DESIGN FOR VEHICLE STRUCTURAL CRASHWORTHINESS VIA CRASH MODE MATCHING

by

Karim T. Hamza

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Mechanical Engineering)
in The University of Michigan
2008

To my parents Dalal and Tarek, and my lovely wife Aml

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Professor Kazuhiro Saitou for giving me the opportunity to work on this challenging topic and for providing continuous guidance, mentorship, advice, and support. I'd also like to thank the committee members for their valuable advice and comments.

This work was partially supported by Nissan Technical Center, North America, Inc. (NTCNA). Special thanks are due to Ms. Kumi Hatsukano, Mr. Noboro Tateishi, Mr. Masakazu Nakamura, Mr. Ken Jimbo and Mr. Akira Toyama for many insightful discussions. Partial support also came from ESI North America, Inc. and Livermore Software Technology Corp. through academic discounts of their commercial crash simulation software.

I would also like to thank members of the University of Michigan's Center for Advanced Computing, whose technical support greatly assisted this work. Special thanks go to Dr. Abjit Bose, Dr. Randall Crawford, Dr. Matthew Britt and Dr. Andrew Caird.

I would also like to thank all University of Michigan professors and administrative staff with whom I had the opportunity to take courses or engage in discussions. Finally and most importantly, I'd like to thank my family, friends, lab-mates, and colleagues for being there… for making the whole experience worthwhile.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

ABSTRACT

Vehicle crashworthiness is an important design attribute which designers strive to improve. However, design for structural crashworthiness is a difficult task. A vehicle structure must have the strength to shield the passenger compartment, as well as the compliance to cushion the impact energy. The physics that govern the crash phenomenon involves nonlinear interactions of impact, plasticity and contact mechanics. With current state of the art, no analytical models of reliable accuracy are deemed possible for vehicle structures of realistic level of complexity. The best known analysis method is nonlinear finite element (FE) modeling that includes the fine geometric details of the structural components, fine element sized mesh and detailed nonlinear material models. Detailed FE analysis however requires enormous computational resources thereby severely hindering the success of general-purpose optimization approaches that otherwise prove effective in a broad spectrum of problems. With limitations on the available computational resources, an approach which is more of an art used by vehicle designers is that of manipulation of the crash mode, or *crash mode matching*. The crash mode is the gross-motion of the structure, which is qualitatively assessed by observing the time history of deformation of various zones of the structure. Crash mode matching involves adjusting the design variables of the structure in order to achieve a desirable structural deformation history. An experienced designer can make a good estimate of what the desirable structural deformation history should be, and can perform the necessary adjustment to the design variables to attain it and thus obtain good designs. This manual crash mode matching typically requires only a few trial FE runs. This dissertation aims to develop an algorithmic design methodology for parametric structural crashworthiness optimization by formalizing the crash mode matching approach.

In order to formalize the crash mode matching approach, a quantitative representation of the crash mode is introduced. The crash mode is defined as a matrix of time series, with dimensions of the matrix being the structural location and type of deformation. The time series in each element of the matrix records the deformation history. A comparison metric is then introduced for the *degree of matching* between crash modes of different designs. The metric is the integral of the error between the time series in the elements of the crash mode matrix. Thus the comparison metric is itself a matrix of error values, with its dimensions being the structural location and type of deformation. Finally, an algorithm is design for automated crash mode matching. The algorithm heuristically directs stochastic sampling of the design space to regions which are expected to have better match to the desired crash mode. This is achieved by adjusting the mean and standard deviation of a normal distribution that governs the stochastic sampling of each design variable. Adjustment of the mean and standard deviation is performed via Fuzzy logic rules that are defined by the algorithm user in analogy to the type of decisions that an experienced designer would make when observing certain conditions in the structural crash mode. Introducing randomness into the sampling procedure allows for the algorithm to have global convergence properties, as well as accounting for the fact that *different expert designers* may have different opinions on how to modify a design.

Implementation of the proposed framework is applied to two real-life case studies involving front half of a vehicle, as well as full vehicle models. The studies show the success of the proposed methodology in attaining high performance designs, while requiring a modest number of detailed FE runs, and hence reasonable computational resources.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Passenger vehicle crashworthiness is extremely important for public safety. According to the National Highway Traffic safety administration (NHTSA)[*], in the year of 2000, there were more than six million highway traffic crashes, in which included:

- 41,821 fatal crashes
- 3,189,000 crashes resulting in injuries
- 4,286,000 crashes resulting in property damage only

Although continuous efforts are spent towards accident prevention through better highway traffic management and accident prevention systems in vehicles, the possibility of occurrence of a crash can never be brought to zero. Therefore, vehicle safety in the event of a crash is an important design attribute for vehicle manufacturers. Aside from government and insurance standards, which are nowadays a requirement before vehicle manufacturers are permitted to put a new vehicle on the road, vehicle manufacturers strive to improve the crashworthiness of their vehicles as a moral responsibility as well as improve the market attractiveness of their vehicles.

[*] http://www.nhtsa.dot.gov/

1.2 Background

1.2.1 Design for Structural Crashworthiness

Safety systems in vehicles after a crash event initiates are mostly passive in nature, except for air bags. Passive crash safety systems in a vehicle may be categorized into:

- Structural crashworthiness systems: this is the performance of the vehicle structure during the crash event
- Non-structural systems: such as seat belts

Seat belts and airbags serve to cushion high acceleration spikes on the passenger, but have insignificant contribution on dissipating the crash energy of the vehicle. The major opportunity for continued improvement of the overall vehicle crashworthiness is that of structural design for crashworthiness, which is the focus of this research. In the rest of this dissertation, the term “crashworthiness”, unless otherwise stated, will be referring to structural crashworthiness.

Design for vehicle structural crashworthiness is a difficult task due to the combined effect of several issues:

- Actual crash testing, is a destructive type testing and only possible after a vehicle prototype is built, thus very expensive and time consuming

- Analytical modeling of real vehicle structure is all but impossible, since the underlying physics involves combinations of dynamic impact, plasticity, contact mechanics and very complex geometrical shapes.

- Numerical modeling of the crash event is difficult. Recent advances in the finite element (FE) method have been incorporated into commercial packages that enable designers to obtain fairly accurate estimates of a structural crash performance. However, to achieve acceptable accuracy, the FE model of the vehicle typically includes hundreds of thousands, or even millions of elements. Such detailed FE models require enormous (expensive) computational resources. While many attempts have been made at reduced order numerical models, none have proven consistent accurate crash performance predictions

- Design decisions are often non-obvious. The structure should have enough compliance to absorb and cushion the crash energy, but must retain integrity in sensitive areas such as the passenger compartment and fuel system. Also, due to the underlying nonlinear physics, the design variables sometimes have strong nonlinear interactions that are difficult to predict *a-priori*. The nonlinear variable interactions also lead to high chance of existence of local optima.

In view of such difficulties, the dominant design automation approach in practice is that of conducting design of experiments (DOE) to sample the design space via a FE model (Phadke 1993), then use the samples to construct a response surface model (RSM). The RSM maps the inputs (design variables) to the outputs (crashworthiness measures), and is then used within an optimization algorithm to estimate desirable vales for the design variables (Simpson *et. al.* 2004). The design suggested via the RSM is then re-tested via FE simulations, and if discrepant, a new DOE/RSM study may be conducted. The main difficulties of the approach however are:

- Difficulty in attaining good fidelity for the RSM. If the design domain is large (Simpson *et. al.* 2004), an extremely large number of samples in DOE are needed to properly capture the functional nonlinearity between inputs and outputs. If the design domain is confined to small vicinity, then the results obtained via the RSM are local only to such vicinity.

- The computational resources required for the approach are quite enormous. A study in (Yang *et. al.* 2001) required 512 processors running in parallel for 72 hours for a half-million element FE model. As computer computational speed increases, so does the level of detail in FE models, thereby the total invested time and cost to perform this kind of study remains all but prohibitive.

- Response surface modeling is a general purpose technique that abstracts the underlying physics of the problem and does not attempt to take advantage of them. Abstraction of the underlying physics prevents expertise-based design adjustment. For example, a designer observing the structural behavior during crash of a FE model may gain insight as to where reinforcement needs to be made at a structural zone that is deforming more than intended. Such type of insight is not available when only observing input-output relations from a response surface model.

1.2.2 Structural Crash Modes and Crash Mode Matching

Another approach used in practice by vehicle designers, is that of *manipulation of the crash mode*. This involves the pre-planning of a desirable sequence of deformation, folding and crush of the different structural zones, then the adjustment of the structural design to attain the desired structural deformation history. Such approach is more of an art than a formal algorithmic procedure. There are no formal rules for discovering a good structural deformation sequence, or for adjusting the design variables to attain such a sequence. Despite the lack of formality in this approach, an experienced designer is typically able to attain a good design with only a handful of trial iterations, which is significantly more efficient than any other known automated design approach.

A simple illustration of the crash modes is shown in Fig. 1.1, where design of a single structural member is considered. Since the crash energy has to be absorbed via deformation in the structure, an example of a desirable crash mode is shown in Fig. 1.1.b. In this crash mode, the structure deforms at the front (zone 1) in order to absorb crash energy. With most of the crash energy dissipated as plastic deformation in zone 1, this prevents excessive deformation near the passenger compartment (zone 2). An example of an undesirable crash mode is the one shown in Fig. 1.1.c. In this crash mode, zone 1 is excessively strong (or zone 2 is insufficiently strong), resulting in most of the structural deformation occurring in zone 2 (near passenger compartment), which is undesirable.

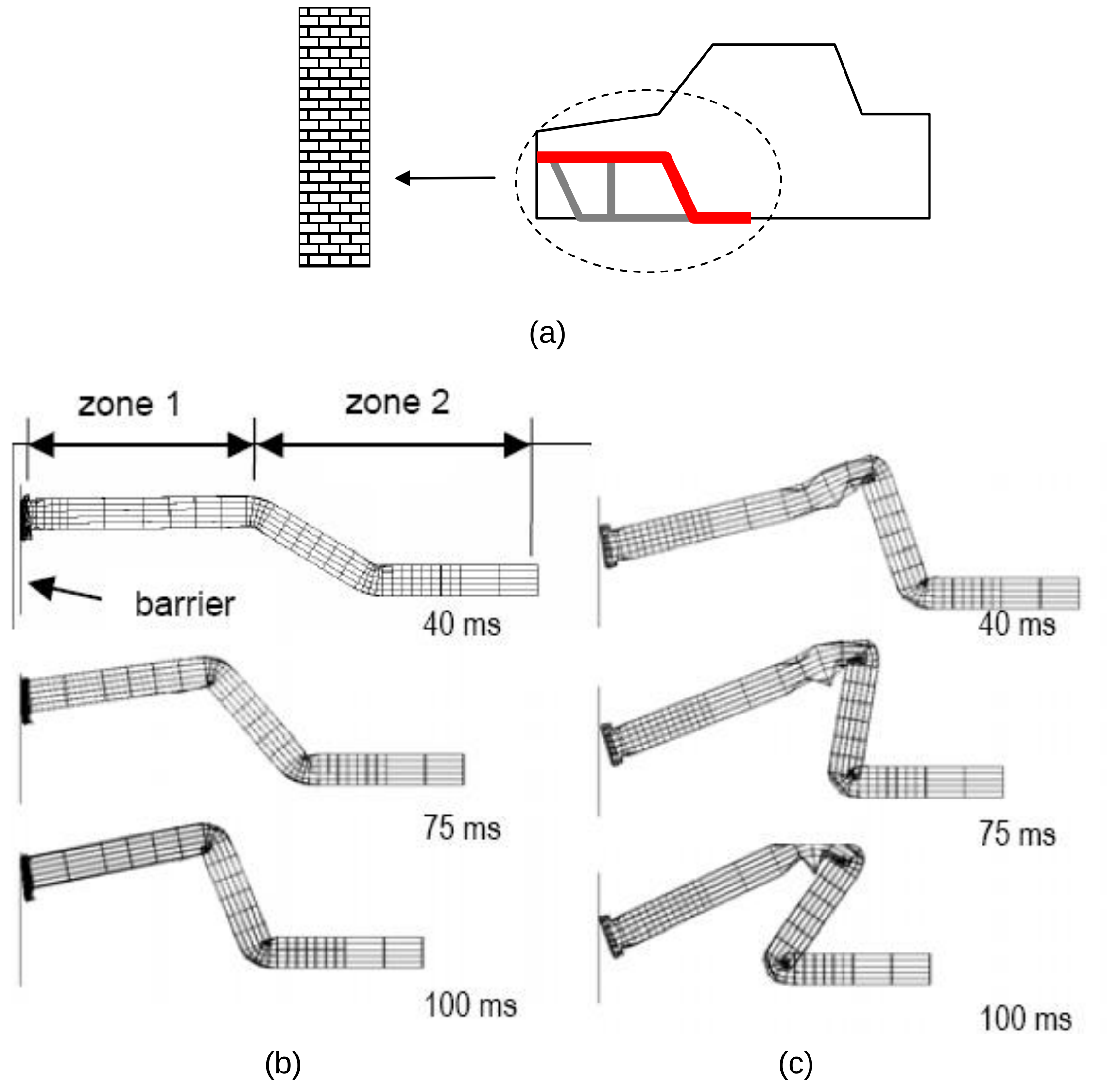


Figure 1.1. Examples of crash modes in Vehicle Structures: (a) Structural member considered. (b) Desired crash mode. (c) A design that has a bad crash mode.

An example of crash mode matching approach is shown in Fig. 1.2. Steps of the crash mode matching in this example are listed as:

- Given a design whose crash mode is sketched in Fig. 1.2.a, which initially deforms in zone 1 but does not absorb all the crash energy there, so deformation in zone 2 follows, a designer decides to make zone 1 stronger in order to absorb more energy in zone 1
- Making zone 1 too strong however, may result in zone 1 not deforming and all (sketched in Fig. 1.2.b), which results in all the deformation happening in zone 2 (undesirably large deformation). A stronger structural zone absorbs more crash energy only if deformation happens in that zone. A designer attempting to correct the crash mode might then decide to make zone 2 stronger and slightly reduce the strength of zone 1.
- With a proper balance of the relative strength between zones 1 and 2, with enough strength in zone 1, the designer is able to attain the desired crash mode where minimal deformation occurs in zone 2 (sketched in Fig. 1.2.c)

In analogy, the RSM approach is like a blind person with a cane (cannot see the underlying physics), while the crash mode matching is like a weak sighted person (cannot fully predict the physics, but has some grasp on it). In general, being weak sighted is better than totally blind. Thus, if it is possible to formalize the crash mode matching approach, the approach would potentially give a better edge over all others that don’t recognize the underlying physics of the crash. Attempts at such formalization seem to be largely overlooked in the literature.

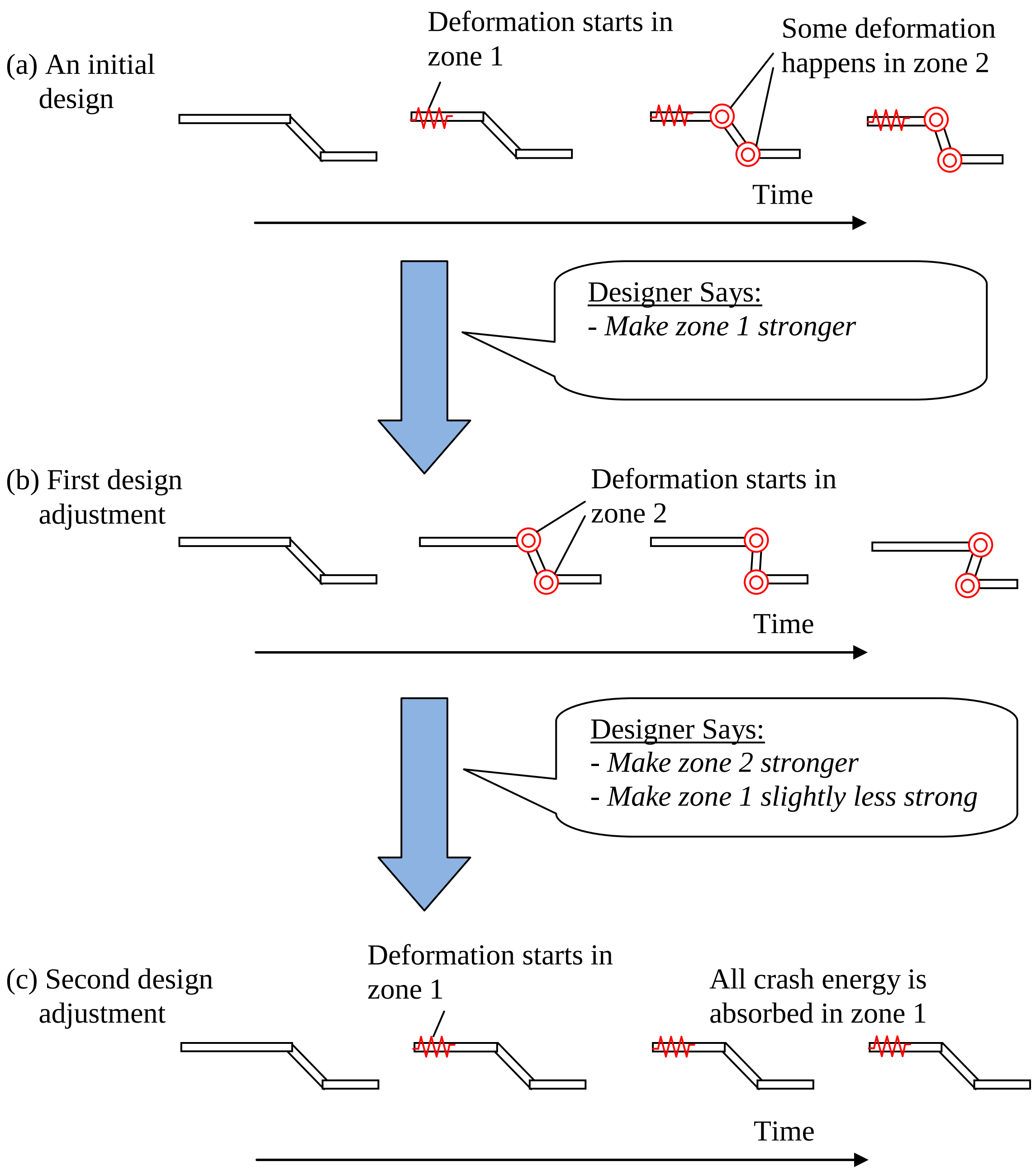

(a) An initial design
Deformation starts in zone 1
Some deformation happens in zone 2
Time
Designer Says:
- *Make zone 1 stronger*
(b) First design adjustment
Deformation starts in zone 2
Time
Designer Says:
- *Make zone 2 stronger*
- *Make zone 1 slightly less strong*
(c) Second design adjustment
Deformation starts in zone 1
All crash energy is absorbed in zone 1
Time

Figure 1.2. Example of Crash Mode Matching

1.2 Thesis Goal

This research aims to achieve the following goals:

- *Formalization of the crash mode matching approach*
- *Design of an algorithm that implements crash mode matching for parametric structural crashworthiness optimization*

1.3 Approach

In order to formalize the crash mode matching approach, the first step was to introduce a quantitative representation of the crash mode. The representation is defined as a matrix of time series, with dimensions of the matrix being the structural location and type of deformation. The time series in each element of the matrix records the deformation history (discussed in detail in chapter 5). This representation is to replace qualitative visual observation of the structural deformation history.

