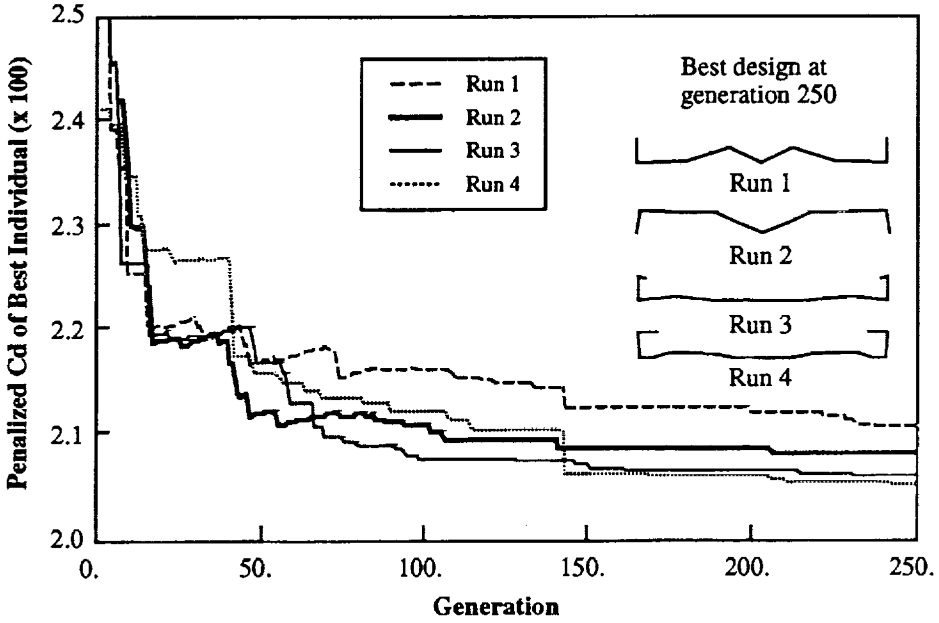


Figure 6.10: Optimization histories. Case 1. $C_{d_p} = 0.004$

Figure 6.11 indicates that a single solution becomes clearly optimal as the relative importance of parasite drag increases. Horizontal winglet extensions are not beneficial, but the winglets are all of maximum height (10% of span). Winglet area is 6.6% of planar area, so nonplanar area is indeed reduced as the importance of parasite drag increases. With parasite drag contributing 40% of total drag, the genetic algorithm gets very close to the correct chord and dihedral distribution.

6.5.3 Summary of Results

These results confirm that the optimal topology is sensitive to the level of parasite drag. The nature of the best solution is similar for all cases, with the wing

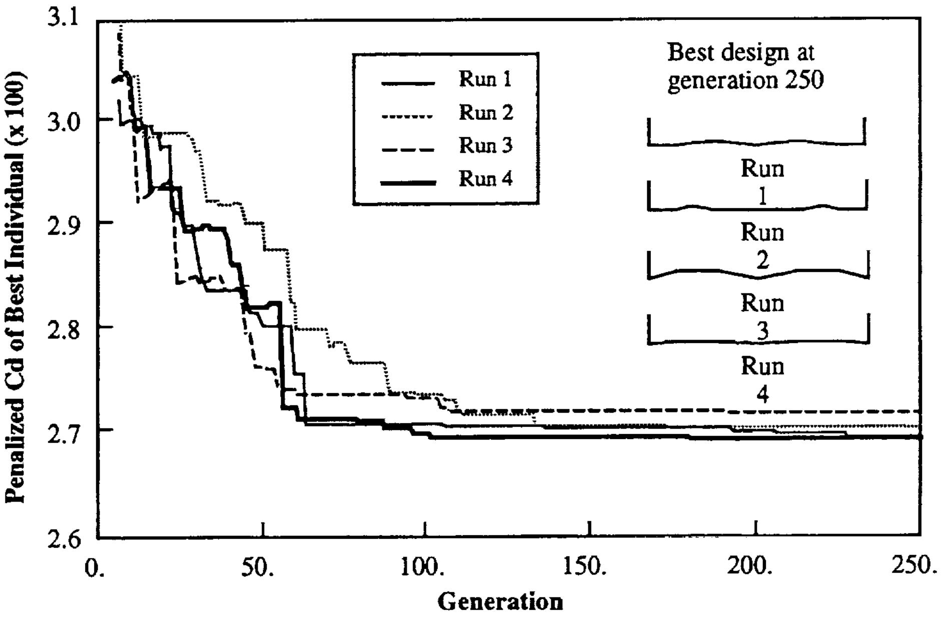


Figure 6.11: Optimization histories. Case 2. $C_{d_p} = 0.010$

being wrapped around the edge of the box, and the arc-length being reduced as parasite drag increases. There is no simple analytic prediction for the best arc-length for an arbitrary level of parasite drag, but the variable-complexity algorithm converges successfully to the correct shape (Fig. 6.12).

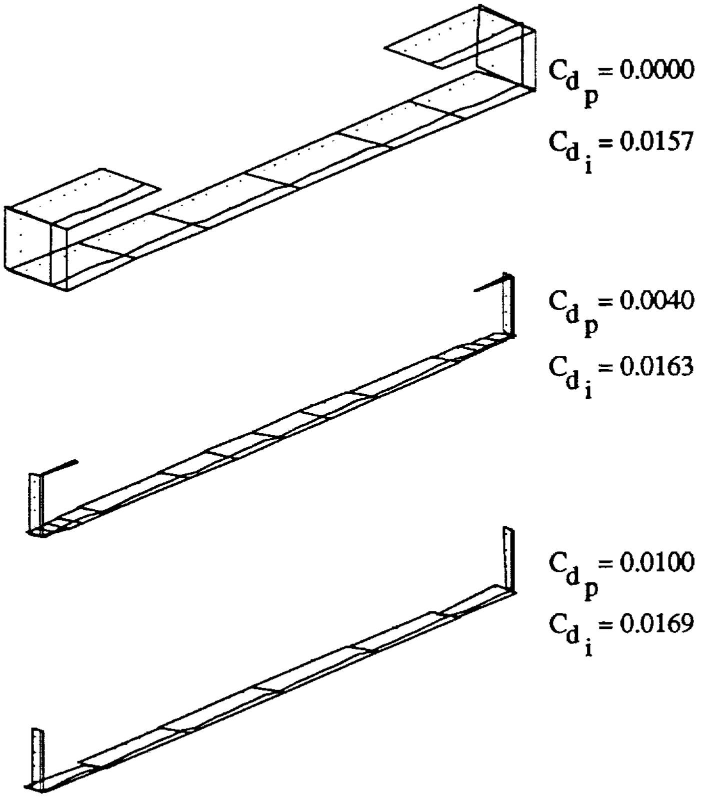


Figure 6.12: Comparison of optimal topologies for different levels of parasite drag.

These results are all for a lift coefficient of 1.0. For realistic cruise lift coefficients, induced drag contributes a smaller fraction of total lift, and winglet area is expected to be further reduced. Inclusion of structural weight constraints will also influence the optimal design. Such considerations will be incorporated in future investigations of complete aircraft configurations.

6.6 Comparison with Calculus-Based Optimization

The integration of panel methods with calculus-based optimizers is problematic if panel width is allowed to vary. This difficulty was a motivating factor driving the decision to use genetic algorithms in this domain. Even when panel width is constant, however, calculus-based optimizers can have trouble. Simultaneous consideration of the conflicting requirements for minimum drag and acceptable geometry causes a wing of fixed arc length to fold into non-optimal shapes that are strongly influenced by the initial arrangement of lifting surfaces. Figure 6.13 shows the results of three optimization runs from different starting points. In the second run, the wing folds into a geometry that cannot be accurately analyzed by the vortex-lattice method, and the optimizer cannot move from this illegal design. Clearly, the calculus-based method produces unacceptable results when used alone.

Calculus-based methods can be usefully combined with genetic methods to improve the quality of the search. In the truss optimization studies presented in the Chapter 5, a hybrid method was constructed, with the smooth subspace being optimized for each candidate topology generated by the genetic method. Such a scheme is not appropriate here, because calculus-based optimization would increase the cost of each function evaluation by three orders of magnitude. Instead, the final design produced by the genetic algorithm is refined by calculus-based optimization of dihedral and chord. Figure 6.14 shows the influence of this refinement on the design for the low level of parasite drag. It yields a 1.5% improvement in total drag. When the parasite drag level is increased, final refinement produces less than 0.25% improvement in total drag.

The efficiency of calculus-based optimization in the neighborhood of the optimum confirms that the exact solution should not be sought by the genetic method. Several runs from different starting populations can identify several near-optimal candidate topologies. The calculus-based method allows accurate

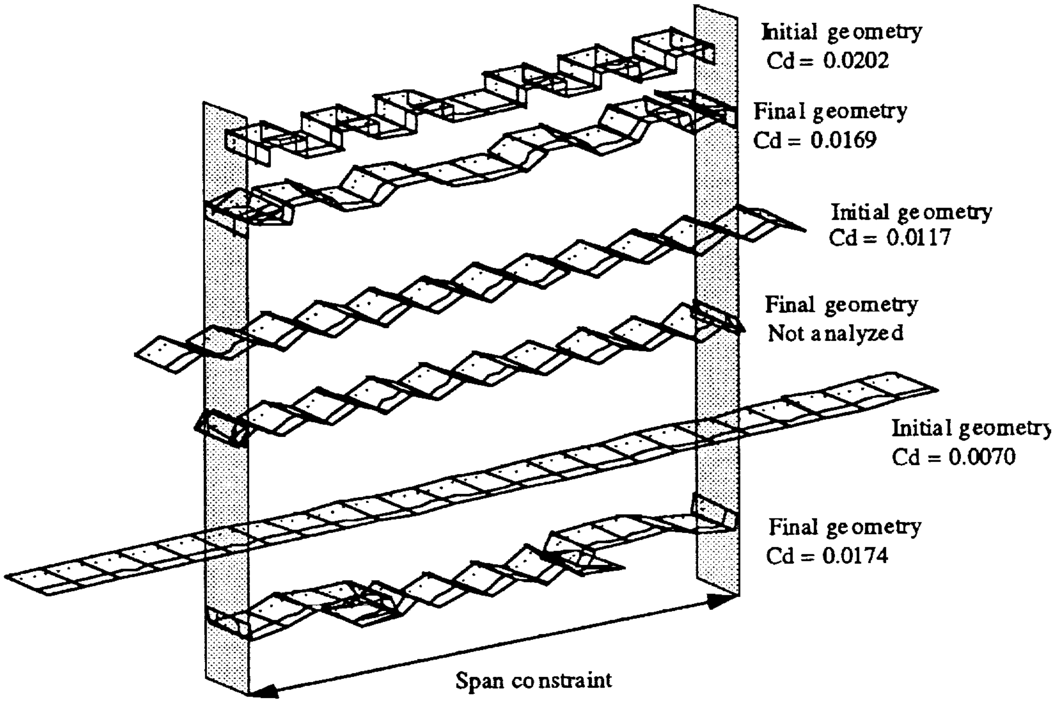


Figure 6.13: Calculus-based optimizer is trapped at local minima.

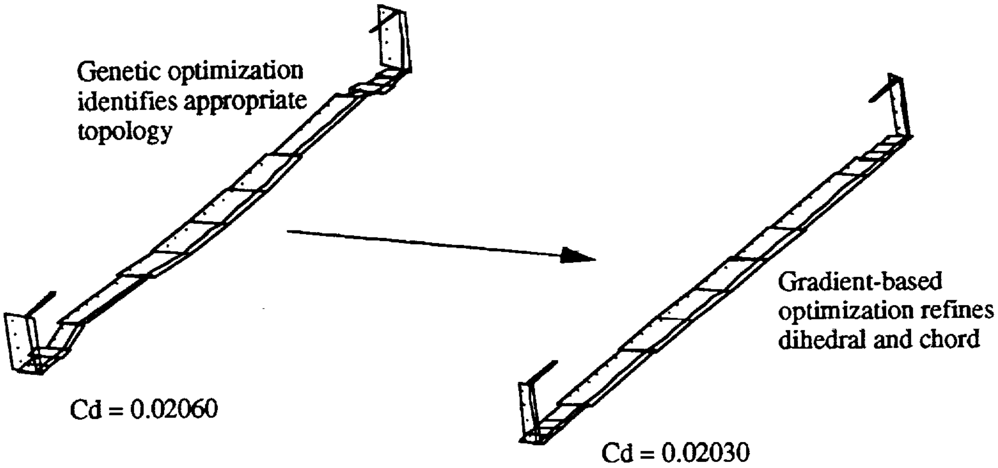

Genetic optimization identifies appropriate topology
Cd = 0.02060
Gradient-based optimization refines dihedral and chord
Cd = 0.02030

Figure 6.14: Final refinement by gradient-based optimizer.

comparison of competing concepts. Once again, different optimization methods used in combination are more effective than either single method.

6.7 Summary

The application of genetic algorithms to topological design of wings has again demonstrated importance of constraint-handling for these methods. The form of constraints strongly influences algorithm efficiency. Repair methods generally reduce the size of the search space, so that small populations provide an accurate sample of useful building blocks. Penalty methods can be applied more generally, but the penalty weight should be chosen so that performance declines gradually as the margin of constraint violation increases.

Attempts to optimize non-planar topologies using fixed-complexity representations of the wing were unsuccessful. In all cases, the wing was folded to satisfy the geometric constraints, but the nature of the folding was strongly dependent on initial conditions, and suboptimal topologies were always produced. Fixed-complexity wings all use the maximum number of design variables, so the cost of each function evaluation is relatively expensive. Calculus-based optimization can make a limited contribution in this domain, by efficiently refining the detailed geometry of a specified topology.

The variable-complexity genetic algorithm is better able to separate the influences of the objective and the geometric constraints. An extended encoding, with several entries for each variable, helps to achieve this separation. Expression of different sections of the string is controlled by environmental factors. Selection works on one part of the string when constraints are inactive, and at different points when constraints are activated. Distinct strategies are evolved to cope with changing environmental circumstances. This extended encoding scheme is an important development for evolutionary optimization in constraint-bound domains.

Chapter 7

Genetic Optimization in the Quasi-Procedural Environment

The last four chapters have shown that genetic algorithms are effective search tools, both in standard fixed-complexity form, and with variable-complexity representations permitted. Their usefulness is increased when they are applied in conjunction with calculus-based optimizers. In this chapter, the integration of a genetic algorithm into the general design system is described.

The interaction between the quasi-procedural executive and genetic algorithms provides two chief benefits. Re-ordering the population so that similar individuals are evaluated consecutively minimizes the cost of assessing each generation during optimization. When the variable-complexity algorithm increases the detail of the parameterization during optimization, the quasi-procedural system can automatically adjust the computational path to include more sophisticated analyses that make use of the extra parameters. These benefits are discussed in this chapter.

7.1 Efficient Evaluation of the Population

In Chapter 2, it was shown that the quasi-procedural method can reduce optimization time when performance updates are calculated after only a few variables have been modified. This is a common situation in genetic optimization, because most variables do not change value under the action of the genetic operators. Offspring share at least half of their variables with a parent. When the common features are recognized, only a subset of the performance parameters need be updated.

Some members are exact clones of a parent, when they are reproduced without crossover or mutation. Evaluation of these members is not required. The performance of the parent is inherited at the time of reproduction.

When mutation occurs without crossover, most variables are unchanged. Even when crossover occurs, at least half the design variables are inherited directly from one parent. If all information about the parent is retained, and can be loaded as the current design, the update is relatively cheap. The standard system, however, retains detailed information only about the last candidate design that was evaluated. The memory required to store the complete database for each member of the population is prohibitive for complex problems. A parallel version of the genetic algorithm, with the population distributed over many machines, might alleviate this difficulty, but such an implementation is beyond the scope of this work.

Apart from the similarity between parents and offspring, the genetic algorithm produces similarity between members of the same population. The highest fitness members of one generation can participate in the production of several offspring, and these offspring are expected to share some characteristics. The population can be arranged so that similar individuals are evaluated consecutively. This reduces the number of variables being updated, and consequently the computational expense of the updating.

7.1.1 Measuring Difference Between Population Members

Before the population can be ordered according to difference, a scheme must be developed to measure that quantity. Efficiency is influenced by the number of invalidations that occur when the values in the database are changed. The extent of change for any variable is unimportant, because all changes for that variable cause the same invalidations. Different variables cause different numbers of invalidations, so the difference measure should take account of which variables are changed, rather than simply counting the total number of changes. For many tightly-coupled aerospace systems, some design variables can influence all the analyses. When two population members have different values for one such variable, they are completely different, because consecutive evaluation would result in maximum computation for the second design.

The variables should be arranged according to the number of modules that are invalidated by them. With the assumption that there is an order that produces an increasing number of invalidations[1](i.e. that the set of invalidations caused by the variable in position m is a subset of the invalidations caused by the variable in position $m + 1$), the difference between two designs can be measured by noting the last variable that has a different value in each of them. The order of difference between two population members is m, where m is the position of the last different-valued variable in the ordered set of design variables.

When the extent of the difference in each variable is also noted, a comparison of the entire population with the current best individual can be used to infer the difference between any two individuals in the population. Individuals with the same order of difference to the current best need to be ordered relative to each other. The extent of their difference in the variable at position m is used to do this, because members with the same value at m should be evaluated consecutively. The difference metric should therefore account for the extent of

[1] This assumption was discussed in Section 2.3.2, on ordering variables for gradient computation. It is not always exactly true, but for tightly-coupled problems the number of unnecessary computations that result from its violation is generally small.

all differences between the current best design and each design in the population. The metric used here to compare the $Best$ member with the ith member is:

$$DiffVal(Best, i) = \sum_{m=1}^{N} diff(m)$$

where

$$diff(m) = \begin{cases} 10^{m-1}(0.55 + 0.45(\frac{var(i,m)-var(Best,m)}{range(m)})) & \text{if } var(i,m) \neq var(Best,m) \\ 0 & \text{otherwise} \end{cases}$$

N is the number of design variables and $range(m)$ represents the allowable range of values for the variable at position m.

Once the difference values have been assigned, the population members must be sorted for evaluation in the correct sequence. They are arranged according to increasing value of $DiffVal$. The Shell sorting algorithm [103] is used. The operations required to sort a population of size M is on the order of $1.5Mlog_2M$.

7.1.2 Performance of the Ordering Scheme

A simple aircraft synthesis task was used, in Section 2.3.2 of this thesis, to evaluate the influence of ordering on gradient calculation. A similar task is appropriate for the current study, except that *Number of Engines*, *Number of Wing-Mounted Engines* and *SeatsAbreast* are added as extra design variables. With these discrete-valued variables included, search should be conducted by the genetic optimizer rather than the calculus-based method used in Chapter 2.

Figure 7.1 shows the best arrangement of design variables for this task. Each column of the grid in that figure is associated with a design variable, and each row is associated with an analysis subroutine. A shaded box in the grid indicates that the analysis routine of that row needs to be executed when estimating the gradient for that column.

The figure shows that only 4 design variables (*Altitude.FinalCr*, *MachNumber.Landing*, *MachNumber.TO*, *FlapDeflection.TO*) produce significantly fewer subroutine invalidations than the other variables. Unless consecutive population members differ from each other in only these four variables, the quasi-procedural

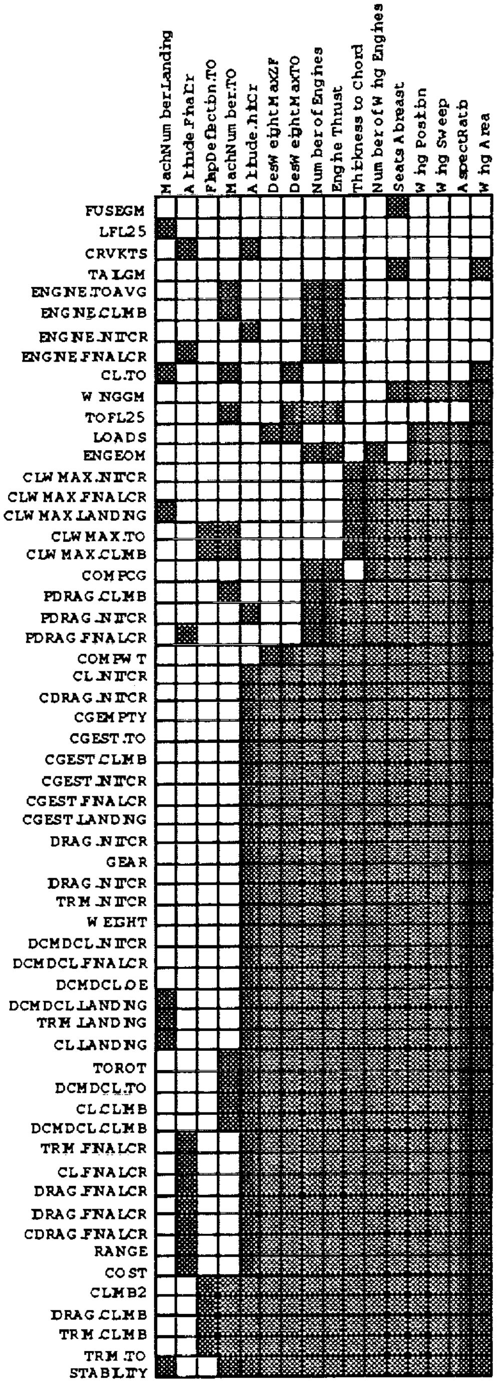


Figure 7.1: Best arrangement of variables to minimize computation required to evaluate the population.

method is unable to greatly reduce the computation required to evaluate the population. Indeed, comparison of computation for unsorted populations with computation for sorted populations confirms that the quasi-procedural method only reduces the work by 10% when sorting is exploited. While this influence is somewhat problem-dependent, sorting is unlikely to have a dramatic impact on computational requirements for the genetic algorithm.

7.2 Computational Path Generation

The applications of the variable-complexity genetic algorithm presented thus far have each used a single type of building. Designs are composed of similar elements, and complexity changes as the number of elements changes. The algorithm can also be used for designs composed of several types of building block.

The complexity of description for aircraft synthesis studies increases in this fashion. A wing may be initially described by gross parameters such as *Wing Area*, *Wing Sweep* and *Thickness-to-chord*, but later in the design process flaps, slats and chord extensions are likely to be considered, and more detail will be added for the basic wing, by prescribing twist and thickness distributions. Analyses that use gross parameters to estimate performance will not be appropriate when local properties are supplied, so more complex analyses must be used.

The quasi-procedural executive generates an efficient computational path automatically, while most database systems require a user-specified procedure. This ability is particularly helpful when the computational path changes during optimization, because most systems would require user intervention at that point. The freedom to supply more than one analysis to compute the same output variable is a new feature of the updated quasi-procedural system (described briefly in Chapter 2). It can be used, in conjunction with the variable complexity genetic algorithm, to switch analyses depending on the inclusion or omission of detailed variables in each candidate design in the population.

A simple example of this process is outlined here. *WingPosition* is a variable that refers to the location of the wing on the fuselage (measured as a fraction of fuselage length from the nose of the aircraft). It is included in the optimization task to allow satisfaction of constraints on trim and stability. Human designers often avoid such considerations at the conceptual design stage, deferring the choice of wing location until the preliminary design stage. Figure 7.2 illustrates how a variable-complexity design task can use *WingPosition* as an optional variable.

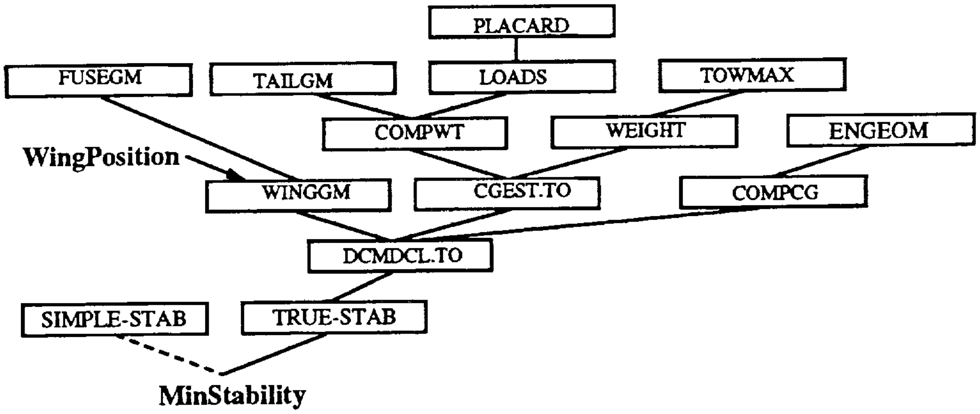


Figure 7.2: Alternative computation paths for *MinStability* depend on specification of *WingPosition*.