The next step was to define a comparison metric for the degree of match between the crash modes of a design and that desired. The metric is the integral of the error between the time series in the elements of the crash mode matrices. Thus the comparison metric is itself a two dimensional matrix of error values, with its dimensions being the structural location and type of deformation, which is a fairly compact data set to examine.

The last step was to introduce an algorithm for automated crash mode matching. The algorithm heuristically directs stochastic sampling of the design space to regions which are expected to have better match to the desired crash mode. This is achieved by adjusting the mean and standard deviation of a normal distribution that governs the stochastic sampling of each design variable. Adjustment of the mean and standard deviation is performed via Fuzzy logic rules that are defined by the algorithm user in analogy to the type of decisions that an experienced designer would make when observing certain conditions in the structural crash mode. Introducing randomness into the sampling procedure allows for the algorithm to have global convergence properties, as well as accounting for the fact that *different expert designers* may have different opinions on how to modify a design.

Being a random search algorithm, its performance may vary from one run to another for the same problem. However, the algorithm does conform to the necessary conditions to assure global convergence given sufficient number of iterations. Although the global convergence property has little practical value when considering only few allowed iterations, the case studies presented in this thesis demonstrate the possibility of attaining good designs with relatively few iterations.

A necessary additional piece of information that the developed algorithm requires is the quantitative values for the desired crash mode. While discovery of the desired crash mode remains an open-ended problem, it is often possible to adapt prior knowledge from previous vehicle designs to estimate the desired crash mode. While not proven, it is also observed that reduced order dynamic models could be effective for discovery of desirable crash modes. Demonstrated in the thesis is that although lacking in accurate predictions, reduced order models are often capable of capturing the gross motion of a structure, hence able to observe crash modes.

1.4 Organization of the Dissertation

The outline of this dissertation is summarized as:

Chapter 1 introduces the problem of structural crashworthiness design, highlights the main difficulties associated with it, and states the course of action to address some of those difficulties.

Chapter 2 reviews the state of the art literature related to the field. The literature review includes sections that examine types of optimization problems involved, crash simulation models as well as the optimization algorithms.

Chapter 3 presents a formal definition of the crash worthiness design optimization problem that is the focus of this dissertation and gives an overview of the proposed methodology.

Chapter 4 discusses crash modes in the qualitative sense via simple examples. The main hypothesis about crash modes is stated. Exploration of crash modes for a given vehicle structure is also discussed. Equivalent mechanism model, which is a reduced order dynamic model developed specifically for approximated modeling of structural members subjected to crash conditions, is presented as one of the options for qualitative exploration of crash modes.

Chapter 5 presents numerical measures for quantitative analysis of crash modes, as well as for comparison of crash modes in different structures.

Chapter 6 introduces an automated design algorithm for structural optimization for crashworthiness via crash mode matching. Analysis of the global convergence properties of the algorithm via Markov chains is also presented.

Chapter 7 presents the first case study in this dissertation. The case study involves optimization of an idealized vehicle model, with the front half of the vehicle modeled as box-section structural members and subjected to frontal crash against a rigid barrier. The optimization objective is to reduce the structural mass, while complying with constraints on allowable deformation and maximum acceleration.

Chapter 8 presents the second case study in this dissertation. The case study involves a multi-objective optimization for minimizing deformation and acceleration in a full vehicle model subject to offset frontal crash against a deformable barrier.

Chapter 9 concludes the dissertation. A summary of contributions and discussion of future extensions is presented

CHAPTER 2
RELATED WORK

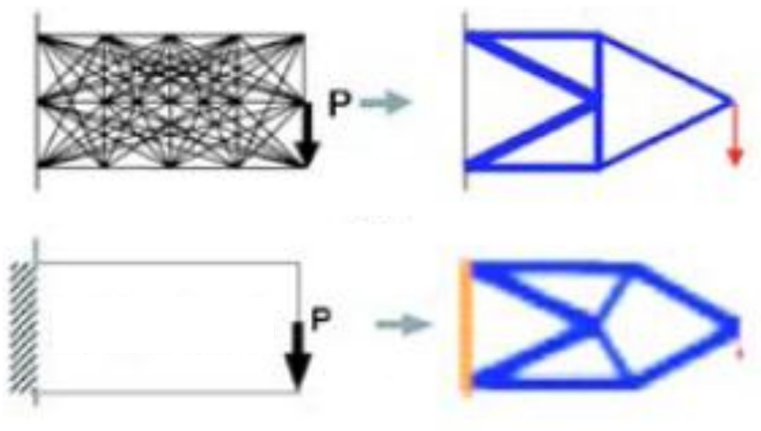
This chapter presents a review of relevant literature to the research conducted in this dissertation. At first, the review covers the broad categories of structural crashworthiness design then focuses on the main interest of the dissertation, which is parametric structural crashworthiness design. Special sections are dedicated to modeling and algorithms for crashworthiness. The chapter concludes with a highlight on the dissertation thrust areas.

2.1 Design Optimization for Structural Crashworthiness

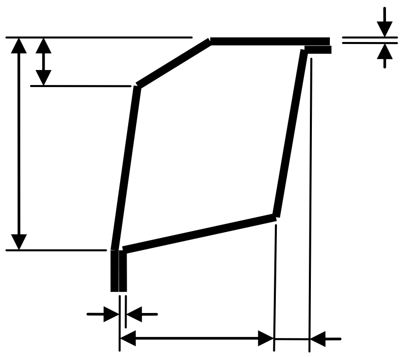
Design optimization for vehicle structural crashworthiness may be categorized into two broad categories according to the objective of the optimization. Those are:

- Topology Optimization for Structural Crashworthiness
- Parametric Optimization for Structural Crashworthiness

In topology optimization, the objective is to perform optimum allocation of *structural material* to an allotted space with the structure is allowed to occupy. Examples of topology optimization for structural crashworthiness may be found in: (Mayer, Kikuchi and Scott 1996, Soto and Diaz 1999, Luo *et. al.* 2000, Mayer 2001, Gea and Luo 2001, Soto 2001).



(a) Example Topology Optimization: structural conceptual design generation (Saitou *et. al.* 2005)



(b) Example Parametric Optimization: Sizing of the dimensions of a sheet metal component cross section

Fig. 2.1. Topology Optimization verses Parametric Optimization

The output of topology optimization is typically a *structural concept* rather than an actual final design of the structure (Fig. 2.1.a). While useful at the early design stages of a completely new vehicle, results of topology optimization seldom take into consideration many of the structural details such as component manufacturing and assembly, joints and non-structural components packaging. Since such construction details will effectively alter the structure from the idealized model optimum topology, the need for further design adjustments after topology optimization is still a necessity. In some cases, the structural topology is all but already pre-dictated by the vehicle styling, and components packaging. In which case, topology optimization for structural crashworthiness has secondary importance relative to optimizing the final structures, a task which is the objective of parametric optimization.

Parametric optimization for structural crashworthiness proceeds from fixed topology. In parametric optimization, a set of *sizing* design variables is defined. Typical design variables definitions include dimensions, sheet metal thicknesses and materials of structural components. Examples of parametric design optimization for structural crashworthiness may be found in: (Yang *et. al.* 1999, Chen 2001, Kurtaran *et. al.* 2001, Yang *et. al.* 2001, Redhe *et. al.* 2002, Andersson and Redhe 2003, Gu *et. al.* 2004, Hamza and Saitou 2005)

Parametric optimization for structural crashworthiness can tackle very detailed FE models (Yang *et. al.* 2001) of the vehicle that have high performance prediction accuracy (Fig. 2.1.b). Parametric optimization is suitable for final design adjustments, and it is the focus of this dissertation. In further discussion in the dissertation, parametric structural crashworthiness design optimization is referred to as “crashworthiness design.”

2.2 Crash Simulation Models used in Crashworthiness Design

Any design optimization process typically involves several iterations of trial designs. Building real prototypes in order to perform the NHTSA standard safety tests is a requirement, however conducting such tests is extremely costly and time consuming, and is thus only used for final design verifications. Some *virtual* crashworthiness testing is required during the design iterations. Closed-form analytical solutions are not an option for use as a simulation tool for virtual crashworthiness testing due to the extreme complexity of the crash phenomenon. Closed-form analytical solutions are all but impossible to obtain for vehicle structures of any real-life level of complexity. Instead, numerical approximation models have to be used. Typical numerical approximation used for crash simulation may be classified into three broad categories:

- Detailed nonlinear finite element (FE) models
- Reduced order dynamic models
- Functional approximation or response surface models (RSM)

2.2.1 Nonlinear Finite Element Models

Detailed nonlinear finite element models provide the best known accuracy solutions to the estimation of structural crashworthiness performance. With the implementation of many recent advancements in the finite element methods into commercial software packages (LSTC 2001, ESI 2003), these models have become the norm for the estimation of structural crashworthiness performance. Examples of detailed FE for crashworthiness design of a full vehicle models may be found in (Yang *et. al.*

2001, Soto 2001, Andersson and Redhe 2003, Hamza and Saitou 2005) and for crashworthiness design of individual components (sub-structures) in (Chen 2001, Koanti and Caliskan 2001, Kurtaran *et. al.* 2001, Soto 2003, Hamza and Saitou 2003).

While nonlinear finite element models provide the best known accuracy in estimation of structural crashworthiness performance, their typical downside is two-fold:

- Models that examine only sub-structures are not guaranteed to provide a good estimate of the sub-structure when other interacting members are present
- Models that include full vehicle details require enormous computational resources, thereby hindering the direct applicability of many (if not all) design optimization algorithms

2.2.2 Reduced Order Dynamic Models

Reduced order dynamic models for structural crashworthiness design have been around since the time when the computational resources for FE methods were not easily available (Song 1986). These types of models range from lumped mass models (Beneet *et. al.* 1991) coarse-mesh FE models (Chellapa and Diaz 2002) and fine-grained lumped models (Abramowicz 2003, Abramowicz 2004, Takada and Abramowicz 2004, Hamza and Saitou 2004). While differing in details of implementation, these types of models have common treats (to various extents):

- These models attempt to capture the gross motion of the structure during the crash event
- The models require less computational resources the detailed FE models, since they deal with a reduced number of differential equations

Reduced order dynamic models also have typical downsides:

- Introduction of an extra level of abstraction between the model and the actual physical structure of the vehicle. For example, the dimensions and sheet metal thickness in some structural component in a FE model are a representation of the same dimensions and sheet thickness in the real structure. In a reduced order dynamic model however, a nonlinear spring may represent a structural member. The spring parameters (such as stiffness) are not as easy to correlate to actual dimensions of the structural member in the real structure

- Less accuracy of performance prediction, as with any dynamic model with reduced number of degrees of freedom compared to FE models (Takada and Abramowicz 2004, Hamza and Saitou 2004)

These downsides are the reason why such models are seldom used in practice by vehicle designers

2.2.3 Response Surface Models

Functional approximation models or response surface models (RSM) are general-purpose meta-models, which are popular in many engineering applications. A meta-model is essentially a “model of a model”. When some application involves a model that has expensive computations (such as a detailed FE model), then the meta-model provides a computationally efficient approximation of it.

Several examples of such meta-models are listed in the review article in (Simpson *et. al.* 2004). In general, RSM are constructed via two steps:

1. Conducting design of experiments (DOE) to *acquire sample data* of some trial designs. Performance evaluation of the sample data typically involves the computationally expensive numerical models.

2. *Fitting of the sample data* via a functional approximation meta-model. Examples of such meta-models include polynomial regression (Box *et. al.* 1978, Myers and Montgomery 1995, Yang *et. al.* 2001), various types of neural networks (Dyn *et. al.* 1996, Haykin 1998, Hansen and Salamon 2002,) as well as Kriging (Krige 1951, Sasena *et. al.* 2002).

RSM models that are constructed via detailed nonlinear FE models seem to be the dominant popular design automation approach for structural crashworthiness optimization in practice. However, RSM models for crashworthiness have some serious drawbacks. Their popularity is mainly attributed to the *unpopularity* of the other options; extreme computational resource requirements for using detailed FE throughout all the design iterations, or the extra level of abstraction involved in using reduced order dynamic models. The downsides of RSM models are:

- Although inputs to a RSM can be the same design variables as inputs to a detailed FE model, there is a high level of abstraction and loss of the physical sense of the problem. After a RSM model is constructed, one could observe the predicted performance change (of total deformation for example) verses a change in a design variable, such as a structural component size, but provides no insight into *how* such performance change happened, such as which structural component deformed more or which zone of the structure is a likely candidate for a reinforcement.

- Due to the nonlinear nature of the underlying physics of the crash phenomenon, there are often many nonlinear interactions between the design variables. Such interactions make it extremely difficult to obtain high fidelity RSM over a wide range of the design variables while using a reasonable number of sample in the DOE

- RSM models are general purpose meta-models that are independent underlying-physics of the application they are used in. Oftentimes when tackling the design of a particular engineering application, there’s more information about the physics that could aid the designer (for example, in a linear elastic structure, adjusting for uniform strain is optimal utilization of structural material to maximize crash energy absorption). Thus use of a meta-model that has this independence of underlying-physics, eliminates the chance to incorporate physical-based design insights in guiding the optimization.

2.3 Optimization Algorithms for Structural Crashworthiness

Publications in the literature that employ reduced order models tend to have their focus set on *modeling* and accuracy issues rather than design optimization *algorithms*. This is mainly because reduced order dynamic models tend to have low computational requirements, therefore can be easily linked with off the shelf optimization algorithms that are available in commercial optimization software packages. However, the accuracy and fidelity of reduced order dynamic models are usually the prime concern.

Publications that focus on crashworthiness optimization algorithms are usually referring to algorithms that would be linked to a detailed nonlinear FE model, often within the framework of a RSM that serves as a surrogate for performance estimation within the iterations of the optimization algorithm.

In (Yang *et. al.* 2001), the optimization algorithm that runs on the RSM was Sequential Quadratic Programming (SQP). The constructed RSM was a second order polynomial regression, and hence SQP seemed a natural choice. Construction of the RSM is the bulk of the computational work when compared to several calls to the RSM for estimating performance. Thus when employing more elaborate RSM models than second order polynomial regression, several global search techniques (Michalewiz, and Fogel 2000) such as Simulated Annealing, Genetic Algorithms, Tabu search, DIRECT can be effectively employed. Genetic algorithms (GA) seem to be a popular choice to run optimization on the RSM (Chen 2001, Andersson and Redhe 2003). The basic steps in a genetic algorithm (Goldberg 1989) are explained as follows:

1. Define the *search space* via upper and lower bounds on all design variables
2. Randomly initialize a *Current Population* of designs *P* within the search space
3. Evaluate the objective function(s) for every design in *P*
4. Assign a *Fitness Value* to every design in *P*, the fitness function is chosen so that the better designs have better fitness
5. Pass the current best design into a *New Population* of designs P_{new}
6. Select two designs from *P*, the selection is randomized, but giving higher probability of selection to designs that have higher fitness.
7. Perform *mating* between the selected designs to produce two new designs that are added to P_{new}. The *mating* usually involves a combination of simple copying, linear averaging, projection and partial randomization.
8. Repeat at step 6 until the number of designs in P_{new} is equal to those in *P*, then P_{new} replaces *P* and becomes the *Current Population*
9. Repeat at step 3 until termination condition. Termination condition may be a pre-set number of iterations or discovery of a design with a target performance
10. Return the best design in current population

Convergence of the genetic algorithm to the global optimum within the search space pending satisfaction of certain convergence conditions is shown in (Goldberg 1989). From a practical point of view, application of GA is seldom *guaranteed* to provide the global optimum; however, its capability to continually sample the search space and not get trapped in a local optimum is the main source for its popularity. The anatomy of GA also makes the number of design samples it requires rather huge, which makes it prohibitively expensive to link to a detailed FE model, but a good on a pre-constructed RSM. The thing to remember is that design recommended by the GA can be the *optimum of the RSM model* (not the detailed FE model), which is only as good as accuracy of the RSM model itself.

Another class of RSM-based algorithms is that of incremental model building and enhancement of the RSM model. A notable approach that could prove effective, although not known to have been applied to Crash optimization problems, is the SuperEgo algorithm (Sasena *et. al.* 2002). This algorithm combines DIRECT as a sampling algorithm along Kriging as an incrementally enhanced RSM. The difficulty in applying this approach is that the technique is possibly more sensitive to the increase in the number of design variables (requires drawing more new samples of detailed FE simulations, exponentially increasing as the number of design variables increases) than genetic algorithms running on RSM (only uses the initial pre-constructed RSM model).

Among possible remedies of inaccuracies in non-incrementally enhanced RSM models, is the construction of several RSM from different sets of DOE samples, then running an optimization algorithm on each RSM model separately. The recommended optimum of each RSM is then examined via the detailed FE model, and the best among all is the one reported. A notable optimization algorithm that works along these lines is provided in (Hamza and Saitou 2005). The technique is termed Multi-scenario Surrogates Co-Evolutionary Genetic Algorithm (MSCGA). MSCGA runs GA populations on two or three RSM models constructed from different sets of DOE samples, with the GA "fitness" giving credit to designs that are estimated to have high performance among all the RSM models. The reported result of MSCGA is a set of designs, with the set extremities being the individual optimum of each RSM model, and the "center" of the set being designs that are estimated to have good performance among all the RSM models.