The boxes in this figure represent analysis subroutines, and the lines connecting the boxes represent information transfer between routines. The figure shows two alternative computational paths for updating the value of the *MinStability* constraint. When *WingPosition* is included in the genetic string, the TRUE-STAB subroutine is used, and complex analyses produce an accurate evaluation of stability. If *WingPosition* is omitted, the SIMPLE-STAB subroutine produces a default value. Note that this is not the same as providing a default value for the input variable, *WingPosition*, because such a value would run the complex analyses, but still only produce an estimate for stability (based on the default value of *WingPosition*). The significance of avoiding the complex analyses can be appreciated by noting that 19 of the 58 subroutines listed in Fig. 7.1 do not need to be executed when accurate assessments of trim and stability are not required.

A detailed development of this methodology requires a careful investigation of the role of each design variable in the optimization task. Design variables which exert strong influence on particular constraints without greatly affecting the objective can be omitted from the variable-complexity encoding. New analyses, needed to estimate the constraints when the appropriate design variables are unavailable, must be provided. Such work lies beyond the scope of this thesis, but it should be conducted when more accurate and flexible analyses are used for aircraft synthesis studies. An appropriate task is described in the Future Work section of the next chapter.

Chapter 8

Conclusions and Suggestions for Future Work

This thesis presents several new developments aimed at expanding the role of formal optimization in the aerospace conceptual design process. The complication of accurately analyzing tightly coupled aerospace systems is generally addressed by parameterizing the entire design, and iteratively adjusting the input parameters. Although this approach has the character of an optimization procedure, three aspects of existing design systems limit the range of applications handled by available optimization algorithms. Flexibility needs to be increased in the architecture that integrates analyses and optimizers, in the optimization algorithms themselves, and in the parameterizations used to formulate specific optimization tasks. This chapter summarizes contributions that have been made in each of these areas, and provides suggestions for future work on all of them.

8.1 Conclusions

8.1.1 Integration of Analyses and Optimizers

The quasi-procedural architecture uses a loose-coupling approach to integrate analysis modules through a central database. It automatically generates a computational path, whereas most database managers require a user-specified procedure. Consequently, it was chosen to control the design system developed here. The basic architecture was extended and enhanced, so that it could handle more general data types and more complex analyses. It was combined with NPSOL, a high quality calculus-based optimization algorithm, to produce a baseline design system. Apart from the practical benefit of providing a useful optimization tool for industry, this system allows comparative assessment of new capabilities introduced here.

The efficiency of the quasi-procedural architecture for large scale tasks was confirmed by using it to integrate complex analyses for aircraft synthesis studies. The flexibility and extensibility provided by this architecture became more valuable as the system size increased. A procedure was outlined for automating the modification of existing source code to communicate through the database. The influence of analysis structure on optimizer efficiency was also investigated. Replacement of iteration loops with extra design variables and constraints reduced analysis effort. Optimization algorithms were modified to improve performance through exploitation of the quasi-procedural executive.

8.1.2 Optimization Algorithms

The statistical mechanism of genetic optimization has been critically evaluated, and key requirements for successful location of the optimum were identified. There must be a correlation between pieces of the genetic string and performance of the design, so that superior designs can be constructed by recombining building blocks from different ancestors. The population must be sufficiently large for promising features to be recognized and retained. The appropriate size depends on the topology of the design space, which is affected by encoding and constraint-handling. Repair methods satisfy constraints efficiently, but they must be developed as part of the analysis system for each domain. Penalty methods are more generally applicable, but weights should be chosen carefully so that penalties for violation are increased gradually. The influence of these implementation issues was demonstrated by application to interplanetary trajectory design. The genetic algorithm provided more convenient and more efficient location of the optimum than the grid search technique commonly used in this domain.

A genetic algorithm should be used to identify near-optimal rather than fully optimal designs, because gradient methods are generally more efficient in the neighborhood of the best concept. The flexible design environment allows convenient switching between optimization alternatives, as the appropriate method changes during design development. The population of candidate designs can be used to identify several local optima simultaneously, by introducing speciation operators. The designer's choice between near-optimal concepts may be driven by considerations that were not modelled in the formal optimization task.

The creation of a variable-complexity algorithm, which works with genetic encodings of variable length, was driven by an understanding of the importance of building blocks and encoding. Large numbers of variables increase the number of building blocks in the string, so correlation between individual building blocks and performance is difficult. If simple encodings are used initially, useful buidling blocks are more readily identified, and can be re-used in more complex descriptions as optimization proceeds. This process is discussed further in the next section.

8.1.3 Flexible Parameterization

A variable-complexity genetic algorithm was created to allow parameterization to change during optimization. It is able to start with a simple description that captures the gross features of a design, and add detail as the design matures. This broadens the scope of formal optimization for automated conceptual design.

The blockstacking problem showed that fixed-complexity encodings are deceptive when a poor value for a single variable causes catastrophic performance for an entire design. Variable-complexity encodings that initially use fewer variables are less likely to include such a poor value, so promising building blocks are more readily identified. In general, simple descriptions make it easier to correlate performance with each building block in the encoding.

The process of decoding the variable-complexity string is similar to embryonic development. Later sections of the string modify the structure produced by earlier decoding. This process allows environmental factors to influence development, and also allows limited performance assessment to be conducted while construction is in progress. Discovery of optimal wing topologies relied on development being guided by constraint activity. The genetic string was extended to include alternative encodings for each variable. Different copies were expressed when constraint activity changed. Appropriate values for different circumstances were selected independently. The design is essentially constructed as it is decoded, so the extended genetic string should prove useful for handling manufacturing considerations that are often omitted from aerospace optimization tasks.

Application of the new algorithm to topological design of trusses demonstrated that optimizer performance improved when domain-specific knowledge was incorporated in the variable-length encoding. Candidate structures were forced to include attachments to the fixed nodes and to the loaded node. Mechanisms and unloaded structures were removed from consideration. In general, variable-length encodings that describe only the elements that appear in the design are more natural and more efficient than encodings that refer to all possible elements.

8.2 Suggestions for Future Work

The study of the relationship between optimizer and analyses suggests several possibilities for improvement. Automatic differentiation could increase optimizer efficiency by avoiding the cost of finite-differencing to estimate gradients. The method used to remove iteration loops, which shifts the task of matching the values of two variables from the analyses to the optimizer, can be extended to isolate the set of analyses into independent groups. An optimizer would then be responsible for co-ordinating the execution of each group, and checking that they use consistent values for all parameters. Modification of the quasi-procedural executive, to permit parallel execution of subtasks, would further increase the flexibility and efficiency of the design package.

Genetic optimization requires a large number of function evaluations to identify promising building blocks in candidate designs. Future development should include parallelization of the algorithm, to reduce the time required for convergence. This should be achieved quite simply, because evaluation of different population members is already conducted independently. Genetic algorithms could be improved by the introduction of more advanced reproduction operators, such as duplication and deletion. The speciation function used with the standard genetic algorithm should be modified for the variable-complexity method.

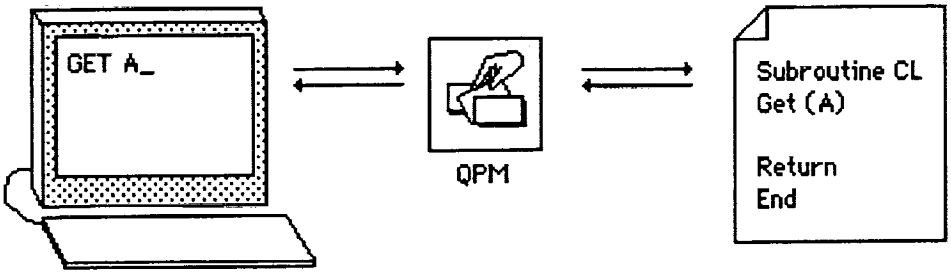
The potential benefits of flexible parameterization for general optimization have not been fully explored here. A more general investigation of shape grammars for topological design is warranted. The major applications of the variable-complexity algorithm in this thesis used a single type of building block, and varied the number of blocks during optimization. Further study of tasks which involve several types of variables should be conducted. A full aircraft synthesis task that includes structural and aerodynamic variables, with manufacturing and structural weight constraints as well as performance requirements, would be a suitable candidate for this investigation. It would also permit further study of the importance of quasi-procedural path generation for variable-complexity optimization.

Appendix A

Technical Details of Quasi-Procedural Architecture

A.1 The Quasi-Procedural Method

The quasi-procedural method is a form of event-driven program. The method may be triggered by the user who requests the value of a variable, or by a subroutine that can also request the value of a variable needed for further computations (Fig. A.1).



```
GET A_
```

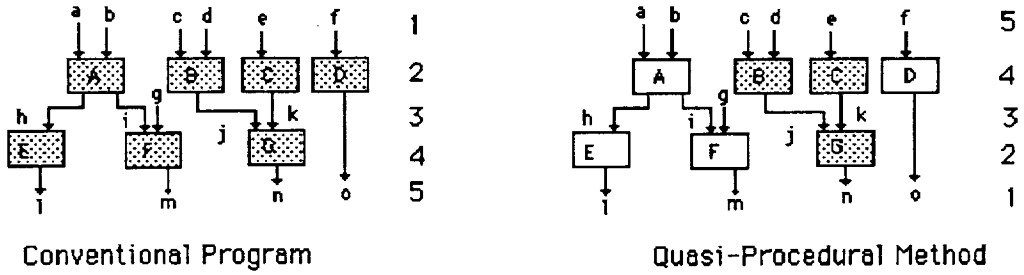
```
Subroutine CL
Get (A)

Return
End
```

Figure A.1: Quasi-procedural method may be triggered by user or subroutine.

The quasi-procedural method differs from conventional programming architectures in that the program is not strictly procedural. While a conventional program proceeds from all of the inputs to all of the outputs, a quasi-procedural program invokes only the subroutines required to produce a valid value of the requested variable. The program is flexible, because it builds a new procedure to find each requested variable. It is efficient because the procedure that it builds always performs the minimum computation required to update the requested variable.

In Fig. A.2, boxes represent analysis routines and lower case letters represent variables. If we need to compute the value of *n*, for instance, a conventional program would require that we specify *a-f* and run all of the routines A-G. In the quasi-procedural method, the system recognizes that only routines B, C, and G need to be run and only the inputs *c*, *d*, and *e* are necessary. It does this by actually calling the subroutines from the bottom, up. Execution of each routine is suspended while the path is constructed, and is resumed procedurally after the path is complete.



Conventional Program

Quasi-Procedural Method

Figure A.2: Quasi-procedural method executes only the necessary subroutines.

The system starts with the variable *n*. It checks the database to get information about *n*. If *n* is valid, the system takes its current value from the database. If it is invalid (as it is in this example), the system takes the name of the routine which will deliver *n* as output, and runs it. Here, that routine is G.

When routine G is called, it needs values for its input variables, j and k. To get j, the system runs routine B, and to get k, it runs routine C. Finally, execution of routine G can be resumed, and the updated value of n is produced.

All of this computational path generation is handled by the subroutine GET, which is included in the quasi-procedural executive. A flow chart for computational path construction by this subroutine is shown in Fig. A.3. Figure A.4 indicates how the GET subroutine is called recursively during path construction.

With the appropriate computational procedure generated automatically, developers of the analyses need only be concerned with writing the relevant procedural portion of the system: the individual computational routines. These should be written in standard Fortran, with calls to subroutines in the Genie library included in an appropriate way.

A.2 Consistency Maintenance

The above description of quasi-procedural execution mentioned that the validity of each variable is checked when it is found in the database. An invalid variable is inconsistent with the current set of specified inputs. If a computation were run to update the value of that variable, its value would change. The variable would also become valid, because the value produced by the calculation would be consistent with the input set. This makes it clear why a computation is performed only if the variable is invalid. A computation for a valid value does not change its value, so it is a redundant calculation. The tracking of variable validities is called consistency maintenance.

When the system is initialised, all the computed results are invalid. A result variable becomes valid by being computed. It will only return to being invalid if any specified input on which it depends is changed. The simplest consistency maintenance scheme would label all results invalid whenever any specified input value was modified. However, this conservative scheme would be inefficient, producing many redundant computations. Consider a simple example. Fuselage diameter depends on seat width, number of seats abreast, and aisle width, but

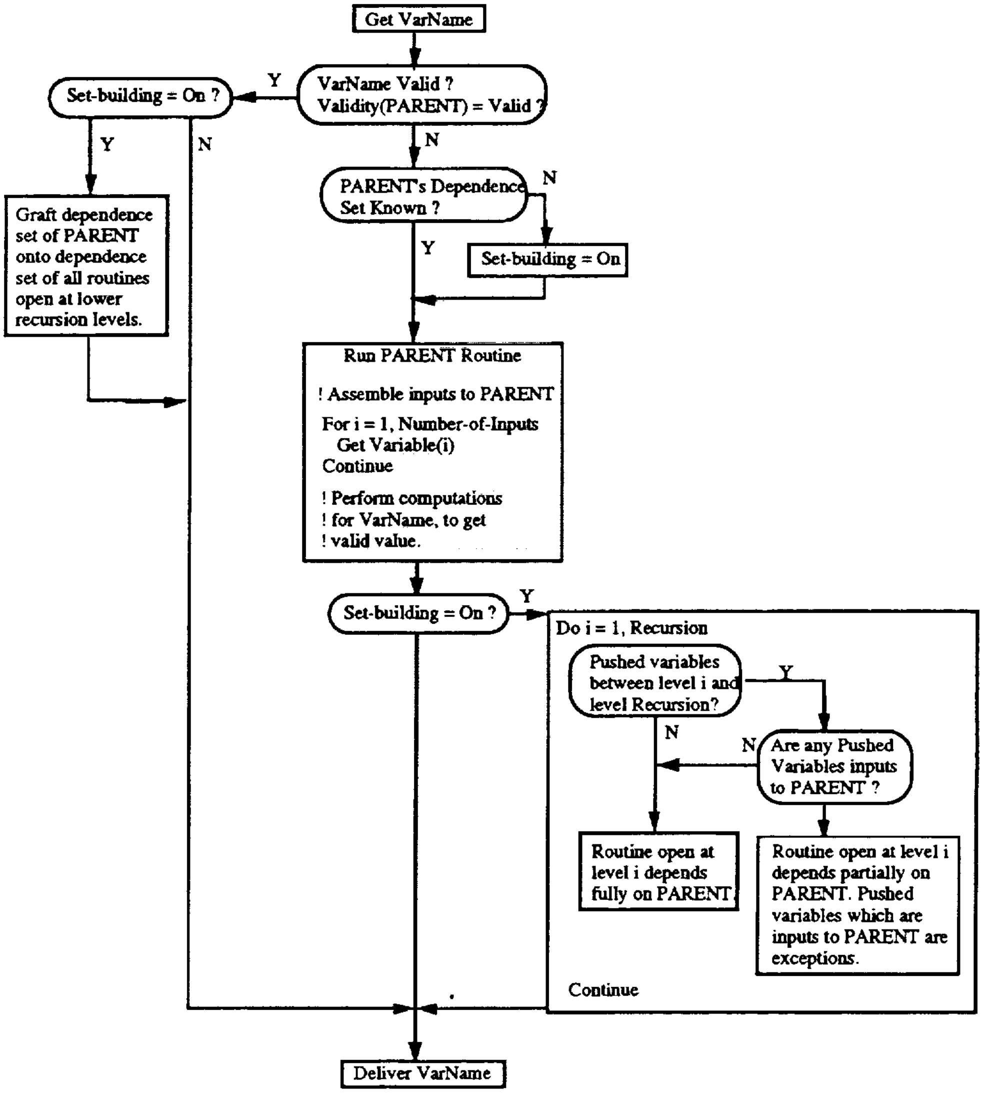
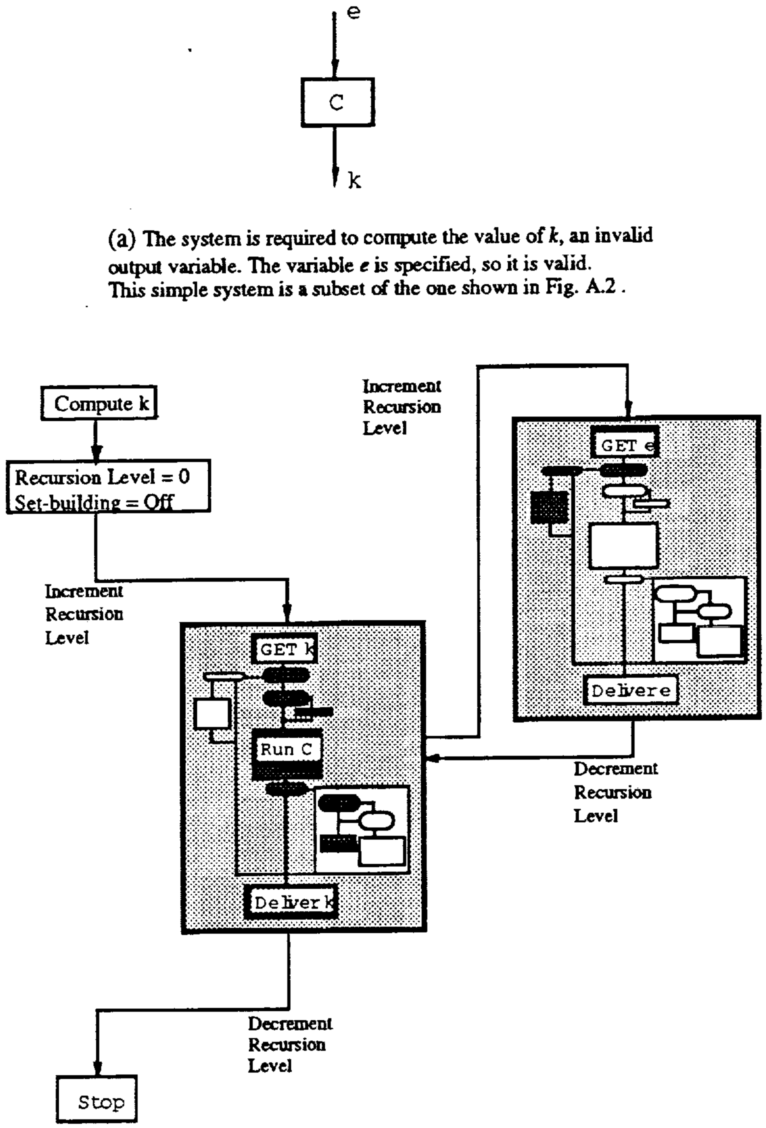

Get VarName
VarName Valid ?
Validity(PARENT) = Valid ?
Y
N
Set-building = On ?
Y
N
Graft dependence set of PARENT onto dependence set of all routines open at lower recursion levels.
PARENT's Dependence Set Known ?
N
Y
Set-building = On
Run PARENT Routine
! Assemble inputs to PARENT
For i = 1, Number-of-Inputs
Get Variable(i)
Continue
! Perform computations
! for VarName, to get
! valid value.
Set-building = On ?
Y
Do i = 1, Recursion
Pushed variables between level i and level Recursion?
Y
N
N
Are any Pushed Variables inputs to PARENT ?
Routine open at level i depends fully on PARENT
Routine open at level i depends partially on PARENT. Pushed variables which are inputs to PARENT are exceptions.
Continue
Deliver VarName

Figure A.3: Flow diagram for the GET subroutine.


e
C
k
Compute k
Recursion Level = 0
Set-building = Off
Increment Recursion Level
GET k
Run C
Deliver k
Increment Recursion Level
GET e
Deliver e
Decrement Recursion Level
Decrement Recursion Level
Stop

(a) The system is required to compute the value of *k*, an invalid output variable. The variable *e* is specified, so it is valid. This simple system is a subset of the one shown in Fig. A.2 .

(b) Computation of *k*.
Each lightly shaded box represents the GET routine, which is described in detail in Fig. A.3. The sections of the GET routine which are actually run are marked with the darker shading. GET is first called to deliver *k*. *k* is invalid, so Subroutine C is launched. As Subroutine C is run, it issues a call to GET *e*. Execution of subroutine C is suspended, until the valid value of *e* is delivered. The consistency maintenance scheme records the dependence of *k* on *e*.

Figure A.4: Recursive calls to the GET subroutine during path construction.

it does not depend on wing area. When the wing area is modified by the user or the optimizer, the simple consistency maintenance scheme will label the fuselage diameter as invalid. But if fuselage diameter is recomputed, the value will not change (because it is not dependent on the modified input variable, wing area), so it should really be labelled as valid. An efficient consistency maintenance scheme must invalidate only the computable variables which are dependent on the modified input variable.

Dependency information can be derived during the calculation of computed results, because the computed result depends on the inputs to all the subroutines called during the course of the computation. Consider again the example used in the explanation of the quasi-procedural method (Fig. A.5). The diagram of computational paths is similar to a family tree, and the analogy with genealogy is a strong one. The dependency relationship 'n depends on j' is the same as 'n is a descendant of j', or 'j is an ancestor of n'.

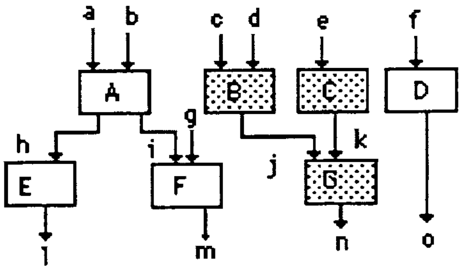


Figure A.5: Consistency information is developed for the shaded subroutines and associated variables.

Initially, all computed results are invalid. When the user asks for n to be computed, the system finds that it is invalid, and so it calls routine G, which produces n as output. As G runs, it asks for j. j is an input to a subroutine called during the computation of n, so n depends on j. The system finds that j is invalid, so it calls for routine B to run. This calls for c and d, and shows that n depends on c and d, too. Note, though, that c and d have been called during the computation of j, so it is also true that j depends on c and d. Now that the computation of j is complete, G continues to run, and asks for k. Hence, n depends on k. Routine C is called to produce k, so k depends on e, and n depends on e. Finally, routine G is resumed once more, and runs to completion. The consistency information is summarised in Table A.1. All other variables remain invalid, and their dependency sets are still unknown.

Variable	Dependency Set	Validity
n	$\{j,c,d,k,e\}$	Valid
j	$\{c,d\}$	Valid
k	$\{e\}$	Valid

Table A.1: Summary of consistency information.

Now consider the case where k is computed first. Routine C is called to produce k, which shows that k depends on e. When n is computed, routines G and B are run as before, but when the system asks for k it finds that it is valid, and there is no need to run routine C. The system still has to find out that n depends on e. This is done by grafting the dependency set for the valid variable onto the dependency set for the variable currently being computed. In the example, the dependency set for k, which is $\{e\}$, is grafted to the partial set for n, which is $\{j,c,d,k\}$ to produce the correct set for n, $\{j,c,d,k,e\}$.

The determination of dependence relationships is made more complicated by the local specification of variables through the use of the PUSH command. PUSH fixes the value of a variable while a certain calculation is performed, but when the calculation is finished, the fixed value is removed from the database. A good example of the use of local specification is provided by the calculation of drag. Drag depends on local flight conditions (density and airspeed), so its value changes throughout a flight, and density and airspeed must be specified at the correct local conditions whenever drag is calculated. When drag is calculated quasi-procedurally, density and airspeed are PUSHed by the routine which requests drag. The problem is that variables which depend on drag do not depend on density and airspeed , because they are associated with a specified density and a specified airspeed. Consider the case of range and second segment climb, shown in Fig. A.6.