2.4 Thrust Area for this Dissertation

Detailed nonlinear finite element models are so far the only type of models whose accuracy of crashworthiness performance estimation is deemed acceptable by designers, prior to final design verifications involving prototype building and crash testing. The computational resources for the detailed FE models are enormous, and continue to be so (as computers become faster, the FE models become more detailed). Conventional design optimization methods for structural crashworthiness typically involve some adaptation of a conventional optimization algorithm and combining it with a conventional RSM model that is constructed on a pre-set sample of designs. The conventional methods have their limitations set by the availability of computational resources to construct and/or refine RSM models.

Another approach which is used in practice by vehicle designers is that of manipulation of the crash mode, or *crash mode matching*. The approach, which is more of an art rather than an algorithmic procedure, involves adjusting the design variables of the structure in order to achieve a desired deformation history. A notable work along these lines may be found in (Soto 2003), where topology optimization of one structural member is conducted to attain a desired time-deformation pattern. The concept of re-formulating the crashworthiness design problem into that of matching a desired deformation history seems otherwise overlooked in the literature. Hence, the thrust area for this dissertation is that of developing a formal algorithmic procedure for crash mode matching. It is conjectured that this approach, which incorporates knowledge of the physical behavior of the crash phenomenon into the design optimization process is to have its advantages in computational efficiency compared to the conventional methods.

CHAPTER 3

METHODOLOGY OVERVIEW

This chapter lays out the proposed methodology for crashworthiness design. A formal description of the scope of parametric structural crashworthiness optimization problems is presented, then an overview of the general steps in the proposed methodology is laid out.

3.1 Scope of Optimization Problems

The proposed methodology is intended for parametric optimization of structural crashworthiness problems. The problem formulation could be stated as:

Minimize: $f(\boldsymbol{x})$ (Equation 3.1)

Subject to: $\boldsymbol{g}(\boldsymbol{x}) \leq 0$ (Equation 3.2)

Where:

$\boldsymbol{x}$	is the vector of design variables x_i $i = 1, \ldots, nVar$.
f	is the objective function to be minimized. For multiple objectives, f is replaced by $\boldsymbol{f}$, which is a vector of objectives to be minimized.
$\boldsymbol{g}$	is the vector of inequality constraints g_i $i = 1, \ldots, nCon$. Constraints are set in the negative-null form.

It is also assumed that one can define $CM(\boldsymbol{x})$: the crash mode of the structure. $CM(\boldsymbol{x})$ is a matrix of time series, explained in detail later in chapters 4 and 5. While $CM(\boldsymbol{x})$ does not necessarily represent objectives or constraints, the proposed methodology requires the capability to evaluate it in order to guide the search algorithm.

Notes:

- It is assumed that evaluation of some of the objectives and/or constraints for some value of $\boldsymbol{x}$ requires computational simulation of the crash performance of the structure (or experimental evaluation). Typical quantities that are regarded as objectives and/or constraints include: displacement/deformation, acceleration and structural mass.

- The problem formulation does not include equality constraints. The reasoning behind this is as follows:
    - Constraints in crashworthiness problems are generally one-sided. For example, it may be unacceptable to exceed a target value of maximum acceleration, deformation, or injury criteria. However, lower values of maximum acceleration, deformation or risk of injury are not harmful.
    - The crash phenomenon often exhibits noise (whether true measurement noise in an actual crash test or numerical integration errors in a FE model) in some of the measurable quantities such as acceleration. Constraining a measurable quantity to an exact single target value is impractical and can result in making the entire design space infeasible.

- The design variables may be discrete, continuous, or mixed. Typical studies include design variables that correspond to the dimensions or materials of structural components. However, the proposed methodology requires fixed topology, so the design variables don't correspond to topological variations.

3.2 Overview of the Proposed Methodology

The proposed methodology follows along the lines of a typical design methodology for crashworthiness optimization, with some additions and variations on some steps. Section 3.2.1 provides an overview of the steps in a *typical* design methodology in research, then section 3.2.2 highlights the variations in the proposed methodology.

3.2.1 Typical Steps of Parametric Crashworthiness Design

Typical steps of parametric crashworthiness design are shown in Fig. 3.1. The first step in any design optimization methodology is to construct a reliable model for performance prediction of design changes. Parametric structural crashworthiness optimization proceeds from fixed structural topology that has been determined via topology optimization and vehicle styling. The construction of a detailed FE model is the usual practice at this stage. The FE model serves for design performance prediction throughout the optimization process. The next step is the translation of performance requirements (such as acceleration, deformation and injury criteria) into objectives and constraints. Creation of measurement points is done in the FE model to ensure proper recording of the objectives and constraints during a FE simulation run.

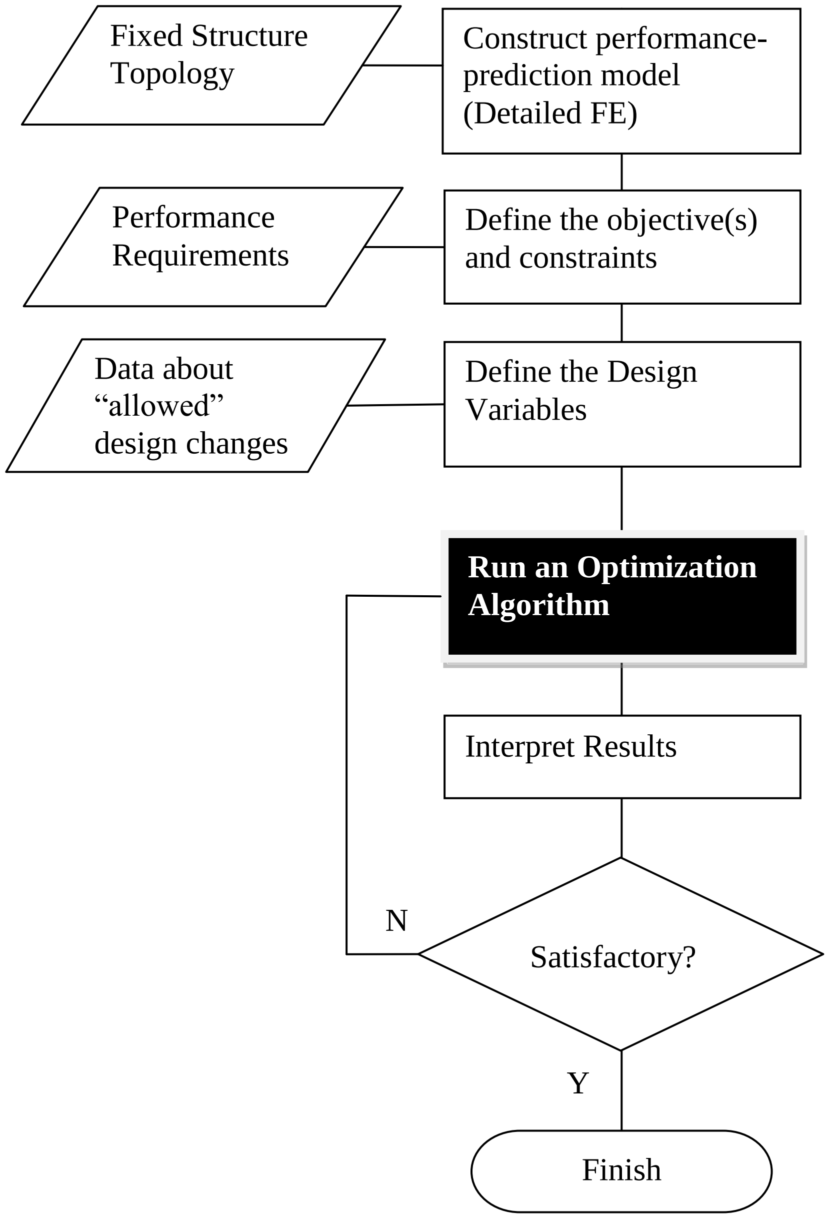

Fixed Structure Topology
Construct performance-prediction model (Detailed FE)
Performance Requirements
Define the objective(s) and constraints
Data about "allowed" design changes
Define the Design Variables
Run an Optimization Algorithm
Interpret Results
N
Satisfactory?
Y
Finish

Figure 3.1. Typical steps for parametric crashworthiness design

The next step usually involves some designer experience, as it is the translation of the existing data about the structural components' dimensions and materials into design variables. A designer may choose to limit the number of optional choices for some components, or fix some of them as constants in order to avoid having too many design variables and too large a design search space, which is not good for any optimization algorithm. On the other hand, too few variables or choices per variable may not allow the optimization algorithm to find possibly good designs because they are not in the search space. Once the design variables are decided, it is usual practice to add linking scripts to the FE model so that design variable changes can be automatically updated in the FE model.

With the FE model ready and the objectives, constraints and design variables setup; the designer uses available computational resources to run an optimization algorithm. The discussion in this section is generic, so the algorithm in question may be any of the generic design optimization algorithms. The algorithm is drawn as a black box in Fig. 3.1 to emphasize that once started, the human user has little or no control on the designs that the algorithm would recommend for exploration. The nature of the crashworthiness problem in a vehicle structure of a realistic level of complexity makes it impossible to guarantee optimality. However, most algorithms that would be used for such a problem have better chance of discovering good designs when allowed to perform many design space samples. The number of samples is often limited by the availability of computational resources to run as many FE simulations.

It is worth noting that the success of the typical approach is often dependent on how well the designer made use of his "non-algorithmic" skills in terms of proper choice of design variables, their ranges and the optimization algorithm to use. Too few or too restricted ranges on the design variables may exclude possibly good designs from the search space the optimization algorithm examines. Too many design variables/choices may cause failure of the optimization algorithm to discover any good design within a resource-wise-feasible number of design samples. Realistically speaking, there could be an outer loop (not shown in Fig. 3.1), where in the case of failure to discover satisfactory results after several attempted optimization runs, the designer might have to use a different optimization algorithm, or modify the definitions of the design variables.

The proposed methodology presented in the next section incorporates more of the designers' knowledge about crashworthiness within the methodology. It is conjectured that such knowledge would allow the optimization algorithm (the part where the designer has little or no control on) to discover good designs while using a smaller number of design samples. This is equivalent to more exploration, hence better likelihood to discover better designs using comparable amount of computational resources.

3.2.2 Proposed Methodology for Crashworthiness Design

The proposed methodology (Fig. 3.2) follows the same steps as typical methodologies in the construction of the performance prediction model (detailed FE) and the definition of objectives, constraints and design variables. However, there is more information from the designers' knowledge-base, or design space exploration via simplified reduced order dynamic models, that is incorporated into the methodology. The pivot of such additional knowledge is that of the crash modes.

The concept of the crash mode (CM) is qualitatively understood by vehicle designers as the “time history of deformation in various structural zones”. The CM is often like a “fingerprint” of a design and dictates whether it would exhibit high quality performance (in terms of the objectives and constraints) or not. Conscientious is that good designs have good CM and bad designs have bad CM, and conversely, designs with good CM are good designs, while designs with bad CM are bad designs.

In analogy to the definition of the design variables, the designer also needs to define the crash mode for the constructed FE model (definition of the CM, both qualitatively and quantitatively are discussed in detail in Chapters 4 and 5). The designer also needs to define what would be a desirable crash mode as well as a set of generic design adjustment rules for crash mode matching. The adjustment rules are analogous to an expert-system but are fairly simple to construct for the automated crash mode matching algorithm presented in this thesis, which is discussed in detail in Chapter 6.

The automated crash mode matching algorithm performs the role of an optimization algorithm in a typical structural crashworthiness optimization methodology; which is the automated sampling on the design space in order to discover good designs. However, it is hypnotized that the proposed methodology setup (incorporating knowledge base) would allow the proposed algorithm to have more success than a generic optimization algorithm at discovering high quality designs while utilizing reasonable computational resources. Examples and case studies presented in this dissertation serve to support the hypothesis.

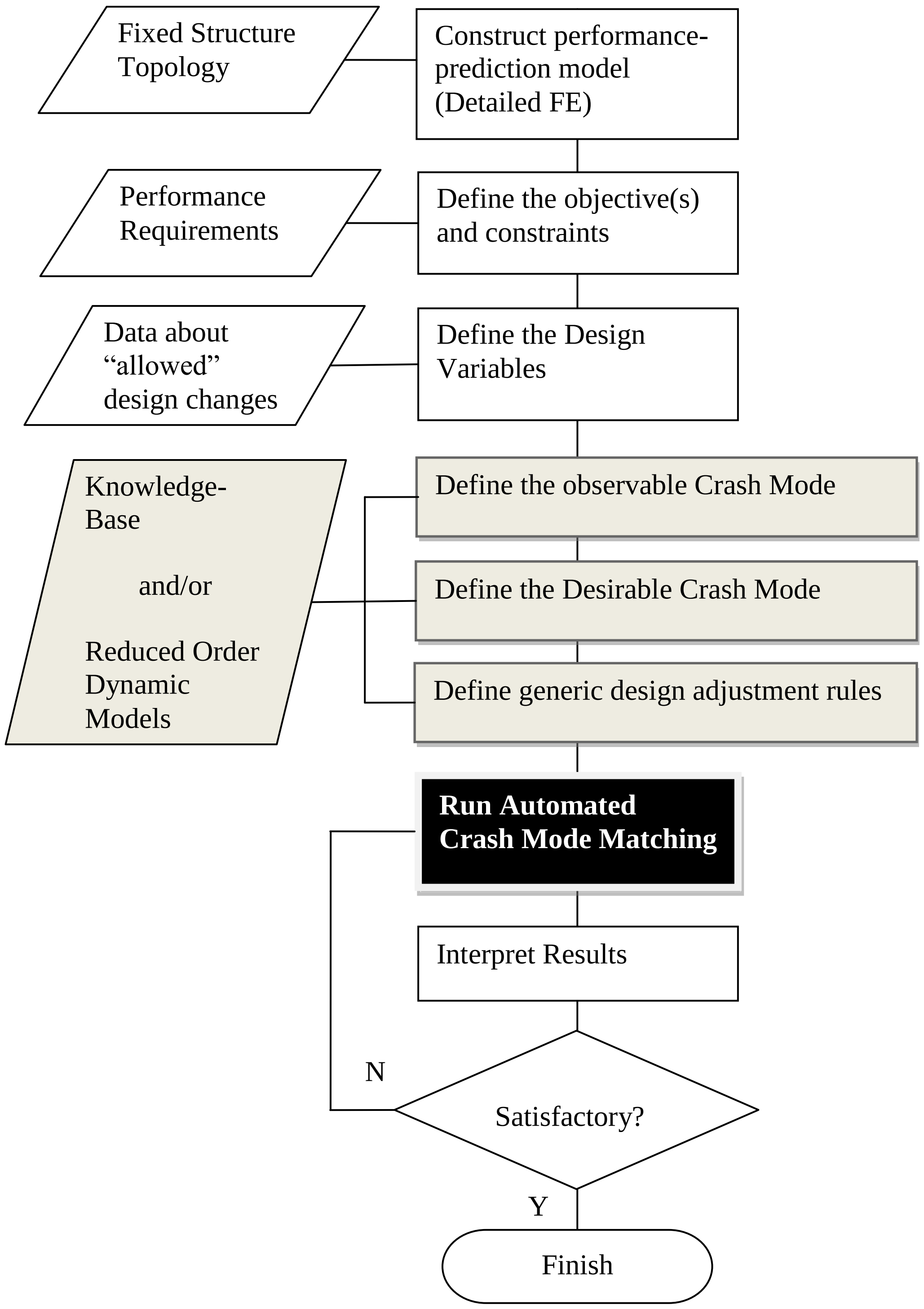

Fixed Structure Topology
Construct performance-prediction model (Detailed FE)
Performance Requirements
Define the objective(s) and constraints
Data about "allowed" design changes
Define the Design Variables
Knowledge-Base and/or Reduced Order Dynamic Models
Define the observable Crash Mode
Define the Desirable Crash Mode
Define generic design adjustment rules
Run Automated Crash Mode Matching
Interpret Results
N
Satisfactory?
Y
Finish

Figure 3.2. Steps in the Proposed Methodology for crashworthiness design

3.3 Summary

This chapter presented the scope of optimization problems addressed in this thesis and a high level overview of the proposed methodology. The proposed methodology enhances the existing methodology by incorporating the concept of crash modes and an automated crash mode matching algorithm. The crash modes are discussed in detail in chapters 4 and 5, while the automated crash mode matching algorithm is discussed in detail in chapter 6.

CHAPTER 4

QUALITATIVE EXPLORATION OF CRASH MODES

This chapter presents two example problems of structures that are subjected to crash conditions. The examples are simple enough to thoroughly analyze via solving multiple instances of the problem in order to explore all the possible crash modes for the said structures. The examples serve to support the hypothesis that crash modes could be useful in guiding an optimization search. The chapter proceeds with a discussion of approaches for exploration and discovery of desirable crash modes for problems of realistic complexity. One such crash mode exploration method (developed as a utility tool) is presented in detail.