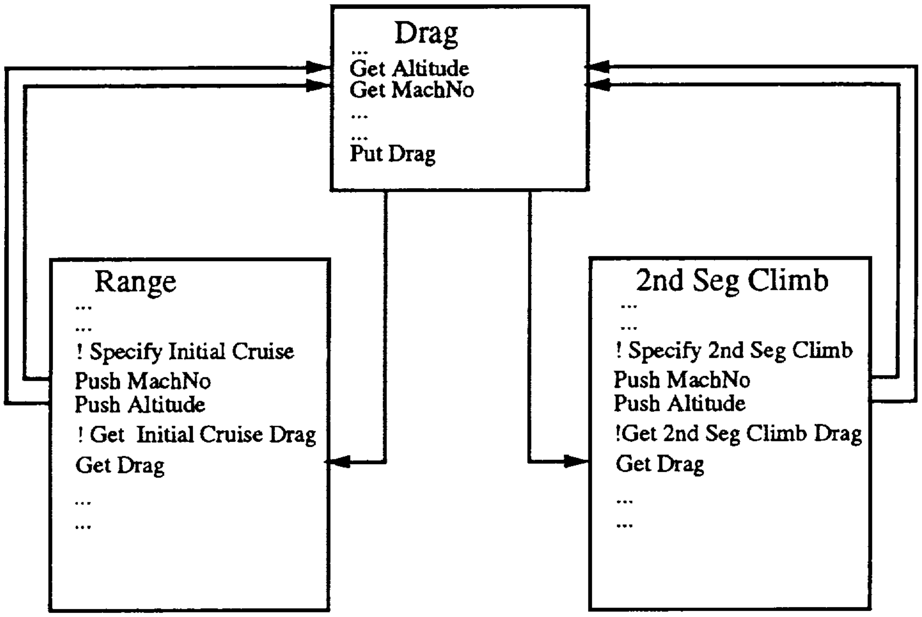


Figure A.6: Partial dependence in quasi-procedural consistency maintenance.

Normally, when range is being computed, the dependence algorithm would add *Altitude* (associated with density) and *MachNo* (associated with airspeed) to the dependence set for *Range*. Similarly, *Altitude* and *MachNo* would be included in the dependence set for *SecondSegmentClimb*. This would mean that *Range* would be invalidated whenever *SecondSegmentClimb* is computed, and vice versa. This problem is avoided by adding a requested variable to the dependence set for the variable being computed only if it is not currently PUSHed.

When the dependence sets are known for the computed variables, it is a simple matter to find invalidities. Whenever a specified variable is changed, all the dependence sets are searched. Any computed variable which has the modified variable as an element of its dependence set is labelled as invalid, because its value is inconsistent with the newly specified value.

In practice, it is more efficient to make dependence sets between analysis routines rather than between variables. Analysis routines typically have several inputs, and these are grouped as one element in the dependence set for the routine, so the searching is reduced. This slightly changes the consistency maintenance scheme. Now, the validity of a variable is ascertained by checking the validity of the routine which calculates it. Invalidities are propagated by first invalidating any routine which has the modified input as an input, and then invalidating any routine which has one of these routines in its dependence set. Finally, there may be partial dependence between routines, when some inputs are pushed and others are not. Consider the case of *Range* and *Drag*. It has been shown above that *Range* does not depend on some inputs to the DRAG routine: *Altitude* and *MachNo*. However, *Drag* also depends on many other inputs, such as *WingArea*. A modification to *WingArea* should invalidate *Range*. Where there is partial dependence, the subroutine is included in the dependence set with the variables that are exceptions appended. Thus, DRAG is included in the dependence set for RANGE, but *Altitude* and *MachNo* are noted as exceptions.

Dependence Set for *Range*
DRAG (Partial) *Altitude MachNo*
RANGE
etc.

When dependence relationships have been established, they can be used to propagate invalidities arising from modification of input variables. A flow diagram for this updating of consistency information is presented in Fig. A.7. The first step in the update scheme is to invalidate each subroutine that has the modified variable as a direct input. The second step, which is iterative, invalidates all subroutines that depend on invalid subroutines. Validity information is then available for all output variables, and variables not affected by the modified input do not require re-execution of analyses.

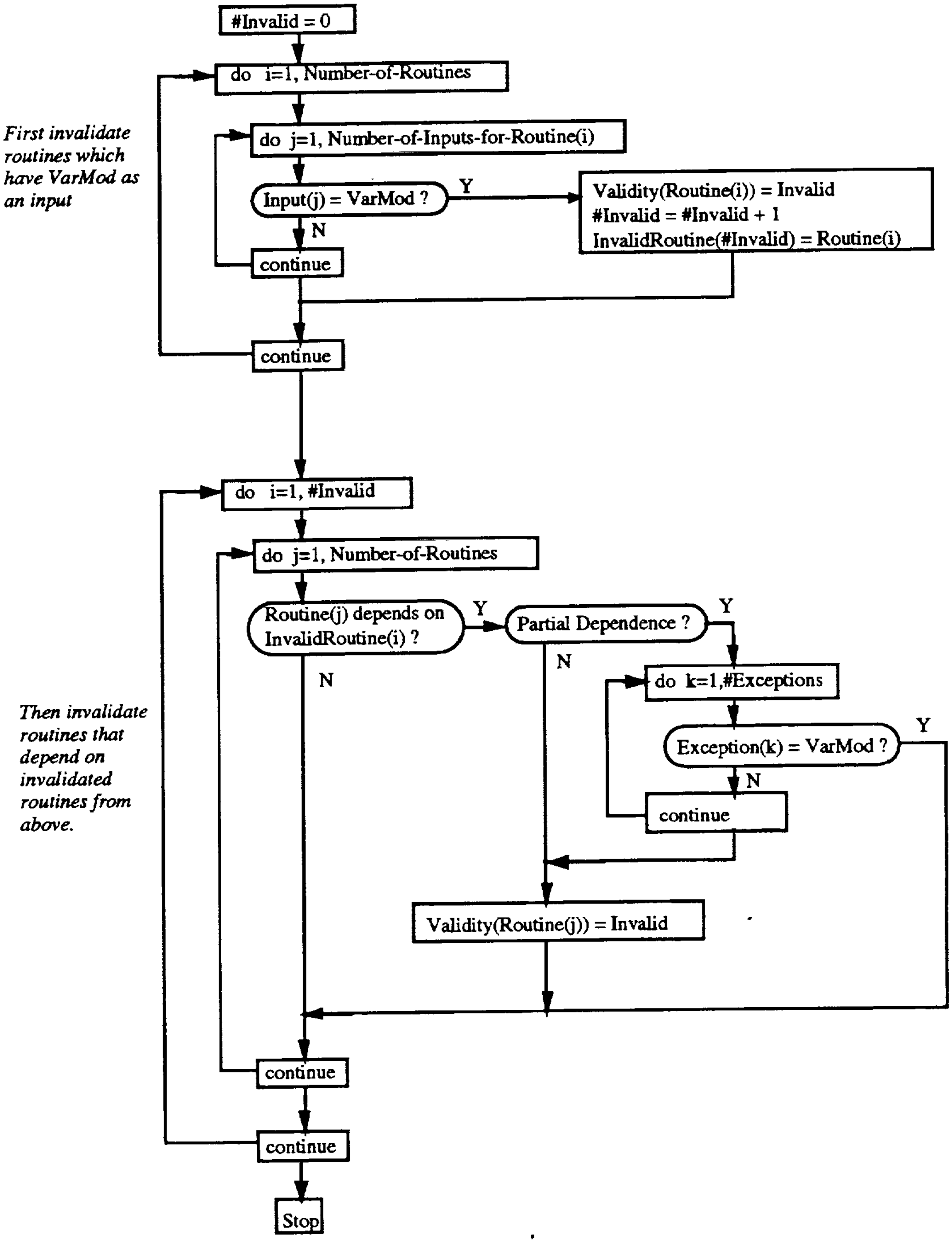

#Invalid = 0
do i=1, Number-of-Routines
do j=1, Number-of-Inputs-for-Routine(i)
Input(j) = VarMod ?
Y
N
Validity(Routine(i)) = Invalid
#Invalid = #Invalid + 1
InvalidRoutine(#Invalid) = Routine(i)
continue
continue
First invalidate routines which have VarMod as an input
do i=1, #Invalid
do j=1, Number-of-Routines
Routine(j) depends on InvalidRoutine(i) ?
Y
N
Partial Dependence ?
Y
N
do k=1,#Exceptions
Exception(k) = VarMod ?
Y
N
continue
Validity(Routine(j)) = Invalid
continue
continue
Stop
Then invalidate routines that depend on invalidated routines from above.

Figure A.7: Updating consistency information when input variable $VarMod$ has been modified.

Appendix B

A Genetic Optimization Package

The software described in this appendix allows both fixed- length and variable-length encodings. Successful applications to wing design (aerodynamics), truss design (structures), and interplanetary trajectory problems are described in the body of the thesis. It was originally based on a PASCAL listing from Goldberg [22], which used simple genetic operators and allowed only fixed-length binary encodings. The new algorithm is written in standard Fortran77, and has been run on Macintosh, IBM RS6000, Sun Sparc stations and SGI workstations. Its structure is outlined in Fig. B.1.

B.1 Operators of Genetic Optimization

Representation Each individual is represented by a string, which is a coded listing of the values of the design variables. The entire string is analogous to a chromosome, with genes for the different features (or variables). It is the genotype of the design. The design that is represented by the design variable values that are decoded fron the string is the phenotype. Binary, integer or real representations of the variables are all possible, and the 'best' encoding is problem-dependent. In this software, each string position is an integer. The

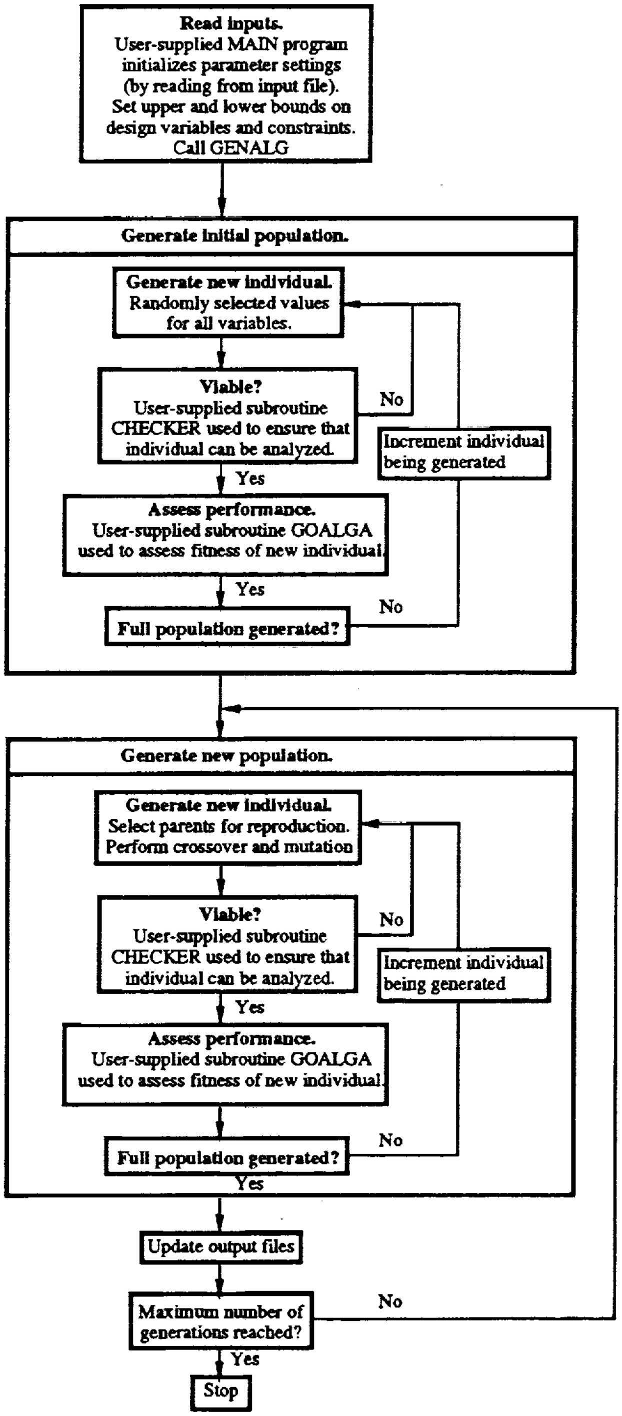

Read inputs.
User-supplied MAIN program
initializes parameter settings
(by reading from input file).
Set upper and lower bounds on
design variables and constraints.
Call GENALG
Generate initial population.
Generate new individual.
Randomly selected values
for all variables.
Viable?
User-supplied subroutine
CHECKER used to ensure that
individual can be analyzed.
No
Increment individual
being generated
Yes
Assess performance.
User-supplied subroutine GOALGA
used to assess fitness of new individual.
Yes
No
Full population generated?
Generate new population.
Generate new individual.
Select parents for reproduction.
Perform crossover and mutation
Viable?
User-supplied subroutine
CHECKER used to ensure that
individual can be analyzed.
No
Increment individual
being generated
Yes
Assess performance.
User-supplied subroutine GOALGA
used to assess fitness of new individual.
No
Full population generated?
Yes
Update output files
Maximum number of
generations reached?
No
Yes
Stop

Figure B.1: Flow diagram for genetic software.

user chooses how many positions will be used for each variable (*nbits*), and the range of possible values for each position (*rnggen*). The string must be decoded to form design variables appropriate for use in the analysis domain (Fig. B.2).

```
cpg   Decoding the kth variable of population member i
cpg   nbits = 1 -> integer encoding, otherwise binary
      if (nbits .eq. 1) then
         tmpvar = chrom(i,k)
      else
         tmpvar = 0
         do 17 k4=1,nbits
           j1=((k-1)*nbits)+k4
           tmpvar=tmpvar+chrom(i,j1)*(2**(nbits-k4))
17       continue
      endif

cpg   Convert integer tmpvar to real var, and modify scale
cpg   to be appropriate for analysis domain
      var(i,k)=scalemin(k) + scalerange(k)*tmpvar/rnggen
```

Figure B.2: Pseudo-code for decoding a genetic string.

Selection and replacement This is the non-random operator in genetic search. The promising members of the current population are favoured to contribute to new designs. In this software, selection is performed by tournament. A number of population members are chosen at random to compete for the right to reproduce, and the individual with highest fitness wins the tournament (Fig. B.3). There is a separate tournament (with new randomly chosen competitors from the existing population) each time a parent is required. When there is a large number of competitors in the tournament, the current best are more strongly favoured. The default tournament size is 2 ($ktourn = 2$).

The best member of the existing population is always retained in the new population (it does not need to be selected in a tournament). This means that the 'best of all generations' is always present in the latest generation, so it is easily identified when the program terminates.

```
      bestvalue = -100.
      do 10 i=1, ktourn
        call rndgen(ain,rand)
        candidate = int(rand*popsiz) + 1
        if (fitns(candidate) .ge. bestvalue) then
           bestvalue = fitns(candidate)
           k = candidate
        endif
10    continue
```

Figure B.3: Pseudo-code for tournament selection.

The initial population is currently generated at random (no selection can be applied at the start of optimization). Future versions will allow user-specified initial population. This would allow re-start from the final population of a previous search. It would also allow an experienced user to bias the search toward regions of expected high performance.

Crossover Crossover is an operation that forms new individuals from combinations of parts of parent strings. It is a powerful operator for recombining useful building blocks from different designs. This software uses a single-point crossover, meaning that each parent string is disrupted at only one point. At the point of crossover, the value of the variable is changed, according to the difference between values in the parents. When strings of varying length are allowed, the crossover point may be different for the two parents, and offspring of different sizes may be produced (Fig. B.4).

Mutation Pointwise mutation is performed on the offspring. Each point in each string has a chance of being mutated (Fig. B.5). The size of the resulting mutation is random, up to a user-specified maximum, $RngMut$.

Species formation In some problem domains it is desirable to locate several local minima, particularly if secondary minima are almost as good as the global

```
      if (rand .le. pcross) then
cpg Select crossover point in 1st parent
cpg (which has string length k1length)
         call rndgen(ain,rand)
         n=int(rand*k1length)+1
         if (n .gt. k1length) n=k1length
cpg Locate position of crossover point within substring (k1off)
         k1off = n - (int(n/(lchrom*nbits))*lchrom*nbits)
cpg If variable-length strings are used, select crossover
cpg point in 2nd parent
         if (.not. growth) then
            n2 = n
         else
            call rndgen(ain,rand)
            n2=int(rand*k2length)
            if (n2 .ge. k2length) n2=k2length-1
cpg Force 2nd crossover point to have same position within substring
cpg as 1st crossover point (k1off)
            n2 = ((n2/(lchrom*nbits))*lchrom*nbits) + k1off
         endif

cpg Retain encoding of 1st parent (up to crossover point n)
cpg Mutate variable at location n.
         call rndgen(ain,rand)
         tmpmut = rand * rngcross*abs((real(chrom(i,n)-chrom(i+1,n2))))
         call rndgen(ain,rand)
         if (rand .lt. 0.5) then
           chrom(i,n)=chrom(i,n) + int(tmpmut)
         else
           chrom(i,n)=chrom(i,n) - int(tmpmut)
         endif
cpg Append encoding from 2nd parent (after crossover point n2).
         translength = k2length - n2
         do 52 counter = 1, translength
           chrom(i,n+counter)=chrom(i+1,n2+counter)
52       continue

      endif
```

Figure B.4: Pseudo-code for crossover.

```
        do 60 j=1,ltchrm
           call rndgen(ain,rand)
cpg Each point in string has small chance of being mutated
           if (rand .le. pmutn) then
              if (nbits .gt. 1) then
cpg For binary string, mutation simply flips the bit value
                chrom(i,j)=1-chrom(i,j)
              else
cpg For integer string, mutation size is random
cpg (up to user-specified maximum: rngmut)
                call rndgen(ain,rand)
                tmpmut = rand * rnggen * rngmut
                call rndgen(ain,rand)
                if (rand .lt. 0.5) then
                  chrom(i,j)=chrom(i,j) + int(tmpmut)
                  if (chrom(i,j) .gt. rnggen)
     >             chrom(i,j) = chrom(i,j) - int(rnggen)
                else
                  chrom(i,j)=chrom(i,j) - int(tmpmut)
                  if (chrom(i,j) .lt. 0)
     >             chrom(i,j)=chrom(i,j) + int(rnggen)
                endif
              endif
           endif
60      continue
```

Figure B.5: Pseudo-code for mutation.

minimum. Choice between these local optima may be made by considering factors not modelled in the optimization task. In these cases, the introduction of a sharing parameter can help to induce 'species' formation during genetic optimization. Basically, this operator degrades the fitness of population members which have many other members nearby (in phenotype space), so that it is unattractive for the entire population to crowd together around the global optimum. The amount of degradation and the definition of 'nearby' can be controlled by the user. Description of sharing operator can be found in Chapter 3.

B.2 The Optimization Problem

The standard form for an optimization problem is to minimize an objective function, J, while satisfying a set of constraints, G, by changing the values of a set of design variables, X. The user must supply a subroutine to calculate objective and constraints (GOALGA). The design variables and fitness measure (which is a function of objective and constraints) must be passed between this routine and the genetic optimizer

In the genetic algorithm, the variables are represented as integer or binary strings. They must be decoded for use in the design domain, as described in the previous section.

Some problems may allow genetic strings which describe designs that cannot be analysed. (One example arises in wing design, where the wing is modelled by a vortex-lattice system. Vortex strength cannot be evaluated for designs that place vortices too close to control points.). A problem-dependent subroutine CHECKER is used to check viability, with non-viable designs being discarded before inclusion in the population. The population always consists of designs that can be analysed.

The software package includes a dummy CHECKER routine which simply returns *viable* = *true*. The user can introduce more elaborate checks on viability when necessary, by modifying CHECKER. This is intended to be used prior to complete analysis of the new offspring, and should be much cheaper than the full analysis.

The objective must be minimization of some scalar function, but the minimum must be positive (genetic algorithm doesn't work with objective values ¡ 0). The objective is passed from the GOALGA routine to the optimizer, where it is converted to fitness (used in selection operation) :

$Fitness = 1/(1 + objective)$

High fitness means high likelihood of reproduction, so this conversion makes designs with low objective likely to propagate into the next generation. It also guarantees fitness measures in the range 0 -¿ 1. Scaling of the objective in the GOALGA routine (to O(1)) will ensure that fitness is calculated with good precision.

Constraints are most easily handled by appending penalties to the objective function calculated in GOALGA. It is important to select the penalty weight carefully. If it is too heavy, selection strongly favours designs that simply satisfy constraints, and the population can converge to a sub-optimal design. If the weight is too light, an infeasible design will be favoured.

The care required for efficient use of penalties makes it desirable to avoid using them where possible. One alternative constraint-handling technique is to 'repair' a candidate design so that it satisfies constraints before evaluating its performance. Repair has been successfully implemented in a wing design problem, where the aim is to minimise drag while constrained to generate a fixed total lift. The incidence of the entire wing can be adjusted to satisfy this constraint before drag is calculated, and the relative incidences of different wing sections are design variables which allow adjustment of the distribution of lift to reduce drag.

When growth is allowed, constraint activity can change during optimization. Variable values that are favoured (selected) when a constraint is not active may not be appropriate when the design grows and makes it active. (An example of this is for wing dihedral variables and span constraints. When the span constraint is not active, low dihedral is favoured, and when the max span is reached there is no diversity of wing dihedral remaining in the population, and it can be difficult to discover winglets). To handle this situation, the algorithm allows multiple entries for each design variable in the genetic string (user sets *Nredun*). A change in constraint activity can cause different entries to be expressed. The constraint activity that causes expression of different entries must be described in subroutine CHECKER.

The genetic algorithm has no guaranteed termination criterion, so there is nothing like Kuhn-Tucker conditions to indicate when search is finished. The search is typically run for a prescribed number of function evaluations or generations. The user sets *MaxCalc* and *MaxGen* in the input file.

The optimum may not have been reached when the program terminates. When optimization performed from several different random starting populations produces similar final results, the user can be confident that the best design has been located. In fact, genetic algorithms are efficient at getting close to the optimum, but not very good at finding it exactly. It is often better to re-start several times and run with moderate population sizes and numbers of generations, rather than relying on a single very large run. The number of repetitions is set by *Nseed* in the input file.

B.3 Input and Output

genalg.inp Genetic algorithm parameters must be set before the genetic algorithm is run. It is convenient to list the desired settings in an input file (genalg.inp) that is read by the main program before the genetic algorithm is called.

nGA.out $n = \mathit{RunNumber} \times 100$
This output file gives an optimization history by generation. It lists generation number, best member of population, worst value in population, best value in population, average value in population and number of identical members in population.

nGA.lst $n = RunNumber \times 100$
This output file records the design variables of the best member in the population whenever those variables change. It also records the parents from which the new member was constructed. The last line records the total number of function evaluations in the run.

Npop.out $N = RunNumber \times 1000 + GenerationNumber$
This output file records the design variables for the population members in a given generation. Entire intermediate populations can be saved during optimization. $Pgen$ sets the gap between saved generations, and $Pfraction$ allows partial generations to be saved ($Pfraction \times popsiz$ individuals are saved).

B.4 User-Specified Input Parameters

$Growth$ (0 - off, 1 - on)

$Nbits$ Number of bits for each variable. If a single integer representation is used, $nbits = 1$.

$RngGen$ The number of distinct values the variable can attain. This is $2^n bits - 1$ if binary representation is used. Decoding of variables : $min + (value/rnggen) * (max - min)$

$PopSiz$ Population size (Maximum is 1000, can be changed by changing $MaxPop$ parameter in genalg.inc).

$MaxGen$ Maximum generations (Maximum is 500, can be changed by changing $LimGen$ parameter in genalg.inc)

$MaxCalc$ Maximum number of fuction evaluations.

$Pcross$ Probability of crossover. Usually 0.7 to 0.9

RngCross Range of Crossover (multiple of difference between parent values). Usually 2.

Pmutn Probability of mutation of each location in the string. The value will be different for integer and binary representations. Generally, choose a value to produce about 0.1 -¿ 1 mutations per string.

RngMut The maximum amount by which the mutation can change the value of the variable (as a multiple of *RngGen*). For binary representation, this is automatically 0.5 (flipping the most significant bit).

Lchrom This is the number of variables in a 'block'. When growth is not used, this should be equal to total variables (they are all in one block).