4.1 Examples of Crash Modes in Structures

4.1.1 Two-Mass-Springs Problem

This section presents a simple structure that demonstrates how the crash mode can significantly influence the crashworthiness performance. The example (Fig. 4.1) portrays a vastly simplified situation of a payload (m_1) that crashes at an initial speed (v_o) onto a wall, with a front deformable structure ahead of it, represented by one mass (m_2) and two nonlinear springs (k_1 , k_2). Both springs behave in an idealized manner that corresponds to a deforming structure during crash (Fig. 4.2), where there is an initial linear spring behavior, followed by a collapse to a steady constant force behavior. This load-deformation pattern is an idealization of the typical behavior of axial crushing of a thin walled box-section (Han and Yamada 2000, Koanti and Kaliskan 2001). Parameters for each spring are as follows:

F_p	is the peak force of the spring
F_s	is the steady force of the spring after the peak
d_p	is the spring displacement at which the peak force of the spring occurs
d_s	is the maximum spring displacement, beyond which, the spring becomes a rigid object

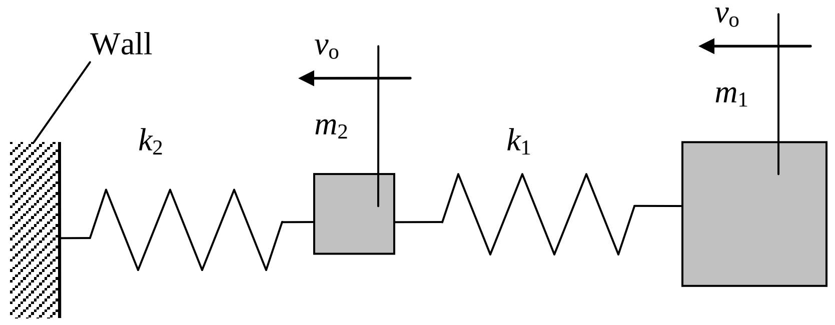


Figure 4.1. Two-Mass-Springs Problem

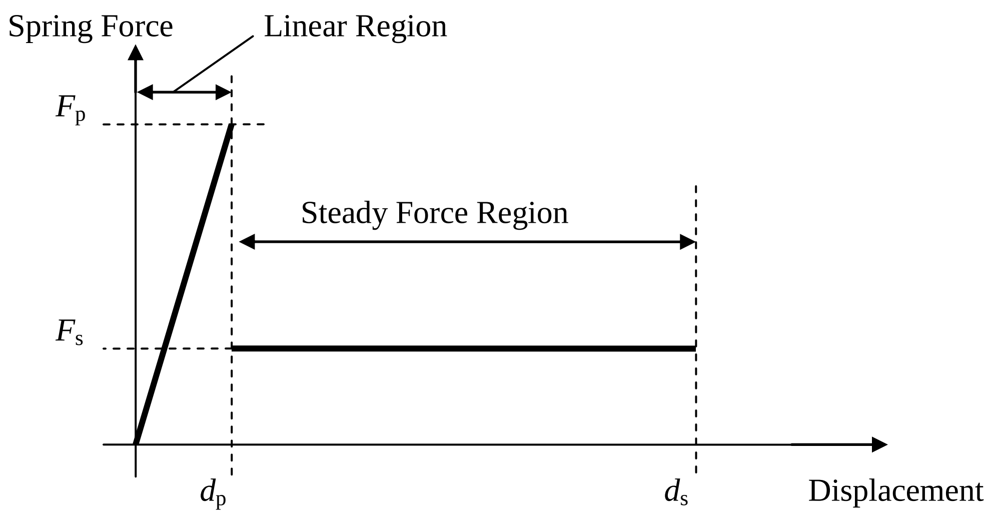


Figure 4.2. Crush behavior of the nonlinear springs

The total energy a spring can absorb is calculated as:

$$E_i = 0.5\, F_{pi}\, d_{pi} + F_{si}\, (d_{si} - d_{pi}) \qquad i = 1, 2 \qquad \text{(Equation 4.1)}$$

The parameter values used in this example are listed in Table 4.1. The setting of absorbable amounts of energy in each spring ensures that the system can stop (or bounce back). It is noted in this study that only one parameter ($r = E_2 / E_1$) is an implicit variable that is allowed to change. r is the ratio of the total energy that may be absorbed in each spring. In essence, this ratio represents the relative strength between the two parts of the deformable structure.

Table 4.1. Parameter values for the two-mass-spring example

Symbol	Description	Value	Unit
m_1	Payload mass	50	kg
m_2 / m_1	Ratio of masses	0.02	
v_o	Initial velocity for both masses	10.0	m/s
F_{p1} / F_{s1}	Peak value to steady value in 1st spring	3.0	
F_{p2} / F_{s2}	Peak value to steady value in 2nd spring	3.0	
d_{p1} / d_{s1}	Peak displacement to maximum displacement in 1st spring	0.02	
d_{p2} / d_{s2}	Peak displacement to maximum displacement in 2nd spring	0.02	
d_{s1}	Maximum displacement in 1st spring	0.25	m
d_{s2}	Maximum displacement in 2nd spring	0.25	m
$E_1 + E_2$	Total absorbable energy	1.2 × 0.5 ($m_1 + m_2$) v_o^2	
$r = E_2 / E_1$	Ratio of absorbable energy in springs	(variable)	

This example is simple enough to allow for analytical expression of the differential equations of motion, based on the various possible situations of the nonlinear springs. The equations of motion for all possible case are:

- Case when both springs are in the linear region:

$$\begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix} \begin{Bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{Bmatrix} + \begin{bmatrix} F_{p1}/d_{p1} & -F_{p1}/d_{p1} \\ -F_{p1}/d_{p1} & F_{p1}/d_{p1} + F_{p2}/d_{p2} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} \qquad \text{(Equation 4.2)}$$

- Case when 1st spring is in the linear region, 2nd spring in steady force region:

$$\begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix} \begin{Bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{Bmatrix} + \begin{bmatrix} F_{p1}/d_{p1} & -F_{p1}/d_{p1} \\ -F_{p1}/d_{p1} & F_{p1}/d_{p1} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ -sign(\dot{x}_2)F_{s2} \end{Bmatrix} \quad \text{(Equation 4.3)}$$

- Case when 2nd spring is in the linear region, 1st spring in steady force region:

$$\begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix} \begin{Bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{Bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & F_{p2}/d_{p2} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} -sign(\dot{x}_1 - \dot{x}_2)F_{s1} \\ sign(\dot{x}_1 - \dot{x}_2)F_{s1} \end{Bmatrix} \quad \text{(Equation 4.4)}$$

- Case when 1st spring is in the linear region, 2nd spring is totally compressed (assumed full plastic collision against wall):

$$m_1\ddot{x}_1 + (F_{p1}/d_{p1})x_1 = 0 \quad \text{(Equation 4.5)}$$

- Case when 2nd spring is in the linear region, 1st spring is totally compressed (assumed full plastic collision between the two masses):

$$(m_1 + m_2)\ddot{x}_1 + (F_{p2}/d_{p2})x_1 = 0 \quad \text{(Equation 4.6)}$$

- Case when both springs are in the steady force region:

$$\begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix} \begin{Bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{Bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} -sign(\dot{x}_1 - \dot{x}_2)F_{s1} \\ sign(\dot{x}_1 - \dot{x}_2)F_{s1} - sign(\dot{x}_2)F_{s2} \end{Bmatrix}$$

(Equation 4.7)

- Case when 1st spring is in the steady force region, 2nd spring is consumed (assumed full plastic collision against wall):

$$m_1\ddot{x}_1 = -sign(\dot{x}_1)F_{s1} \qquad \text{(Equation 4.8)}$$

- Case when 2nd spring is in the steady force region, 1st spring is consumed (assumed full plastic collision between the two masses):

$$(m_1 + m_2)\ddot{x}_1 = -sign(\dot{x}_1)F_{s2} \qquad \text{(Equation 4.9)}$$

A computer program is written via the C++ programming language to solve the equations (4.2 – 4.9), given the initial conditions ($\dot{x}_1 = \dot{x}_2 = v_o, x_1 = x_2 = 0$) for a value of *r*. Since the total absorbable energy (E_1 + E_2) is selected as a constant value (Table 4.1), setting a value of *r* allows calculation of both E_1 and E_2. Equation (4.1) is then used to calculate the peak and steady values of the nonlinear springs (F_{p1}, F_{s1}, F_{p2}, F_{s2}). Numerical integration is performed to obtain the time response of the system via simple logic in the C++ program, which checks the displacement state in each spring, and accordingly selects the correct governing differential equation among equations (4.2 – 4.9). The displacements and velocities of the two masses are then fed as initial conditions for the new governing equation.

It can be shown that varying the ratio *r* for this example problem produces two possible crash modes:

- Crash Mode #1: 1st spring exceeds peak force first, which happens at high values of *r* when spring #2 is stronger. This crash mode is sketched in Fig. 4.3(a).

- Crash Mode # 2: 2nd spring exceeds peak first, which happens at low values of (*r*) when spring #1 is stronger. This crash mode is sketched in Fig. 4.3(b)

Deformation plots of both springs for various values of the strength ratio *r* are shown in Fig. 4.4 and Fig. 4.5 respectively.

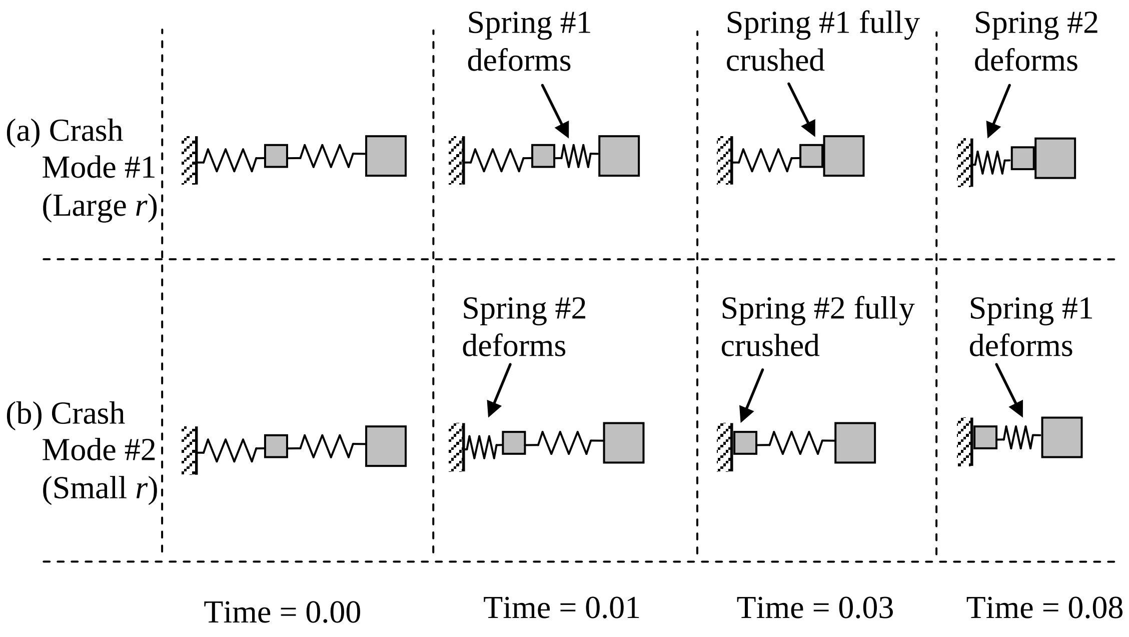


Figure 4.3. Sketching of the Crash Modes in the Two-Mass-Springs Problem: (a) Crash Mode #1, (b) Crash Mode #2

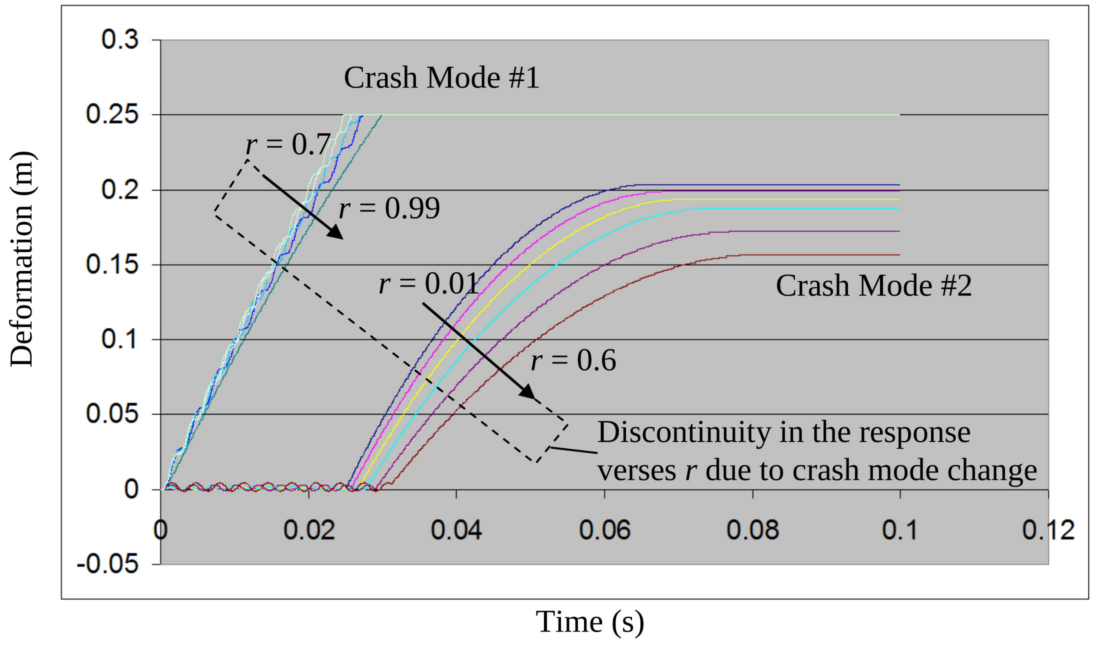


Figure 4.4. Deformation plot for the 1st spring

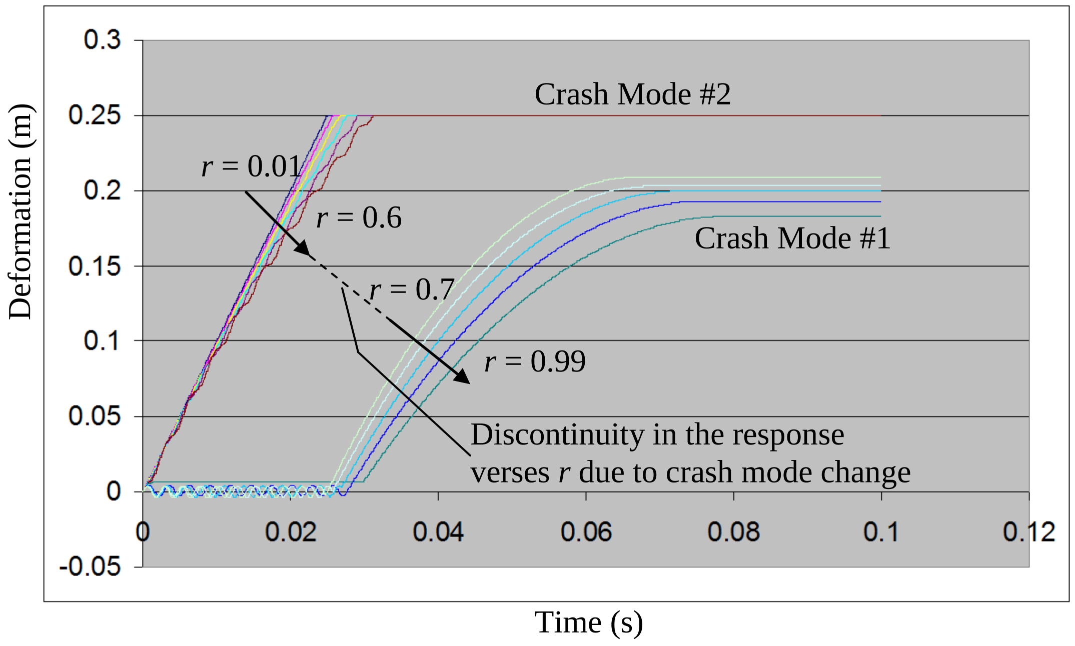


Figure 4.5. Deformation plot for the 2nd spring

Assuming that the objective of designing the springs is to minimize the maximum acceleration experienced at the payload mass, this maximum value is recorded in the C++ solver program. A plot of the maximum acceleration verses the relative strength of the two springs (*r*) is shown in Fig. 4.6.

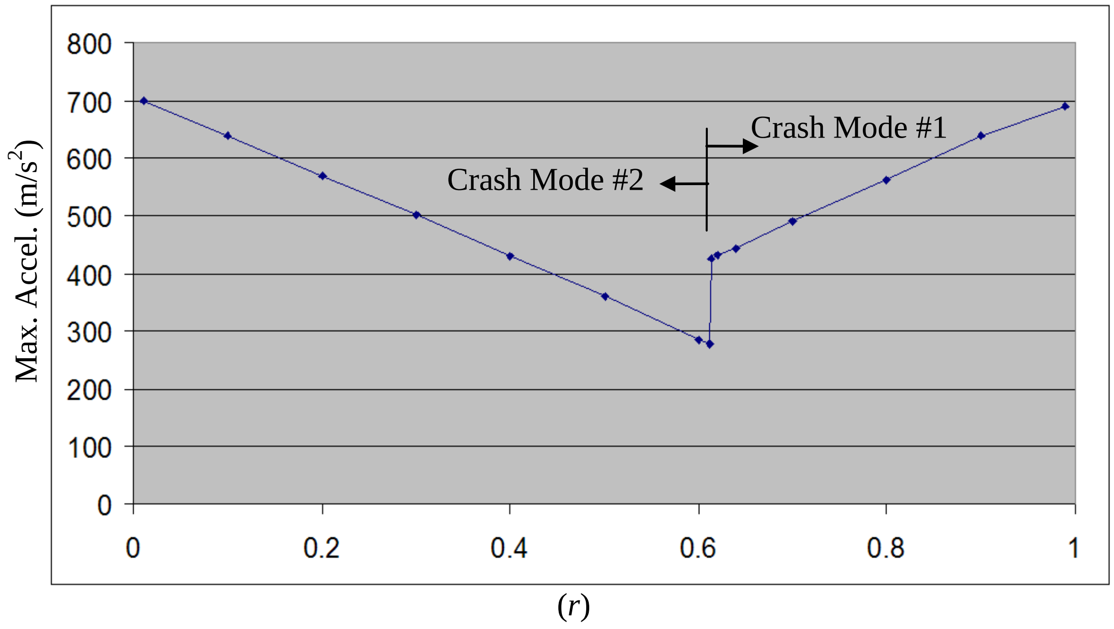


Figure 4.6. Maximum acceleration for the payload mass (*m*$_1$)

It is intuitively understood that extremely low or high values of *r* are undesirable, because the weaker spring will be crushed first and reach the steady force region while providing low deceleration of the payload mass. This will evidently result in the payload mass hitting the stronger spring at higher velocity after the weaker spring is fully compressed, causing higher maximum acceleration to be experienced by the payload. Counter-intuitively, however, the least attainable maximum acceleration is *not* achieved at a perfectly balanced strength ratio (*r* = 0.5). The least attainable maximum acceleration is achieved around the region of (*r* = 0.6). This is due to the dynamic effects of the masses which maintain CM#2 (2nd spring deforms before the 1st spring), even though if 2nd spring is slightly stronger. The 1st spring then serves as a softer cushion for the payload mass resulting in overall lower maximum acceleration. Increasing *r* beyond a certain value (around *r* = 0.61) causes a change of the crash mode to CM#1 (1st spring crushes first), which is accompanied by a significant change in the crashworthiness performance (large undesirable increase in the maximum acceleration).