Nseed This is the number of repetitions to be performed for the same problem. Because the method is probabilistic, several repetitions should be run.

ktourn This is the number of population members that compete to be a parent in each reproduction operation.

OutUnit Unit number for printing to screen. 6 in standard Fortran, 9 for MacFortran

MaxVars Maximum number of variables. The total number of variables for fixed-length representations.

Pgen Number of generations between files recording population. Filenames have form: $(iseed * 1000 + gen)//'pop.out'$.

Pfraction Fraction of population to be recorded at *Pgen* generations.

Nredun Number of representations of each variable. User explains how to choose between alternative values in the CHECKER routine. For fixed complexity optimization, *nredun* = 1.

Npeaks Number of expected local minima. This affects the range over which crowding factor will be applied (higher npeaks -¿ smaller range of crowding penalty) Unless sharing is used, *Npeaks* = 1.

Sscale Severity of crowding penalty. Default = 1.

SigFac Scale factor for range (in phenotype space) over which the crowding penalty should be applied. Default = 0.5. Smaller reduces the distance for which the crowding penalty is applied.

alpha Exponent for severity of crowding penalty as function of distance between individuals. Default = 1, corresponding to linear decrease in penalty with distance. Higher values mean swifter reduction in penalty as distance increases.

MinMax Minimum and maximum bounds for each variable. There should be a minimum and maximum for *nblock* variables, where nblock is the number of variables in a building block (total number of variables when complexity is fixed). *Nblock* = *lchrom/nbits*.

B.5 User-Supplied Subroutines

Subroutine GOALGA This routine is the interface between the genetic algorithm and the domain-specific analyses. It takes design variable values from the genetic algorithm as input. If any constraints are handled by repair, they are evaluated first, and the design is modified as necessary. The objective is then evaluated. If any constraints are handled by penalty, they are evaluated last, and penalties are appended to the objective. The final objective value is output to the genetic algorithm. Evaluation generally requires execution of domain-specific analyses that are called from GOALGA. All necessary analyses must be provided by the user.

Subroutine CHECKER This routine is used to check that the candidate design can be analysed ($Viable = true$) before computation is attempted. If all possible candidates are analyzable, there is no need to check viability, and the simple CHECKER subroutine included in the software package can be used. The routine is also used to control the expression of multiple entries in the genetic string. (This feature is rarely employed, and can easily be ignored by inexperienced users.)

Bibliography

[1] Ashley, H., "On Making Things the Best - Aeronautical Uses of Optimization", Journal of Aircraft, vol. 19, pp 5-28, Jan 1982.

[2] Haftka, R.T.; Grossman, B.; Mason, W.H., "Multidisciplinary Aircraft Design: An Underutilized Capability", p B60, Aerospace America, July 1994.

[3] Nicolai, L.M., *Fundamentals of Aircraft Design*, METS, Inc., 1975.

[4] Gallman, J.W. "Aerodynamic and Structural Optimization of Joined-Wing Aircraft," Stanford PhD Thesis, June 1992.

[5] Torenbeek, E., *Synthesis of Subsonic Airplane Design*, Delft University Press, 1982.

[6] Shevell, R.; Kroo, I., "Introduction to Aerospace Systems Design Synthesis and Analysis", Course Notes, Department of Aeronautics and Astronautics, Stanford University, 1989.

[7] De Filippo, R., "ACSYNT Users' Guide", Northrop Aircraft, 1983.

[8] Rowell, L.F., "The Environment for Application Software Integration and Execution (EASIE) Version 1.0 - Volume 1 - Executive Overview", NASA TM-100573, August 1988.

[9] Tong, S.S., "Coupling Artificial Intelligence and Numerical Computation for Engineering Design (Invited Paper)." AIAA 24th Aerospace Sciences Meeting, Jan 6-9, 1986, Reno, Nv, AIAA-86-0242

[10] Tong, S.S., "Turbine Preliminary Design Using Artificial Intelligence and Numerical Optimization Techniques." Journal of Turbomachinery, January 1992, Vol 114/1.

[11] Bil, C., *Development and application of a computer-based system for conceptual aircraft design*, Delft University Press, 1988.

[12] Cousin, J.; Metcalfe, M., "The BAe (Commercial Aircraft) Ltd Transport Aircraft Synthesis and Optimization Program" AIAA/AHS/ASEE Aircraft Design, Systems and Operations Conference, Sept 17-19, 1990, Dayton, OH, AIAA-90-3295

[13] Myklebust, A.; Gelhausen, P., "Putting the ACSYNT on Aircraft Design", pp 26-30, Aerospace America, September 1994.

[14] Elias, A.L., "Knowledge Engineering of the Aircraft Design Process," in *Knowledge Based Problem Solving*, Kowalik, J., Ed., Prentice-Hall, 1986.

[15] Kolb, M., "A Flexible Computer Aid for Conceptual Design Based on Constraint Propagation and Component-Modelling", AIAA/AHS/ASEE Aircraft Design, Systems and Operations Meeting, Atlanta, September 7-9, 1988. AIAA-88-4427

[16] Takai, M., "A New Architecture and Expert System for Aircraft Design Synthesis," Stanford Ph.D. Thesis, June 1990.

[17] Kroo, I., Takai, M., "A Quasi-Procedural, Knowledge-Based System for Aircraft Design", AIAA/AHS/ASEE Aircraft Design, Systems and Operations Meeting, Atlanta, September 7-9, 1988. AIAA-88-4428

[18] Kroo, I., Takai, M., "Aircraft Design Optimization Using a Quasi-Procedural Method and Expert System", Third Air Force/NASA Symposium on Recent Advances in Multidisciplinary Analysis and Optimization, San Francisco, September 24-26, 1990.

[19] Arora, J.S., *Introduction to Optimum Design*, McGraw-Hill 1989.

[20] Arora, J.S.; Baenziger, G., "Uses of Artificial Intelligence in Design Optimization" *Computer Methods in Applied Mechanics and Engineering*, Vol 54, pp 303-323 1986.

[21] Rogers, J.L.; Barthelemy, J.-F. M., "An Expert System for Choosing the Best Combination of Options in a General Purpose Program for Automated Design Synthesis" *Engineering with Computers*, 1, pp 217-227 1986.

[22] Goldberg, D., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison Wesley, 1989.

[23] KrishnaKumar, K.; Swaminathan, R.; and Montgomery, L.; "Multiple Near-Optimal Solutions for a Structural Control Problem Using a Genetic Algorithm with Niching," AIAA 93-3873, in AIAA Guidance, Navigation and Control Conference, Technical Papers Pt 3, Monterey, CA, August 1993.

[24] KrishnaKumar, K., Goldberg, D., "Genetic Algorithms in Control System Optimization", AIAA Guidance, Navigation and Control Conference, August 20-22, 1990, Portland, OR.

[25] Hajela, P., "Genetic Search - An Approach to the Nonconvex Optimization Problem',*AIAA Journal*, Vol 28, No. 7, 1990.

[26] Bramlette, M., Cusic, R., "A Comparative Evaluation of Search Methods Applied to the Parametric Design of Aircraft", Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, 1989.

[27] Crispin, Y., "Aircraft Conceptual Optimization Using Simulated Evolution", 32nd Aerospace Sciences Meeting, Reno, NV, Jan 10-13, 1994. AIAA 94-0092

[28] Hutchison, M.G., Unger, E.R., Mason, W.H., Grossman, B., Haftka, R.T., "Variable-Complexity Aerodynamic Optimization of an HSCT Wing Using Structural Wing-Weight Equations." 30th Aerospace Sciences Meeting, Reno, NV, Jan 6-9, 1992. AIAA 92-0212

[29] Gallman, J.W., Kroo, I.M., Smith, S.C., "Design Synthesis and Optimization of Joined-Wing Transports", AIAA/AHS/ASEE Aircraft Design, Systems and Operations Conference, Sept. 17-19, 1990, Dayton, OH AIAA-90-3197

[30] Gallman, J.W., Kroo, I.M., "Structural Optimization for Joined-Wing Synthesis," Fourth AIAA/USAF/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization, Cleveland, September 21-23, 1992. AIAA-92-4761

[31] Gill, P.E., Murray, W., Saunders, M.A., Wright, M.H., "User's Guide for NPSOL (Version 4.0): A Fortran Package for Nonlinear Programming," Technical Report SOL 86-2, Department of Operations Research, Stanford University, Jan. 1986

[32] Gill, P.E., Murray, W., Wright, M.H., *Practical Optimization*, Academic Press, 1981.

[33] Haftka, R.T., Gurdal, Z., Kamat, M.P., *Elements of Structural Optimization*, Second Revised Edition. Kluwer Academic Press, 1990

[34] Luenberger, D.G. *Linear and Nonlinear Programming*, Addison-Wesley, 1984.

[35] Vanderplaats, G.N., *Numerical Optimization Techniques for Engineering Design: With Applications*, McGraw-Hill, 1984.

[36] Kroo, I., Wakayama, S., "Nonlinear Aerodynamics and the Design of Wingtips", Final Report, NASA Grant NCC2-683, April, 1992.

[37] Martens, P., "Airplane Sizing Using Implicit Mission Analysis", AIAA 94-4406 AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City, FL, Sept 7-9, 1994.

[38] Paisley, D., Martens, P., "Thoughts on the QPM Conversion Tool", Personal Communication, August, 1993.

[39] Dixon, L. "Use of Automatic Differentiation for Calculating Hessians and Newton Steps", in *Automatic Differentiation of Algorithms: Theory,Implementation, and Application* Ed. Griewank, A., Corliss, G., SIAM, Philadelphia, 1991.

[40] Bischof, C., Carle, A., Corliss, G., Griewank, A., Hovland, P., " ADIFOR: Generating derivative codes from Fortran programs", Scientific Programming, 1(1):1-29, 1992.

[41] Kroo, I.; Gage, P.; Altus, S.; Bischoff, C.; Hovland, P.; "New Approaches to Multidisciplinary Optimization," Distributed Computing for Aerosciences Applications, NASA Ames Research Center, October 18-20, 1993.

[42] Kroo, I., Altus, S., Braun, R., Gage, P., Sobieski, I., "Multidisciplinary Optimization Methods for Aircraft Preliminary Design", AIAA 94-4325 AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City, FL, Sept 7-9, 1994.

[43] McMasters, J.H., "Reflections of a Paleoaerodynamicist", *Perspectives in Biology and Medicine* 29, 3, Part 1, Spring 1986.

[44] Caldwell, C., Johnston, V., "Tracking a Criminal Suspect Through 'Face-Space' with a Genetic Algorithm", Proceedings of the Fourth International Conference on Genetic Algorithms, Ed. Belew, R., Booker, L., Morgan Kaufmann, 1991.

[45] Kosak, C., Marks, J., Shieber, S., "A Parallel Genetic Algorithm for Network-Diagram Layout", Proceedings of the Fourth International Conference on Genetic Algorithms, Ed. Belew, R., Booker, L., Morgan Kaufmann, 1991.

[46] Syswerda, S., Palmucci, J., "The Application of Genetic Algorithms to Resource Scheduling", Proceedings of the Fourth International Conference on Genetic Algorithms, Ed. Belew, R., Booker, L., Morgan Kaufmann, 1991.

[47] Jones, D., Beltramo, M., "Solving Partitioning Problems with Genetic Algorithms", Proceedings of the Fourth International Conference on Genetic Algorithms, Ed. Belew, R., Booker, L., Morgan Kaufmann, 1991.

[48] Back, T., Hoffmeister, F., Schwefel, H.-P., "A Survey of Evolution Strategies", Proceedings of the Fourth International Conference on Genetic Algorithms, Ed. Belew, R., Booker, L., Morgan Kaufmann, 1991.

[49] Dawkins, R., *The Blind Watchmaker*, Norton, 1986.

[50] van Laarhoven, P.J.M., Aarts, E.H.L., *Simulated Annealing: Theory and Applications*, Reidel, 1987.

[51] Davidor, Y., "Epistasis Variance: A Viewpoint on GA-Hardness", in *Foundations of Genetic Algorithms*, ed. Rawlins, G., Morgan Kaufmann, 1991.

[52] Liepins, G., Vose, M., "Deceptiveness and Genetic Algorithm Dynamics", in *Foundations of Genetic Algorithms*, ed. Rawlins, G., Morgan Kaufmann, 1991.

[53] McCulley, C., Bloebaum, C.L., "Optimal Sequencing for Complex Engineering Systems Using Genetic Algorithms", AIAA 94-4325 AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City, FL, Sept 7-9, 1994.

[54] Goldberg, D., "Sizing Populations for Serial and Parallel Genetic Algorithms", Proceedings, 3rd International Conference on Genetic Algorithms, 1989.

[55] Richardson, J.T., Palmer, M.R., Liepins, G., Hilliard, M., "Some Guidelines for Genetic Algorithms with Penalty Functions", Proceedings, 3rd International Conference on Genetic Algorithms, 1989.

[56] Tate, D.M., Smith, A.E., "Dynamic Penalty Methods for Highly Constrained Genetic Optimization", submitted to *ORSA Journal on Computing*, August 1993.

[57] Michalewicz, Z., Janikow, C., "Handling Constraints in Genetic Algorithms", Proceedings of the Fourth International Conference on Genetic Algorithms, Ed. Belew, R., Booker, L., Morgan Kaufmann, 1991.

[58] Janikow, C., Michalewicz, Z., "An Experimental Comparison of Binary and Floating Point Representations in Genetic Algorithms", Proceedings of the Fourth International Conference on Genetic Algorithms, Ed. Belew, R., Booker, L., Morgan Kaufmann, 1991.

[59] Koza, J.R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection* Cambridge, MA: MIT Press, 1992.

[60] Syswerda, G., "Uniform Crossover in Genetic Algorithms", Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, 1989.

[61] Grefenstette, J., "Optimization of Control Parameters for Genetic Algorithms", *IEEE Transactions on Systems, Man and Cybernetics*, SMC-16(1):122-128, 1986.

[62] Goldberg, D.; Richardson, J.; "Genetic Algorithms with Sharing for Multimodal Function Optimization", Proceedings of the 2nd International Conference on Genetic Algorithms, ed. Grefenstette, J.J., Lawrence Erlbaum Associates, 1987.

[63] Deb, K.; Goldberg, D.; "An Investigation of Niche and Species Formation in Genetic Function Optimization", Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, 1989.

[64] Hong, P.E.; Kent, P.D.; Olson, D.W.; and Vallado, C.A.; "Interplanetary Program to Optimize Simulated Trajectories," Volume I - User's Guide, NASA CR-189653, Martin Marietta Astronautics, October 1992.

[65] Friedlander, A.L.; "MULIMP, Multi-Impulse Trajectory and Mass Optimization Program," Report SAI 1-120-383-T4, Science Applications, Inc., April 1975.

[66] Horsewood, J.; Suskin, M.; and Isley, J.; "MAnE, Mission Analysis Environment for Heliocentric High-Thrust Missions," Version 1.0, AdaSoft, Inc., Sept. 1993.

[67] Mead, C.W.; Jones, M.F.; "Optimization of Ephemeridal Parameters for Minimum Propellant Requirements on Multiplanet Roundtrip Swingby-Stopover Missions," TM 54/30-189, LMSC/HREC A791436, Lockheed Missiles & Space Company, May 1968.

[68] Goldberg, D.; "Real-coded Genetic Algorithms, Virtual Alphabets and Blocking", University of Illinois at Urbana-Champaign, Technical Report No. 90001, September 1990.

[69] Braun R.D.; "The Influence of Interplanetary Trajectory Options on a Chemically Propelled Manned Mars Mission," The Journal of the Astronautical Sciences, Vol 38, No. 3, July-September 1990, pp 289-310.

[70] Hoffman, S.J.; McAdams, J.V.; and Niehoff, J.C.; "Round Trip Trajectory Options for Human Exploration of Mars," AAS Paper 89-201, AAS/NASA International Symposium on Orbital Mechnics and Mission Design, Vol. 69, Advances in the Astronautical Sciences, Greenbelt, MD, April 24-27, 1989.

[71] Striepe, S.A.; Braun, R.D.; Powell, R.W.; and Fowler, W.T.; "Influence of Interplanetary Trajectory Selection on Earth Atmospheric Entry Velocity of Mars Missions," Journal of Spacecraft & Rockets, Vol. 30, No. 4, July-Aug. 1993, pp. 420-425.

[72] Striepe, S.A.; Braun, R.D.; Powell, R.W.; and Fowler, W.T.; "Influence of Interplanetary Trajectory Selection on Mars Atmospheric Entry Velocity," Journal of Spacecraft & Rockets, Vol. 30, No. 4, July-Aug. 1993, pp. 426-430.

[73] Walberg, G.; "How Shall We Go To Mars?, A Review of Mission Scenarios," AIAA Paper 92-0481, 30th AIAA Aerospace Sciences Meeting, Reno, Nevada, January, 1992.

[74] Braun, R.D.; and Blersch, D.J.; "Propulsive Options for a Manned Mars Transportation System," Journal of Spacecraft & Rockets, Vol. 28, No. 1, Jan.-Feb. 1991, pp. 85-92.

[75] Lyne, J.E.; and Braun, R.D.; "Flexible Strategies for Manned Mars Missions Using Aerobraking and Nuclear Thermal Propulsion," The Journal of Astronautical Sciences, Vol. 41, No. 3, July-Sept., 1993, pp. 339-347

[76] "Space Transfer Concepts and Analysis for the Exploration Missions," Final Report, Contract NAS8-37857, Boeing Defense and Space Group, Advanced Civil Space Systems, Huntsville, AL, December 1991.

[77] Gould, S.J. *Ever Since Darwin: Reflections in Natural History*, Norton, 1977.

[78] Sakamoto, J.; Oda, J. "A Technique of Optimal Layout Design for Truss Structures Using Genetic Algorithm", AIAA 93-1582, Proc. SDM 93, pp. 2402-2408.

[79] Grierson, D.; Pak, W. "Optimal Sizing, Geometrical and Topological Design Using a Genetic Algorithm", Structural Optimization 6, 151-159 (1993)

[80] Grierson, D.; Pak, W. "Discrete Optimal Design Using a Genetic Algorithm", in *Topology Design of Structures*, ed Bendsoe, M.P.; Mota Soares, C.A.; Kluwer Academic, 1993.

[81] Koumousis, V. "Layout and Sizing Design of Civil Engineering Structures in Accordance with the Eurocodes", in *Topology Design of Structures*, ed Bendsoe, M.P.; Mota Soares, C.A.; Kluwer Academic, 1993.

[82] Hajela, P.; Lee, E.; Lin, C.-Y. "Genetic Algorithms in Structural Topology Optimization", in *Topology Design of Structures*, ed Bendsoe, M.P.; Mota Soares, C.A.; Kluwer Academic, 1993.

[83] Smith, S.F. "A Learning System Based On Genetic Adaptive Algorithms" PhD thesis, University of Pittsburgh,1980.

[84] Holland, J. *Adaptation in Natural and Artificial Systems* University of Michigan Press, 1975.

[85] De Jong, K., "Learning with Genetic Algorithms: An Overview" Machine Learning 3: 121-138, 1988.

[86] Goldberg, D.E., Korb, B., Deb, K. "Messy Genetic Algorithms: Motivation, Analysis and First Results", TCGA Report 89003, May 1989.

[87] Goldberg, D.E., Deb, K, Korb, B. "An Investigation of Messy Genetic Algorithms", TCGA Report 90005, May 1990.

[88] Chirehdast, M., Hae, C.G., Kikuchi, N., Papalambros, P.Y., "Further Advances in the Integrated Structural Optimization System (ISOS)", AIAA 92-4817, Fourth AIAA/USAF/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization, September 21-23, 1992, Cleveland,OH.

[89] Sankaranarayanan, S., Haftka, R.t., Kapania, R.K., "Truss Topology Optimization with Simultaneous Analysis and Design", AIAA 92-2315, Proc SDM 92, pp. 2576-2585.

[90] Bendsoe, M.P., Kikuchi, N., "Generating Optimal Topologies in Structural Design Using a Homogenization Method", Computer Methods in Applied Mechanics and Engineering 71 (1988) pp. 197-224.

[91] Przemieniecki, J.S. *Theory of Matrix Structural Analysis* Dover, 1985.

[92] Goldberg, D.; Samtani, M. "Engineering Optimization Via Genetic Algorithm", in *Electronic Computation: Proceedings of the Ninth Conference on Electronic Computation*, ASCE, 1986.

[93] Mitchell, T.M. "The Need for Biases in Learning Generalizations", in *Readings in Machine Learning*, ed Shavlik, J.W.; Dietterich, T.G.; Morgan Kaufmann, 1990.

[94] Reddy, G.M., Cagan, J. "Optimally Directed Truss Topology Generation Using Shape Annealing", DE-Vol. 65-1, Advances in Design Automation - Volume 1, ASME 1993 p749-759.

[95] Michell, A.G.M. "The Limits of Economy of Material in Frame-Structures.", Phil. Mag. (6), 8, 589 (1904)

[96] Morris, S. "Integrated Aerodynamic and Control System Design of Oblique Wing Aircraft", Stanford Ph.D. Thesis, January 1990.

[97] Wakayama, S. "Lifting Surface Design Using Multidisciplinary Optimization", Stanford Ph.D. Thesis, December 1994.

[98] Giunta, A.A. et al "Noisy Aerodynamic Response and Smooth Approximations in HSCT Design." AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City, FL, Sept 7-9, 1994. AIAA 94-4376

[99] Unger, E.R., Hall, L.E., "The Use of Automatic Differentiation in an Aircraft Design Problem." AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City, FL, Sept 7-9, 1994. AIAA 94-4260

[100] Kroo, I. M., "A Discrete Vortex Weissinger Method for Rapid Analysis of Lifting Surfaces," Desktop Aeronautics, P. O. Box 9937, Stanford, CA 94305, August 1987.

[101] Kroo, I. "A General Approach to Multiple Lifting Surface Design and Analysis", AIAA-84-2507, AIAA/AHS/ASEE Aircraft Design, Systems and Operations Meeting, San Diego, CA, October 1984.

[102] von Karman, T.; Burgers, J. "Airfoils and Airfoil Systems of Finite Span", *Vol II of Aerodynamic Theory, div. E, ch. IV, sec. 17*, W.F. Durand, ed. Springer (Berlin) 1935.

[103] Juliff, P. *Program Design* Prentice-Hall 1986.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 1995	**3. REPORT TYPE AND DATES COVERED** Contractor Report

4. TITLE AND SUBTITLE New Approaches to Optimization in Aerospace Conceptual Design **6. AUTHOR(S)** Peter J. Gage	**5. FUNDING NUMBERS** 505-69-50
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Stanford University Department of Aeronautics and Astronautics Stanford, CA 94305	**8. PERFORMING ORGANIZATION REPORT NUMBER** A-950044
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** NASA CR-196695

11. SUPPLEMENTARY NOTES

Point of Contact: Hiro Muira, Ames Research Center, MS 237-11, Moffett Field, CA 94035-1000;
(415) 604-5888

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified — Unlimited Subject Category 05	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 words)***

Aerospace design can be viewed as an optimization process, but conceptual studies are rarely performed using formal search algorithms. Three issues that restrict the success of automatic search are identified in this work. New approaches are introduced to address the integration of analyses and optimizers, to avoid the need for accurate gradient information and a smooth search space (required for calculus-based optimization), and to remove the restrictions imposed by fixed complexity problem formulations. 1) Optimization should be performed in a flexible environment. A quasi-procedural architecture is used to conveniently link analysis modules and automatically coordinate their execution. It efficiently controls large-scale design tasks. 2) Genetic algorithms provide a search method for discontinuous or noisy domains. The utility of genetic optimization is demonstrated here, but parameter encodings and constraint-handling schemes must be carefully chosen to avoid premature convergence to suboptimal designs. The relationship between genetic and calculus-based methods is explored. 3) A variable-complexity genetic algorithm is created to permit flexible parameterization, so that the level of description can change during optimization. This new optimizer automatically discovers novel designs in structural and aerodynamic tasks.

14. SUBJECT TERMS Aerospace design, Calculus-based optimization, Genetic optimization, Constraint-handling, Program architecture	**15. NUMBER OF PAGES** 187 **16. PRICE CODE** A09

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**