This simple example serves to highlight the concept of a crash mode, as well as its effect in dictating the crashworthiness performance. A more complex example is provided in the next section.

4.1.2 Vehicle Mid-Rail Problem

This section further demonstrates the effect of crash modes on the crashworthiness performance via a more involved example of a vehicle mid-rail shown in Fig. 4.7. The mid rail is assumed to have a uniform rectangular box cross-section throughout its entire length. The box is assumed to be made up of mild steel sheet. Parameters for the model are listed in Table 4.2. The problem has one independent variable (t_1), which is the thickness of the sheet metal in zone #1 (Fig. 4.7), and one dependent variable (t_2), which is the thickness of the sheet metal in zone #2, and is calculated in accordance to (t_1) to maintain a constant structural mass of the rail. It is desirable to explore the region of values for the design variable that result in the minimal deformation in zone #2 (closer to the passenger compartment) as the rail deforms to absorb the kinetic energy in the payload (M). Zone deformation is calculated simply as the difference in position along the x-axis direction between the zone ends.

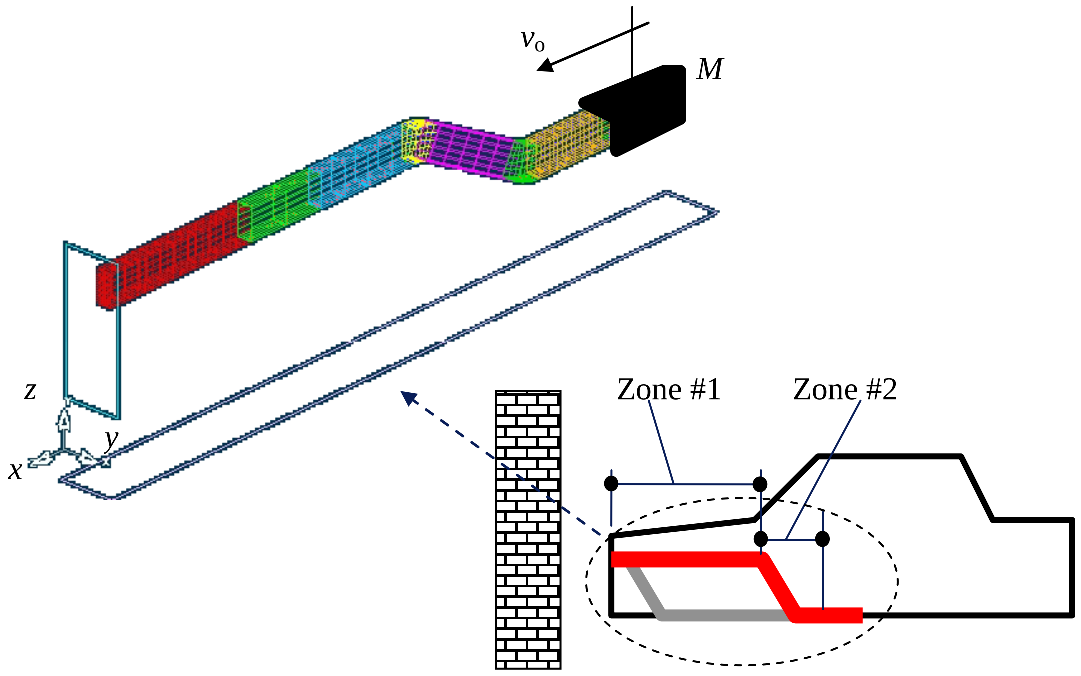


Figure 4.7. Vehicle Mid Rail Model

A finite element model (FE) is constructed for the mid rail using the LS-DYNA commercial software (LSTC 2001). The model response is simulated for several test values of (t_1). Since FE simulations of crashworthiness problems often exhibit noise, as well as undesirable sensitivity to meshing variations, each simulation is performed five times with slight (sub 0.1mm) randomization of the node positions in the mesh. The reported values represent the average response of the five runs.

Table 4.2. Parameter values for the Vehicle Mid Rail example

Symbol	Description	Value	Unit
M	Payload mass	200.0	kg
v_o	Initial velocity	10.0	m/s
L_1	Length of Zone #1	650.0	mm
L_2	Length of Zone #2	375.0	mm
h	Box section height	90.0	mm
b	Box section width	60.0	mm
ϕ	Inclination angle	23.5	deg
d	Z-direction rail axis offset	120.0	mm
t_1	Sheet thickness in Zone #1	(Variable)	mm
t_2	Sheet thickness in Zone #2	$5.6 – 1.8\ t_1$	mm

Summary of the deformation in Zone #2 as function of (t_1) is shown in Fig. 4.8. Three different regions of behavior of the deformation in zone #2 verses t_1 are identified in plot in Fig. 4.8. Examining the animation of the deformation history for designs in the different regions leads to the identification of three crash modes, which are are sketched in Fig. 4.9:

- Crash mode #1 (Fig. 4.9a) occurs at small values of (t_1) and is characterized by a crush (axial deformation) in zone #1, but the deformation is not sufficient to absorb all the crash energy, so it is followed by a deformation in zone #2 in the form of two plastic hinges

- Crash mode #2 (Fig. 4.9b) occurs at some mid-range values of (t_1), where the axial deformation in zone #1 is sufficient to absorb most of the crash energy and results in negligible deformation taking place in zone #2

- Crash mode #3 (Fig. 4.9c) occurs at large values of (t_1), where zone #1 is too strong and doesn't exhibit any appreciable plastic deformation, causing all the crash energy absorption to happen in zone #2 in the form of plastic hinging.

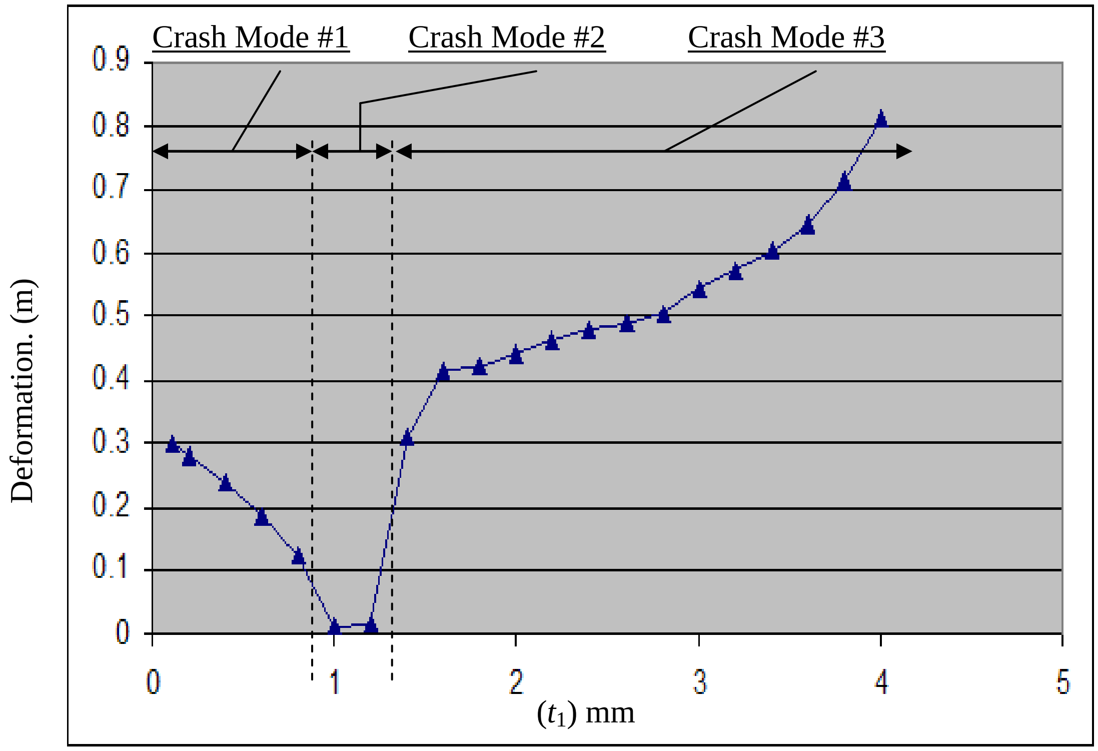


Figure 4.8. Deformation in Zone #2 as function of (t_1)

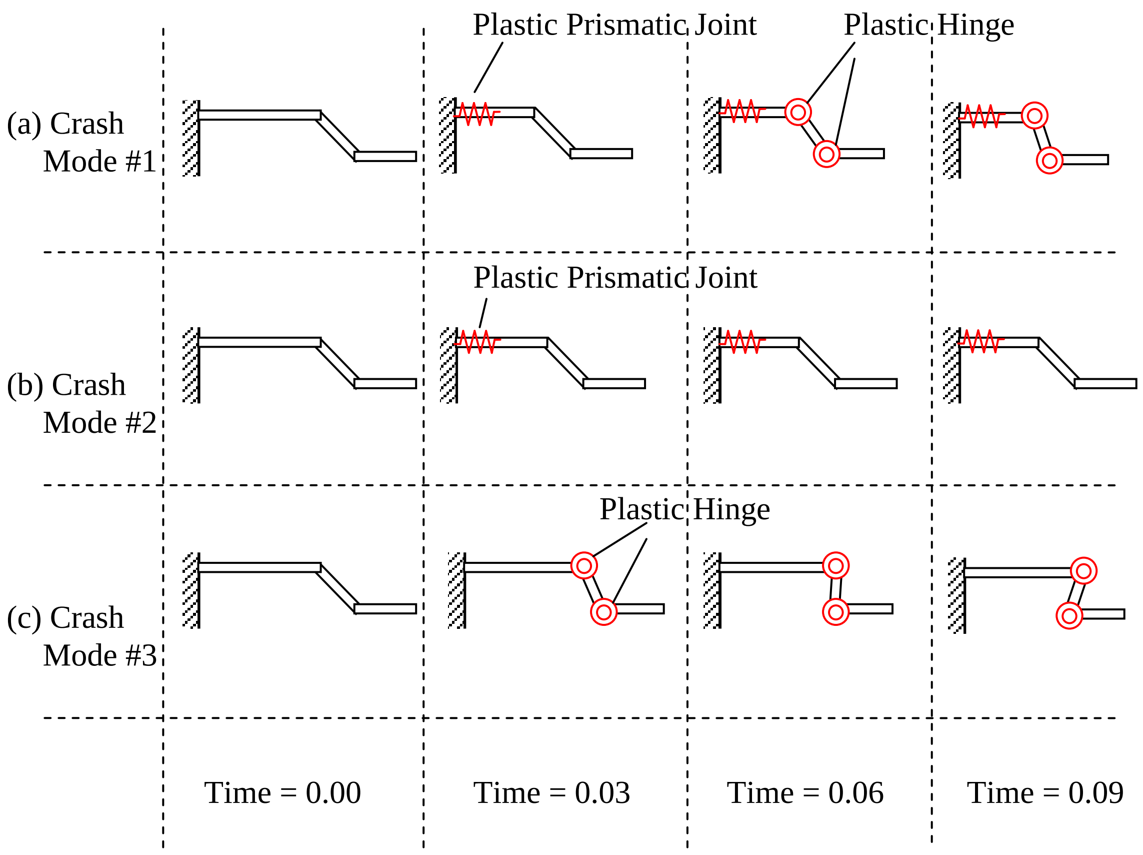


Figure 4.9. Sketching of the Crash Modes in the Vehicle Mid-Rail Problem

From a designer's perspective, CM #3 is the least desirable since it exhibits large deformation in zone #2 of the mid rail, while crash mode #2 is the most desirable since it results in the least deformation in zone #2. Achieving the desired crash mode for this problem requires correct adjustment of the design variable t_1 so that zone #1 is neither too weak (otherwise it doesn't absorb all the crash energy, as in CM #1), nor too strong (otherwise it doesn't deform at all and absorbs no crash energy, as in CM #3).

This example serves to further highlight the importance of crash modes in dictating crashworthiness performance of structures. It demonstrates how changing a design variable too much in what might seem to be a beneficial direction can result in a switch of the crash mode from a good one to a bad one. The switch in the crash mode from good to bad also changes a good design into a bad one. A listing of the hypotheses made regarding crash modes in this thesis is provided in the next section.

4.2 Hypotheses of Crash Modes

Based on the observations from the two examples presented in sections 4.1.1 and 4.1.2, as well as discussions with vehicle designers in practice, the following are the main hypotheses of crash modes:

- Distinctive and drastic changes in the crashworthiness performance (such as deformation or acceleration) happen upon the change of crash modes. This was demonstrated in both examples at the transitions between different crash modes (Fig. 4.6 at $r \cong 0.6$ and Fig. 4.8 at $t_1 \cong 0.9$ and $t_1 \cong 1.3$)
- Designs exhibiting the same crash mode tend to have similar performance in terms of objective functions. Furthermore, the trends in the objective functions versus the design variables seem to maintain their monotonicity within the same crash mode. For example, in Fig. 4.6, increasing the value of r seems to benefit the objective *as long as* crash mode #1 is in effect, but this trend changes when CM #2 is in effect. Also, in Fig. 4.8,

increasing t_1 is beneficial while CM #1 is in effect, but the opposite happens when CM #3 is in effect.

- Crash modes provide a partitioning of the design domain. Thus, attaining the desired crash mode implies a high likelihood of being in the vicinity of an optimal design.

Due to the complexity of crashworthiness models involving vehicle structural models of realistic level of detail, it is perceived impossible to *prove* these hypotheses. The validity of the hypotheses however are empirically demonstrated in the simple examples presented in this chapter, and validated in the case studies of realistic vehicle models in Chapter 7.

4.3 Exploration of Crash Modes of a Vehicle Structure

4.3.1 Options for Crash Modes Exploration

The proposed methodology essentially guides the crashworthiness design optimization via biasing the search to favor desirable crash modes. As such, knowledge of the desired crash mode is a requirement to begin with, before the proposed method can be applied. It is important to note that in many cases of full vehicle structures, such desired crash mode is already “mostly known” from previous experience with older vehicle models or similar structures. In the case of completely new conceptual designs however, the luxury of prior knowledge about the desired crash mode could be unavailable, thus requiring a study in itself. The options for exploring vehicle structural crash modes are discussed in this section.

In the absence of prior knowledge about similar designs, discovery of the desirable crash mode (needed in the proposed methodology) could be as much of a challenging task as the optimization problem via conventional methodology. In the examples presented in section 4.1, exploration of the crash modes was performed via exhaustive search, *i.e.* sampling of a dense grid over all possible values for the design variables. Exhaustive search is only feasible in the simplest of problems, due to the exponential growth in the size of the design space with respect to the number of design variables. Exhaustive search is infeasible for any crashworthiness problem of a realistic level of complexity. Feasible approaches to discover the desirable crash mode are:

- Partial design space sampling via design of experiments (DOE) (Taguchi, G., 1993, Phadke, 1989). In this approach, a systematically well spread out number of sample designs are explored in the design space. A designer applying this approach would hope that one of the samples would be in the vicinity of an optimal design, and hence exhibit the desirable crash mode. The effectiveness of partial sampling however is diminished in problems involving a large number of design variables.

- Performing an optimization run via conventional methodology. Since the crash mode hypotheses state that the optimum design has the optimum crash mode; discovery of the optimum design is also discovery of the optimum crash mode.

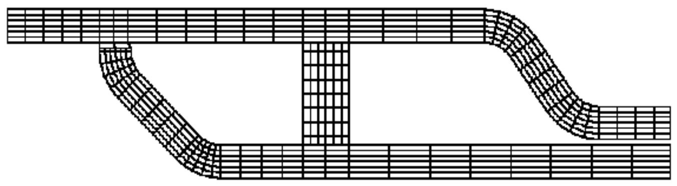
Discovery of the desirable crash mode in the absence of prior knowledge about it might seem like vicious cycle in the logical flow of the proposed methodology. The desirable crash mode is needed to discover the optimum design, yet the optimum design is needed to discover the desirable crash mode. However, one could very well argue that if the good crash model for similar vehicle designs is generic enough to carry over from prior knowledge to a new design, then the good crash mode can carry over from a simplified model of the vehicle structure to the detailed FE model.

The argument of the generality of carry-over of good crash mode across models alleviates the restraint on using simplified crashworthiness models. Simplified models may not be suitable for direct use in optimization (due to inaccurate predictions for objectives and constraints), but they may be suitable for exploring the crash modes. Possible simplified models for exploring the crash modes are:

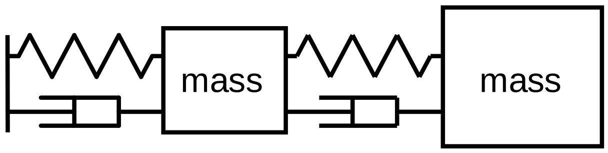
- Coarse-mesh finite element models: where reduction of computational resources required comes at a sacrifice of some accuracy of estimated results

- Coarse-grained reduced order dynamic models: where entire sections of the structure are lumped into equivalent masses and springs/dampers (Beneet *et. al.* 1991) (example sketch in Fig. 4.10b). Clearly the loss of physical detail of the structure is a hindering issue, aside from fidelity

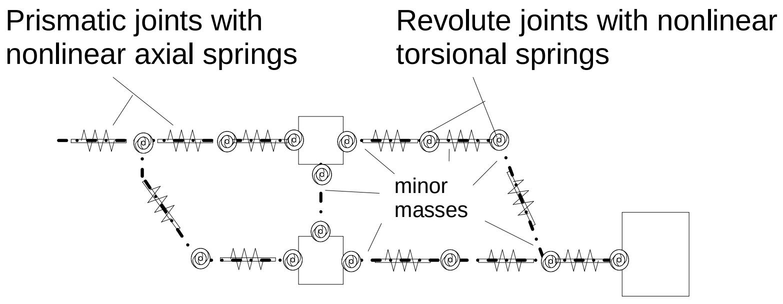
- Fine-grained reduced order dynamic models: where the important structural members are represented member-by-member as sets of masses and springs/dampers (Takada and Abramowicz, 2004, Hamza and Saitou 2005) (example sketch in Fig. 4.10c).While these models resemble coarse mesh FE techniques in the trade-off between accuracy and speed of computation, they are very suited for observing crash modes due to the implicit identification of the important structural members prior to the construction of such models



(a)



(b)


Prismatic joints with nonlinear axial springs
Revolute joints with nonlinear torsional springs
minor masses

(c)

Fig. 4.10. Different models of a vehicle structure: (a) a finite element model of a structure (b) lumped mass model and (c) equivalent mechanism model

Of the considered options for exploring the crash modes of a vehicle structure, the ones expected to be most effective are the coarse mesh finite element methods and the fine grained reduced order dynamic models. One class of fine-grained reduced order dynamic models was developed in this thesis, and called the Equivalent Mechanism (EM) models. The EM models allow for one-to-one correspondence in zone geometry definition and have shown effectiveness in modeling small to moderate sized vehicle structures (Hamza and Saitou 2003-2006). The EM models are explained in further detail in the next subsection.

4.3.2 Crash Modes Exploration via Equivalent Mechanism Models

This section discusses equivalent mechanism (EM) models as a fine-grained reduced order dynamic model that could be employed for structural crash modeling, with the purpose of discovering the desirable crash mode. An EM model (Fig. 4.10c) is a network of rigid beams joined by prismatic and revolute joints with special nonlinear springs. The springs are designed to mimic the force-displacement and moment-rotation characteristics of thin-walled beams often found in the body of vehicle structures. Mapping of the physical dimensions to their equivalent springs is performed by interpolation on pre-compiled databases of structural cross-sections (Hamza and Saitou 2004b); similar to the way civil engineers use standard steel-section tables. In essence, the EM models are *fine-grained* lumped mass models, similar to Abramowicz's Super Folding beam element (Abramowicz 2003, 2004), with the main difference being in the nonlinear springs equation and method of spring properties estimation.

The key assumptions for applying EM models to explore and discover crash modes are:

- The observable crash modes in detailed FE models *exist* in the constructed EM models
- Crashworthiness performance of EM models qualitatively corresponds to that of real structures. In other words, good crash modes in reality are good in EM models and bad ones are bad

Just as with most assumptions about the crash phenomena in vehicle models that have a realistic level of detail, the governing equations are so complex that the presented key assumptions are difficult to prove or disprove. However, some arguments could be drawn out to justify why the assumptions can be acceptable:

Existence of the crash modes in EM models:

- In essence, the crash mode of a real structure is a sequence of crash energy dissipation events. A good crash mode is one that allocates the appropriate amount of energy to the appropriate structural zone at the appropriate time. The EM models allow a one-to-one correspondence in zone geometry definition. Zones in the detailed FE model are typically defined as the collection of elements that form a structural member (e.g. bumper, rail, pillar). Thus, the EM models allow having one zone for every zone defined in the detailed FE. Thus, the crash mode observed in EM is a similar allocation of crash energy to structural zones as a crash mode that occurs in the detailed FE model.

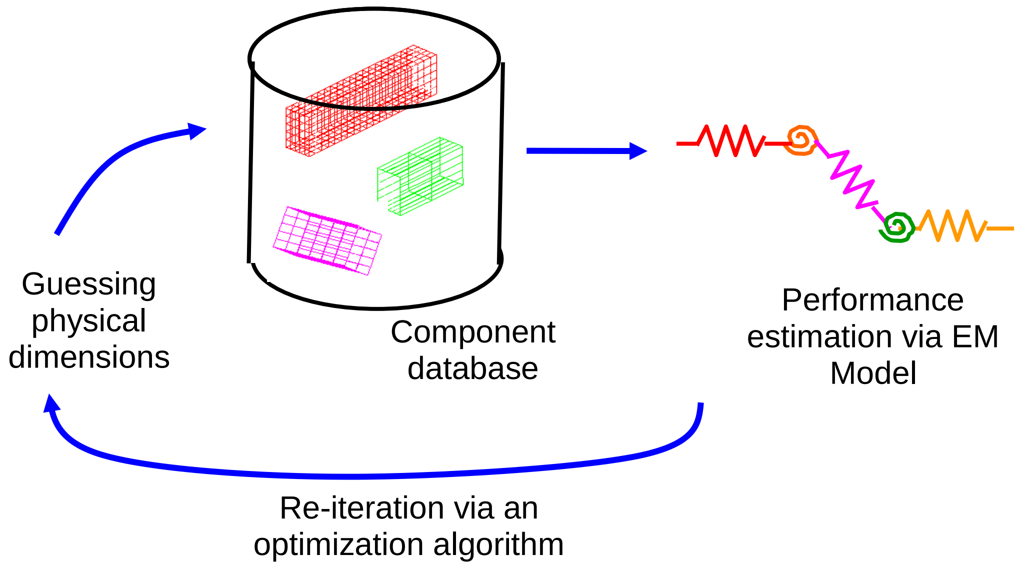

Guessing physical dimensions
Component database
Performance estimation via EM Model
Re-iteration via an optimization algorithm

Fig. 4.11. Procedure to estimate the desirable crash mode via EM models

Correspondence in performance of crash modes:

- Since the crashworthiness performance measures considered in this research are ultimately time histories of structural deformations (displacements and accelerations), having a bad deformation pattern in an EM model (for example: excessive deformation near passenger compartment) would be observed as a bad crash mode. Conversely, a good deformation pattern in EM models would be observed as a good crash mode.

The procedure to apply EM models in order to discover desirable crash modes is illustrated in Fig. 4.11. The procedure goes through an optimization run, where the optimization algorithm guesses values of the design variables. Next a *component* *database* is used to estimate the nonlinear springs in an EM model that corresponds to the detailed FE model. Use of a database of physical components to estimate the parameters of the nonlinear springs ensures that the nonlinear springs in the EM model always correspond to some physically realizable structure. Crashworthiness performance is then estimated via dynamic simulation of the EM model. The estimated performance is then returned to the optimization algorithm, and the optimization algorithm re-iterates. Genetic algorithms, being popular for a broad spectrum of problems (Goldberg 1989, Deb *et. al.* 2000) were implemented, although other algorithms could be applied within this framework. At the end of the optimization run, the main interest is the observed crash mode associated with having good crash performance, rather than the specific values of the design variables. Inaccuracy in the reduced order dynamic models may prevent using the optimum values of the design variables as seen in the reduced order dynamic model. However, it is hypothesized that the observed desirable crash mode can be carried over from the reduced order dynamic model to the detailed FE model, just as one would carry over a desirable crash mode from prior knowledge of similar designs to a new design.

4.3.3 Details of Equivalent Mechanism Models

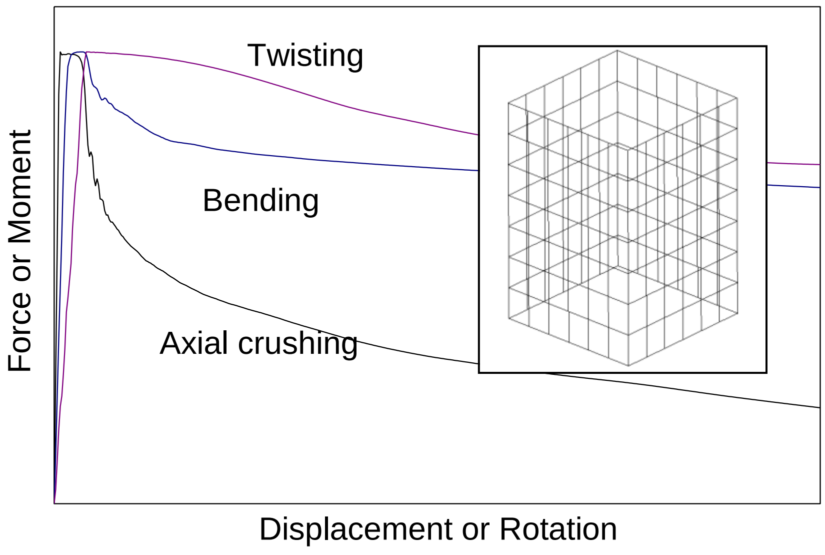
Implementation of EM models is fairly simple. The main structural members of the vehicle frame, which are typically modeled using plate or shell elements in FE models, are approximated as sets of rigid masses connected by prismatic and revolute joints that have special nonlinear springs (Fig. 4.10c). The deformation resistance behavior of the springs is chosen to capture the behavior of the structural members. The EM models are then solved using a conventional dynamic simulation algorithm, thereby providing an estimation of the overall vehicle structure behavior.

Fig. 4.12 shows typical deformation resistance curves of short thin-walled beams with box and hat sections, subject to axial twisting, transversal bending, and axial crushing (Hamza and Saitou 2004b), obtained using the LS-DYNA software. In both plots, the horizontal axis is displacement or rotation, while the vertical axis is the reaction force or moment. These curves were generated using the loading conditions sketched in Fig. 4.13, where one side is held fixed, while the other side is subjected to forced deformation of small uniform velocity (quasi-static). The recorded resistance to motion (axial force in case of axial crush, moments in the cases of bending and twisting) at the moving side is recorded into the component database for the given structural section shape and dimensions.

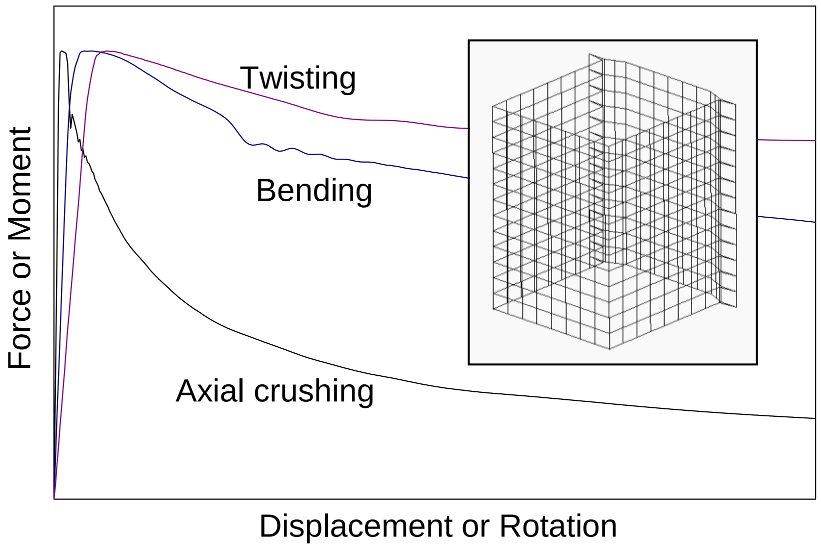
Observation over a wide range of dimensions for the cross-sections geometry and wall thickness, show the load-deformation curves bearing distinct similarities to Fig. 4.12 in:

1) The steep, linear rise in resistance for small deformation
2) The saturation at elasticity limit
3) The gradual drop to a steady-state resistance.

Experimental observations in the literature confirms this deformation pattern (Han and Yamada 2000, Koanti, R. P. and Caliskan 2001) as long as the considered members are short enough so that no multiple folds of the sheet metal are formed (which results in secondary peaks).



(a)



(b)

Fig. 4.12. Typical deformation resistance curves for (a) box section and (b) hat section.

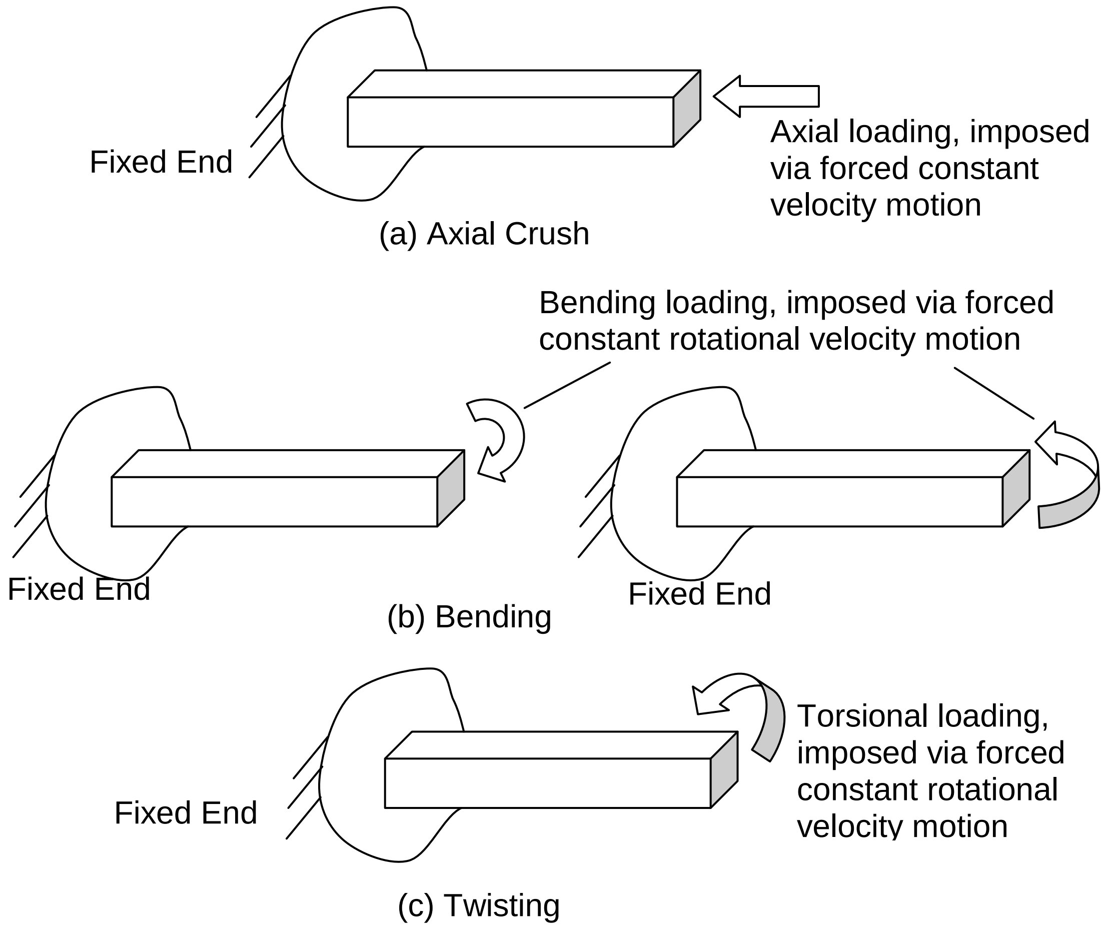

Fixed End
Axial loading, imposed via forced constant velocity motion
(a) Axial Crush
Bending loading, imposed via forced constant rotational velocity motion
Fixed End
Fixed End
(b) Bending
Fixed End
Torsional loading, imposed via forced constant rotational velocity motion
(c) Twisting

Fig. 4.13. Loading conditions for generating the component database curves for EM models. (a) Axial Crush, (b) Bending and (c) Twisting

The spring force (or moment) F_k within EM models is given as a sum of the forces corresponding to each of the four zones illustrated in Fig. 4.14, blended together using sigmoid functions (Mathworks 2001), which are a continuous version of the step function:

$$F_k = sig_1(F)_1 + sig_2(F_2) + sig_3(F_3) + sig_4(F_4) \qquad \text{(Equation 4.10)}$$

where:

$$F_1 = \frac{F_e}{\delta_e}\delta$$ (Equation 4.11)

$$F_2 = F_p - \frac{(F_p - F_e)}{(\delta_p - \delta_e)^2}(\delta_p - \delta)^2$$ (Equation 4.12)

$$F_3 = F_s + (F_p - F_s)e^{\frac{-4}{(\delta_s - \delta_p)}(\delta - \delta_p)}$$ (Equation 4.13)

$$F_4 = F_s + \frac{F_e}{\delta_e}(\delta - L_c)$$ (Equation 4.15)

- δ is the instantaneous amount of deformation, referenced to the un-deformed length of the spring.
- F_e is the maximum elastic force (or moment).
- δ_e is the maximum elastic deformation occurring at the transition between zones 1 to 2.
- F_p is the peak deformation resistance force.
- δ_p is the deformation at which the peak deformation resistance occurs at the transition from zones 2 to 3 (Fig. 4.14).
- F_s is the steady state resistance force after collapse.
- δ_s is the deformation at which the resistance falls within 2% of the steady state value.
- L_c is the maximum deformable length (or angle) occurring at the transition from zones 3 to 4 (Fig. 4.14).

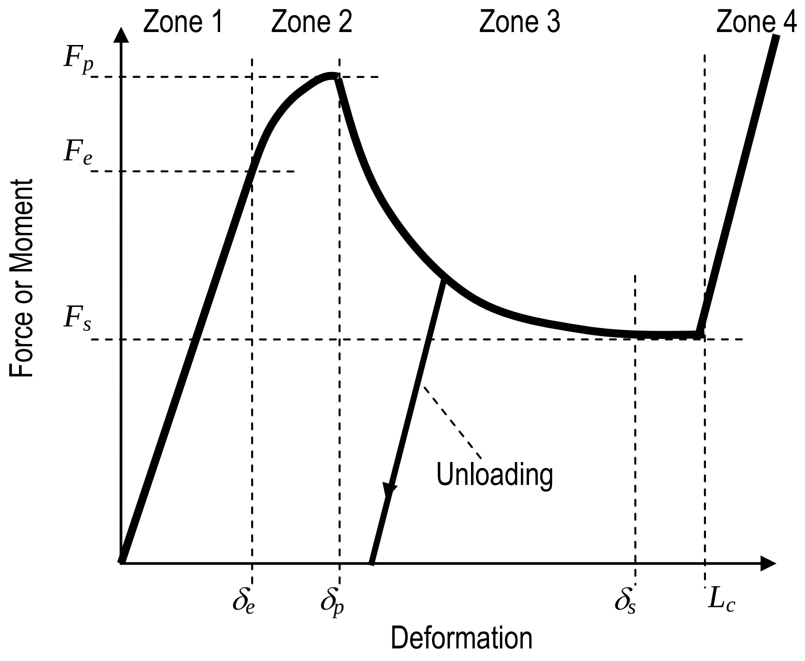


Fig. 4.14. EM nonlinear spring behavior and main curve parameters

In Fig. 4.14, zone 4 represents the high stiffness after crushing the full deformable length. The behavior during unloading is assumed to go parallel to the elastic zone starting from the maximum deformation that had occurred. This unloading regime mimics the energy loss due to plastic deformation and removes the need to include explicit dampers in the EM model.

The maximum deformable length L_c is estimated from the length, geometry and connectivity of the represented structural member. For example, in a crush module of 100mm length modeled using 4 springs in series, the total deformable length L_c for each spring would be 100 / 4 = 25mm.

The estimation of the other tuning parameters is done by referring to component databases of pre-analyzed FE models of the short, thin-walled beams with different sizes of box and hat sections and wall thicknesses. A different set of the tuning parameters (F_e, δ_e, F_p, δ_p, F_s, and δ_s) are identified for different directions of deformation of the nonlinear spring, in order to better represent the difference in behavior between tension and compression, as well as bending in un-symmetric sections.

The task of generating the component database is quite elaborate, however, once generated for a family of cross-sections, they can be used in any EM model much like a civil engineer uses steel-section tables. The current implementation of EM models only has Box (Fig. 4.12(a)) and Hat (Fig. 4.12(b)) sections generated (Hamza and Saitou 2004b). The steps for generating a component database are listed as follows:

1. Construct a base FE model of the cross-section in question. The FE model should be able to get automatically modified by controlling the cross-section geometric variables such as dimensions and sheet thickness(es). The model would also be setup to run the loading cases corresponding to axial, bending (along 2 main axes) and twisting (Fig. 4.13)
2. Construct a grid of data points that span the ranges of the cross-section geometric variables values
3. Run the FE-model for the axial, bending and twisting loading. Record the load-deformation curves.
4. Repeat step #3 a number of times with small-scale randomization (less than 0.1mm) on the node positions of the FE mesh. FE simulations of crash conditions are reported in the literature to sometimes exhibit

idealistic behavior, especially for perfectly straight sections and sheet components. This step is introduced to avoid such idealization. In (Hamza and Saitou 2004b), the FE model for every point in the data grid was run 5 times.

5. Generate an averaged loading curve for each of the loading cases (axial, bending and twisting) using the data obtained in steps #3 and #4.
6. Identify the spring tuning parameters (F_e, δ_e, F_p, δ_p, F_s, and δ_s)$_{\text{axial}}$ that make an EM nonlinear spring (Equation 4.10) fit as best as possible within the average axial load deformation curve
7. Repeat step #6 for the bending and twisting load cases to obtain (F_e, δ_e, F_p, δ_p, F_s, and δ_s)$_{\text{bending-vertical}}$, (F_e, δ_e, F_p, δ_p, F_s, and δ_s)$_{\text{bending-transverse}}$ and (F_e, δ_e, F_p, δ_p, F_s, and δ_s)$_{\text{twisting}}$
8. Repeat steps #3 to #7 for every point in the grid of data points that span the ranges of the cross-section variables
9. The component database is now ready for use, and given some cross-section dimensions, the spring tuning parameters (F_e, δ_e, F_p, δ_p, F_s, and δ_s for axial, bending and twisting) may be recalled for use with an EM model. For dimensions that do not exactly correspond to a grid point in the database, a suitable interpolation scheme is employed. In (Hamza and Saitou 2004b) a radial basis neural network (Haykin, 1998, Mathworks 2001) seemed to provided good interpolation performance.

A further update to the EM modeling capability was the addition of a *Side-squish* nonlinear spring, which allows modeling of situations when a structural member gets squeezed between two rigid bodies (sketched in Fig. 4.15). An example of a side-squish situation is when the vehicle bumper gets squeezed between an object it hits, and some of the power-train components. This situation is not a well represented as axial, bending, nor torsional deformation, yet clearly involves crash energy dissipation. In the EM implementation, the side-squish spring also follows equation 4.10, but with the spring constants identified from a different component database. The database was generated following the steps #1 through #9, with only side-squish crash conditions being considered.

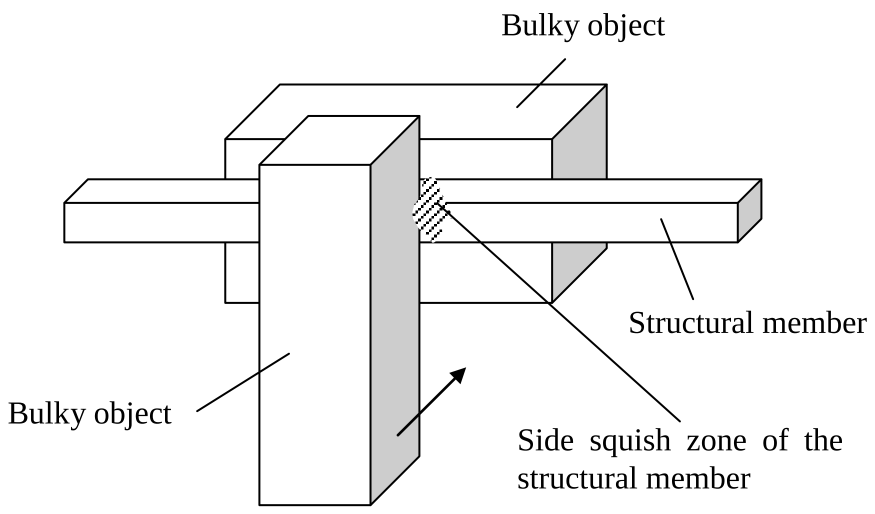


Fig. 4.15. Sketch of a side-squish crash condition

The nonlinear spring described by equation 4.10 is the fundamental component in EM models, which is implemented into a computation code via the C++ programming language. Construction of an EM model is similar to the construction of a FE model made entirely of beam elements. However, instead of nodes and beam elements, there are rigid point-masses connected via prismatic and revolute joints (spherical joints, in case of 3D analysis). A brief tutorial on the use of the developed software to construct an EM model is provided in Appendix A.

Simulations via EM models were successful in the discovery of the desirable crash mode for vehicle structures of small and medium levels of complexity (Hamza and Saitou 2003, 2004(a, c), 2005). Application for the discovery of the desirable CM for a model representing the front half of a vehicle is presented in Chapter 7.

4.4 Summary

This chapter presented a qualitative overview of structural crash modes. Simple examples were presented to demonstrate the crash mode hypotheses as being distinct, characteristic and provide a partitioning of the design space into regions where the design exhibit similar crashworthiness performance. The chapter also discussed the discovery of desirable crash mode(s) for a vehicle structure, and how reduced order dynamic model may be beneficial for the exploration task. Equivalent Mechanism models, being a reduced order dynamic model developed exclusively for crash modeling, were presented in this chapter as a candidate modeling approach for discovery of the desirable crash mode.

CHAPTER 5
QUANTIFICATION OF CRASH MODES

Previous discussions of vehicle structural crash modes in this thesis have only considered the crash mode concept from a qualitative sense. In order to incorporate crash mode matching into the proposed methodology, quantitative definitions need to be established to allow for automated crash mode matching. This chapter introduces formal quantitative definitions for the crash mode as well as quantitative metrics for comparison of degree of mismatch between vehicle structure designs having different crash modes.

5.1 Quantitative Definition of Crash Modes

5.1.1 Definition

The crash mode (CM) is qualitatively understood as *the time history of deformations in various zones of the vehicle structure*. A quantitative definition for the crash mode is proposed in this thesis as a matrix of time series corresponding to different deformation types averaged over different regions of structural zones :

$$CM = (cm_{ij}(t)); \quad i = 1, 2, 3; \quad j = 1, \ldots, m \qquad \text{(Equation 5.1)}$$

where:

$cm_{ij}(t) \in R$ is the total deformation of a structural zone at time t

$t \in [0, T_f]$ is the time instant at which the structure is observed. t ranges from zero to the final crash time T_f

i is an index on deformation type, which can be one of three possible types: 1) axial crush, 2) bending and 3) side-squish

j is an index on the number of zones

m is the number of zones defined for the structure

Successful application of the proposed methodology relies on a proper selection of the structural zones and deformation types. The definition of the crash mode places no restriction on the definition of zones. An extreme case (and obviously improper) would be to regard the entire structure as one zone, which would not show a distinction between qualitatively identifiable different crash modes. Another improper extreme example would be to regard every element in a FE mesh as a separate zone, which while allowing distinction between different crash modes, has too many numerical modes that correspond to only few qualitatively-different modes. Also with the types of deformation, a variety of proper and improper choices are possible. Thus, it is the task of the designers (users of the proposed methodology) to ensure *proper* selection of zones as well as deformation types for the crash mode definition. While typically obvious to an experienced designer, some suggested guidelines for proper selection of zones are:

- A region of the structure, which would be qualitatively judged as a *structural member* should be regarded as one zone, possibly divided into two or three zones at most if it’s a large member

- Only significant types of deformation that contribute to crash energy absorption should be recorded. For example, in the case studies considered in this thesis, the index *i* of the deformation type had three possible values: 1 = axial crush, 2 = bending and 3 = side squish. It is noted that not every zone exhibits all of the deformation types, for example the bumper of a vehicle exhibits significant side squishing, but negligible axial deformation.

5.1.2 Calculation of the Crash Mode for Finite Element Models

Emanating from the definition of the crash mode in equation 5.1, this section provides an implementation to calculate the numeric values of the crash mode for FE models. A computer software is developed via visual C++ to assist the task. Fig. 5.1 shows a screen shot of the program, along with an example FE mesh. The program is compatible with FE mesh file for both LS-DYNA and PAM-Crash softwares.

The program user has the following tasks:

1. Decide on the structural zones, and types of deformation to be recorded for each zone
2. Define key observable cross-sections for deformation measurement for each zone
3. Define key observable nodes for displacement measurement for each observable cross-section

The program then automatically retrieves the time history of motions for the key observable nodes from the commercial software's solution file. The defined nodes for each key observable section are used to calculate the position at the centroid of the cross-section. The relative motion of the centroids of the section (Fig. 5.2) with respect to the initial undeformed position is used to calculate relative displacements and rotations via vector algebra. The total sum of deformation is then calculated for each defined zone. It is noted that the torsional component of the rotation is disregarded as not being a major contributor in the considered case studies, and that the two bending components at every cross-section are vectorially added for one effective bending value.

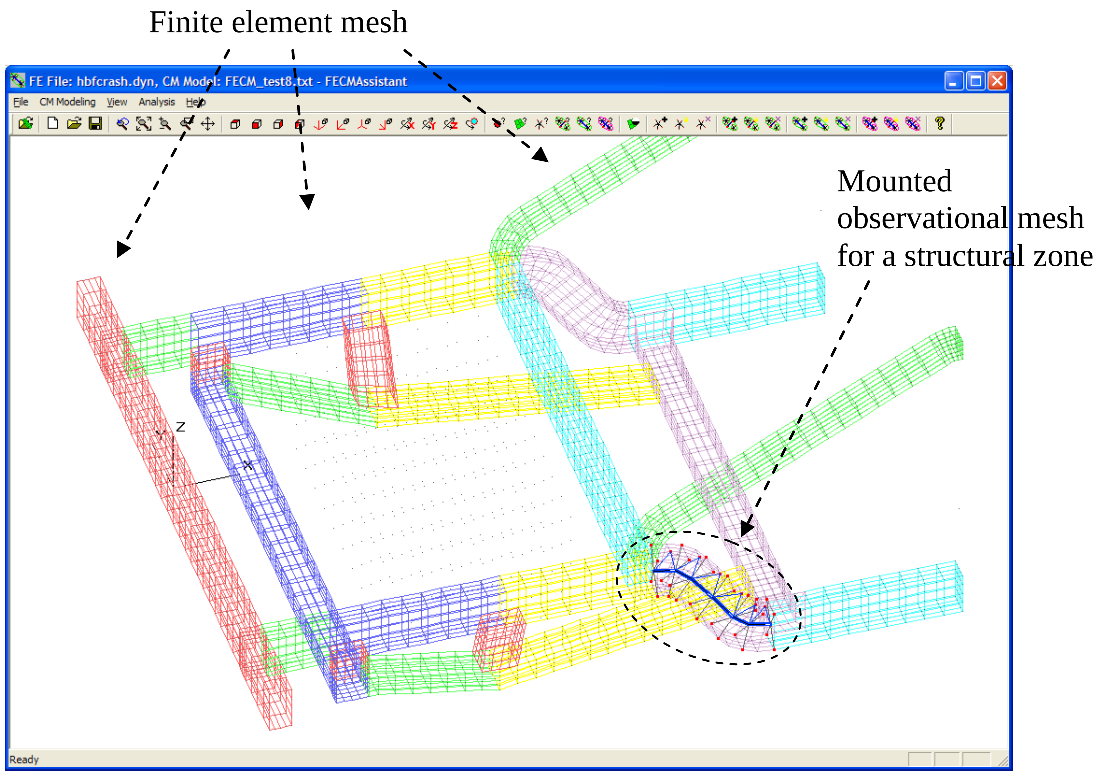

Finite element mesh
FE File: hbfcrash.dyn, CM Model: FECM_test8.txt - FECMAssistant
File
CM Modeling
View
Analysis
Help
Mounted observational mesh for a structural zone
Ready

Figure 5.1. Screen shot of computer program for assisting the calculation of crash mode for finite element models

Details of the vector algebra to calculate the total deformations of a zone are illustrated in Fig. 5.2. The *observational mesh* is defined by the designer and it includes *n* *key cross-sections* in every structural zone and several *key nodes* in each key cross section. The vector points $\boldsymbol{r}_{i,o}$ ($i = 1, \ldots n$) are the positions of the centroids of nodes in each of the key cross-sections in the structure's un-deformed state. Vector points $\boldsymbol{r}_{i,t}$ are the centroids' positions at some time *t* during the crash event.

The total axial crush in a zone at time *t* is calculated as:

$$cm_{1j}(t) = \sum_{i=1}^{n-1} \left\{ \left\| (r_{i+1,t} - r_{i,t}) \right\| - \left\| (r_{i+1,o} - r_{i,o}) \right\| \right\} \qquad \text{(Equation 5.2)}$$

The total bending in a zone at time *t* is calculated as:

$$cm_{2j}(t) = \sum_{i=1}^{n-2} \left\{ ArcCos\left(\frac{(r_{i+2,t} - r_{i+1,t}) \bullet (r_{i+1,t} - r_{i,t})}{\left\| (r_{i+2,t} - r_{i+1,t}) \right\| \left\| (r_{i+1,t} - r_{i,t}) \right\|} \right) - ArcCos\left(\frac{(r_{i+2,o} - r_{i+1,o}) \bullet (r_{i+1,o} - r_{i,o})}{\left\| (r_{i+2,o} - r_{i+1,o}) \right\| \left\| (r_{i+1,o} - r_{i,o}) \right\|} \right) \right\}$$

(Equation 5.3)

The total side-squish in a zone is calculated exactly like the axial crush (using equation 5.2), except the observational mesh for the side squish is selected along the transverse direction of the structural zone (instead of the axial direction in case of the axial crush).

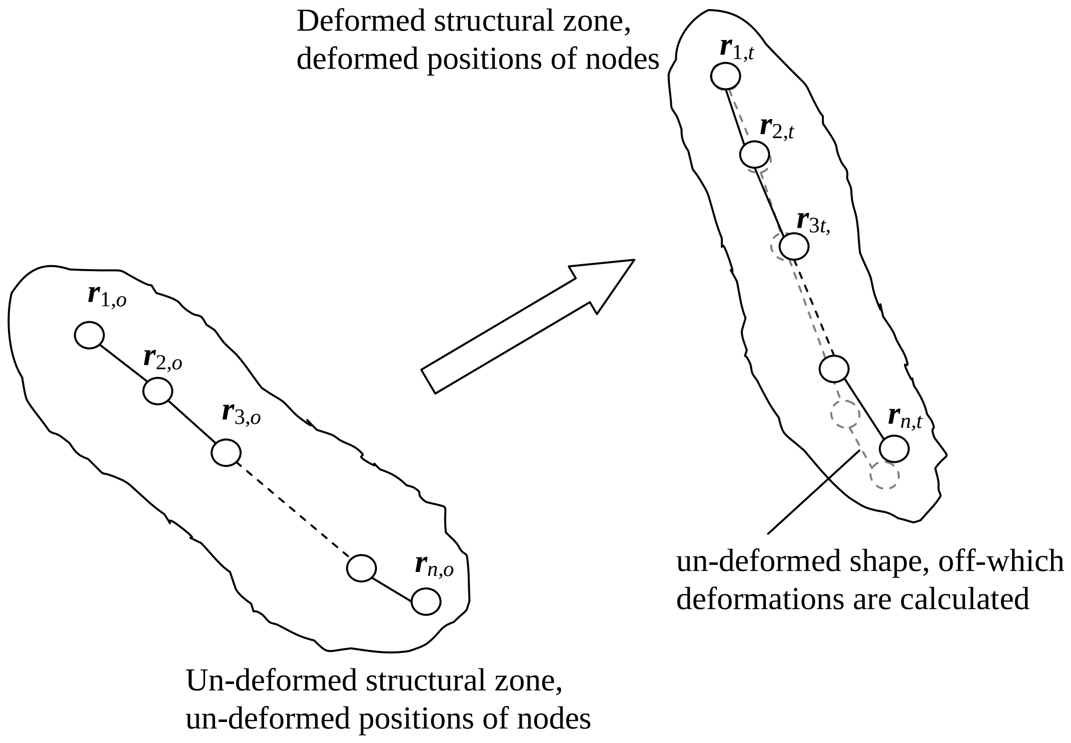


Figure 5.2. Tracking zone deformations on a FE mesh for calculating the crash mode

5.1.3 Calculation of the Crash Mode for Equivalent Mechanism Models

Calculation of equivalent values of the crash mode for EM models is a simple task, because the EM models already incorporate deformable joints with mounted nonlinear springs. The program user needs only to identify the structural members that comprise a zone, and the deformation types of interest. The program then automatically sums over the deformations of the joints in the defined structural zones to calculate the CM values; axial joints for axial crush, rotational joints for bending, and side-squish axial joints for side squish values.

5.2 Comparison of Crash Modes

5.2.1 Metric for Degree of Crash Mode Mismatch

With the capability to calculate the crash mode values for a given structural model discussed in section 5.1.1, this section discusses metrics for judging the degree of match or mismatch between a crash mode of a given structure and a desired crash mode. A mismatch metric is defined in this section that corresponds to the definition of the crash mode in section 5.1.1. In essence, the crash mode mismatch metric is a two-dimensional matrix with dimensions being the number of zones and the number of deformation types. Each entry in the matrix is a time integral of the absolute error between actual and desired crash modes, normalized with respect to the desired crash mode. The mismatch metric is defined as:

$$CMM = (cmm_{ij}); \quad i = 1, 2, 3; \quad j = 1, \ldots, m \qquad \text{(Equation 5.4)}$$

$$cmm_{ij} = \frac{\int_{t=0}^{T_f} \left| cm_{ij}(t) - cm_{ij}^{*}(t) \right| dt}{\int_{t=0}^{T_f} \left| cm_{ij}^{*}(t) \right| dt} \qquad \text{(Equation 5.5)}$$

where:

$cm_{ij}(t)$ is the time series of deformation type i in zone j from the crash mode of the design currently being tested. This data is usually available as a time series rather than a continuous function of time because it is obtained from a FE model that only saves deformation values at discrete time intervals.

$cm_{ij}^{*}(t)$ is the time series of deformation type i in zone j from the desired crash mode

In many practical cases, it is often difficult to have accurate or detailed time history information of the desirable crash mode. This issue motivated the development of a more compact form of the crash mismatching metric. Examination of the profile of a times-series from a typical crash mode (Fig. 5.3) reveals three main stages of the time series:

1. Deformation occurring somewhere else in the structure, and deformation in the considered zone is negligible

2. Plastic deformation: which happens fairly rapid and during which, the considered zone is contributing to dissipate crash energy while plastically deforming

3. Retention of deformed shape: in which the considered zone has consumed its folding space (cannot deform any more), and it is no longer contributing to the dissipation of crash energy. At this stage, if the crash energy is not fully dissipated, the structure typically undergoing further deformation in other zones of the structure, but not the considered zone.

This allows for a compact approximation of the desired crash mode as a step function:

$$cm^{*}_{ij}(t) = \theta(t - t_o)\, d \qquad \text{(Equation 5.6)}$$

$$\theta(t) = \begin{cases} 0 & \text{if } t < 0 \\ 1 & \text{otherwise} \end{cases} \qquad \text{(Equation 5.7)}$$

Where $\theta(t)$ is the unit step function, d and t_o are the magnitude and start time of the step (Fig 3.3), respectively.

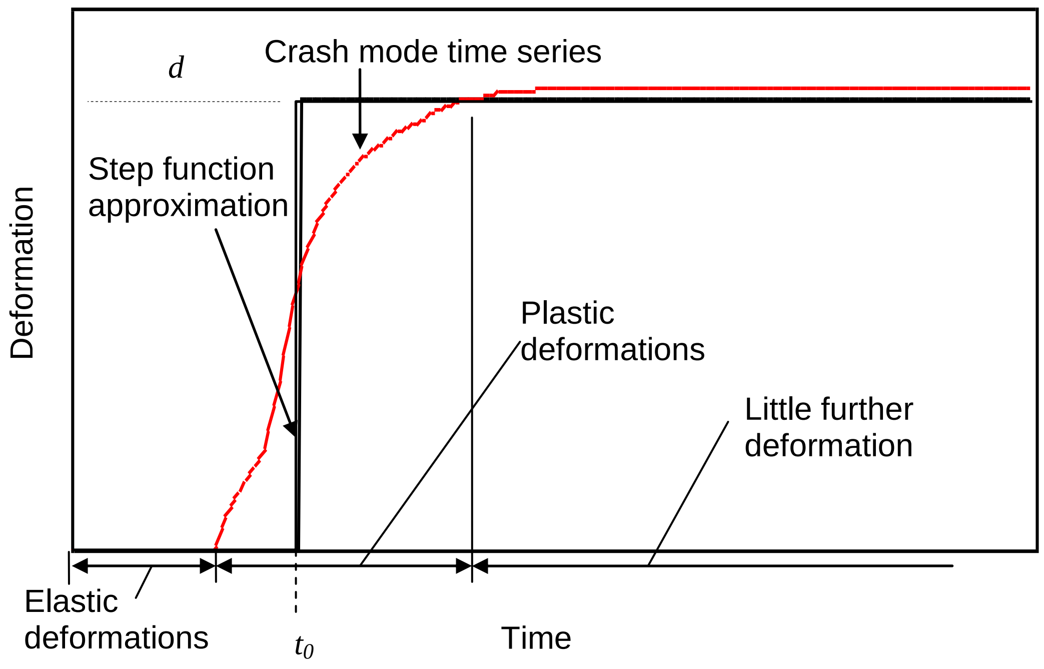


Fig. 5.3. Typical profile of a crash mode time series *cm*$_{ij}$(*t*) and its approximation as a step function.

Adoption of the step function approximation for the desirable crash mode significantly facilitates the use of prior expertise, because the designer using the proposed methodology only needs to provide the pairs (t_0, d) for each cm_{ij}^* instead of a full time series.

5.2.2 Relaxed Metric for Degree of Crash Mode Mismatch

While the crash mismatch metric introduced in equation 5.5 seems like a natural definition, it has the downside of making no numerical distinction between a crash mode exhibiting more deformation than desired values from a crash mode exhibiting roughly equal amount of less deformation than the desired values. Furthermore, crash modes for designs that do positive error over some portion of the time integral then recover via negative error over another portion, while not exactly matching the desired crash mode, are better than other designs that go only one side of the error (positive or negative). This can be argued because the deformations in structural zones are generally associated with energy dissipation, a truly mismatching crash mode of a design is one that completely misses the target value in some zone / deformation type. The need for exact match at every time instant can redundant. Fig. 5.4 shows a sketch of two cm_{ij} time series. The crash mode mismatch metric evaluate to the same value for both series (evaluated via equation 5.5). However, from a qualitative sense, series #1 (exhibiting positive and negative errors over the time history) is a better match to the desired crash mode

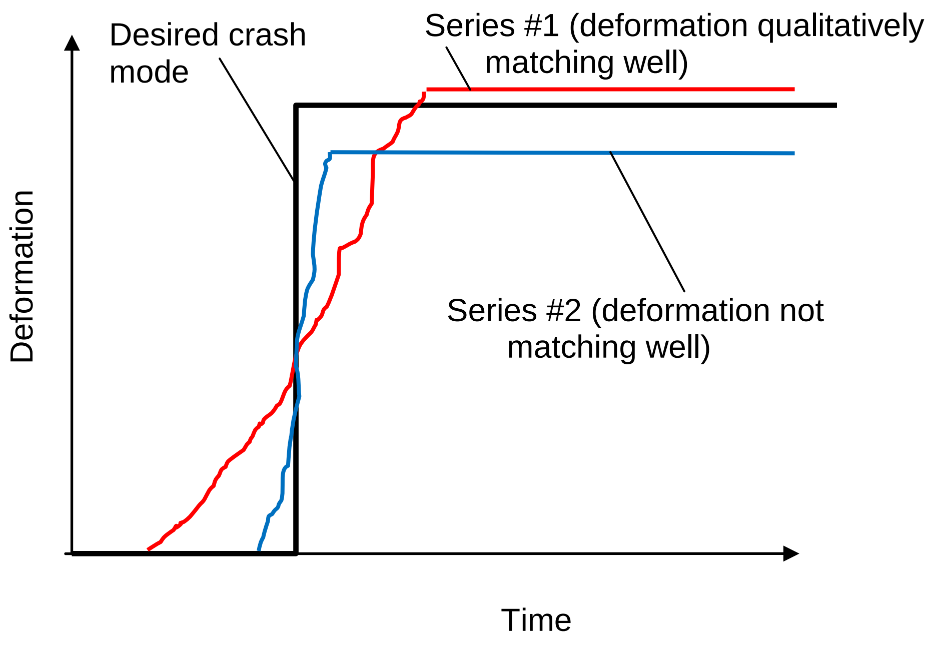


Fig. 5.4. Example time series mismatch relative to a desired crash mode.

To address the shortcomings of the mismatch metric calculated via equation 5.5, a relaxation of equation 5.5 is introduced by removing the absolute operator from the error integrals.

$$cmm_{ij} = \frac{\int_{t=0}^{T_f} \left(cm_{ij}(t) - cm_{ij}^{*}(t) \right) dt}{\int_{t=0}^{T_f} cm_{ij}^{*}(t) dt} \qquad \text{(Equation 5.8)}$$

The relaxed mismatch metric is perceived beneficial as it can take on either positive or negative values, which can be a direct indication to the designer (or the automated crash mode matching algorithm) as to which structural zones are overly strong

(cmm_{ij} evaluates to a negative value) or overly weak (cmm_{ij} evaluates to a positive value). Also, in a scenario such as the one presented in Fig. 5.4, series #1, which has a better qualitative match to the desired crash mode than series #2, would have less mismatch error than series #2 when the relaxed mismatch metric (equation 5.8) is used.

In the examples and case studies presented in chapters 6, 7 and 8, the relaxed mismatch metric (equation 5.8) will be the one used for crash modes comparison and crash mode matching.

5.3 Summary

This chapter introduced quantitative definitions for the crash mode as well as quantitative metrics for comparison of degree of mismatch between vehicle structure designs having different crash modes. The next chapter presents the core algorithm in the proposed methodology that uses the crash mode definitions in this chapter to perform automated crash mode matching.

CHAPTER 6

AUTOMATED CRASH MODE MATCHING ALGORITHM

This chapter presents the final piece in the proposed methodology, which is the automated design optimization algorithm that uses crash modes matching to accelerate the discovery of good designs. A simple example is also provided in this chapter to demonstrate the steps of the algorithm and provide a comparison with other existing optimization algorithms.

6.1 Algorithm Overview

The algorithm for automated crash mode matching is based stochastic sampling of the search space (Fig. 6.1). The algorithm seeks to find the optimum values of the design variables ($\boldsymbol{x}^*$) that minimize the objective(s) $f(\boldsymbol{x})$ (equation 3.1), subject to the constraints $\boldsymbol{g}(\boldsymbol{x})$ (equation 3.2).

The algorithm is started at some initial design and iterations are repeated throughout the search. The algorithm draws out a number of sample designs in every iteration according to a multi-dimensional Gaussian distribution on the design variables. The averages and standard deviations of the Gaussian distributions for each design variable are adjusted at the beginning of every iteration based on the degree of mismatch between the crash mode of the current design and the desired crash mode. A copy the best encountered design is stored separately before then the best among an iteration's samples becomes the new current design. Iterations are continued till the discovery of a design that has satisfactory performance or until a pre-set number of sample designs are examined.

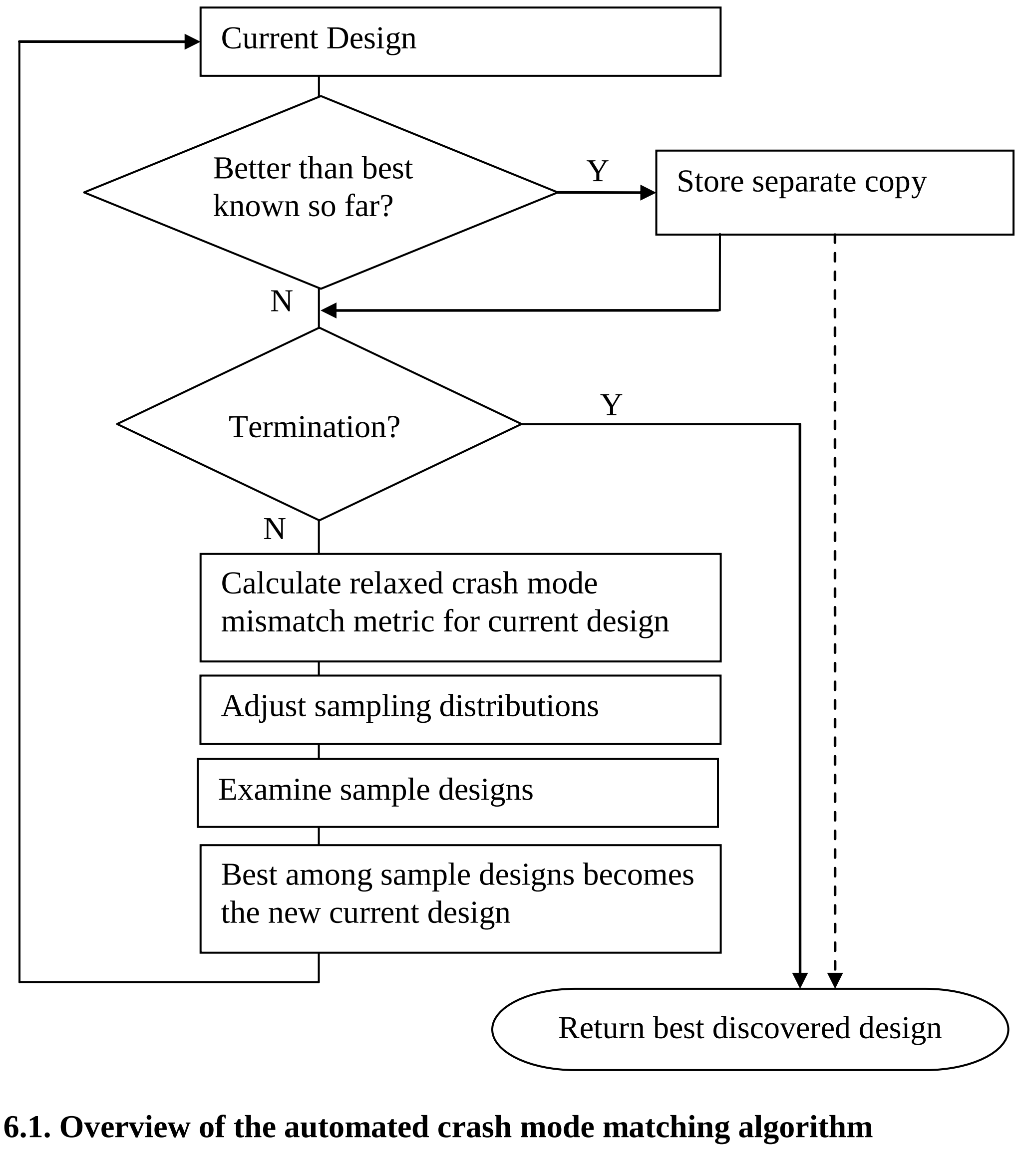

Current Design
Better than best known so far?
Y
Store separate copy
N
Termination?
Y
N
Calculate relaxed crash mode mismatch metric for current design
Adjust sampling distributions
Examine sample designs
Best among sample designs becomes the new current design
Return best discovered design

Figure 6.1. Overview of the automated crash mode matching algorithm

6.2 Algorithm Inputs

There are four categories of the algorithm inputs, these are listed as:

1. Definition of the optimization problem: Detailed FE model and definitions of the Design variables ($\boldsymbol{x}$), (including lower and upper bounds on the design variables, $\boldsymbol{x}_{\boldsymbol{min}}$ and $\boldsymbol{x}_{\boldsymbol{max}}$ respectively), Objective(s) ($f(\boldsymbol{x})$) and constraints ($\boldsymbol{g}(\boldsymbol{x})$).
2. Crash mode matching data: Definition of the zones and deformation types for calculating the crash modes, as well as values for the desirable crash mode $cm_{ij}^{*} = (t_{0ij}, d_{ij})$, $i = 1, 2, 3$, $j = 1, \ldots, m$
3. Sampling distribution adjustment rules: R_l, $l = 1, \ldots, nRules$. The automated crash mode matching algorithm uses fuzzy logic (Hopgood 2001) to adjust the Gaussian distributions on the design variables for sampling the search space. An advantage to fuzzy logic is that it allows application of logical rules to qualitatively assessed quantities then recommends numerical values as an output. An example rule is shown in Fig. 6.2. Every rule has a logical term and an action term. The fuzzy sets used in this dissertation in the logical term can be one of {NH: highly negative, NL: low negative, Z: near zero, PL: low positive, PH: highly positive}. The nominal adjustment value ($\tilde{a}$) is some number set for every defined fuzzy rule. A guideline for selecting the value of $\tilde{a}$ is given as:
    - For small increase/decrease adjustment, $\tilde{a}$ is recommended to be approx. +/- one tenth of the range of the design variable whose distribution is being adjusted. This is only a guideline that seemed

to work well in some test problems. The algorithm user may use other values if appropriate knowledge of the problem is at hand.

- For small increase/decrease adjustment, $\tilde{a}$ is recommended to be approx. +/- one fifth of the range of the design variable whose distribution is being adjusted

4. Tuning parameters for the algorithm: Number of sample designs to examine in each iteration (*nIterSamples*), minimum values for the standard deviations of the Gaussian distributions ($\sigma_{i,min}$ i = 1, …, *nVar*) and a maximum number of iterations (*nIter)*

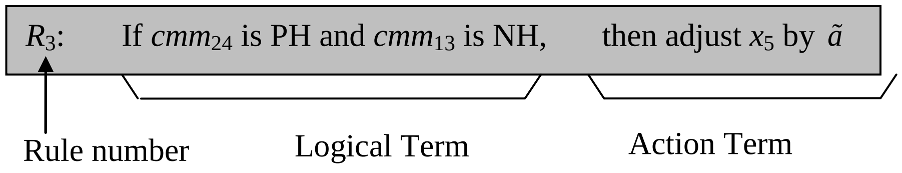

R_3:
If cmm_{24} is PH and cmm_{13} is NH,
then adjust x_5 by $\tilde{a}$
Rule number
Logical Term
Action Term

Fig. 6.2 Example fuzzy design adjustment rule

6.3 Algorithm Steps

6.3.1 Algorithm Pseudo-code

```
1. Start at an initial design $\boldsymbol{x}^{\mathbf{o}}$, Evaluate $\boldsymbol{x}^{\mathbf{o}}$, set current best design $\boldsymbol{x}^{*} = \boldsymbol{x}^{\mathbf{o}}$
2. Initialize iteration counter $iIter = 0$
```

3. If $\boldsymbol{x}^{\mathbf{o}}$ is better than $\boldsymbol{x}^*$, set $\boldsymbol{x}^* = \boldsymbol{x}^{\mathbf{o}}$
4. If $iIter \geq nIter$, goto step #18 (termination)
5. Evaluate the relaxed crash mode mismatch metric for current design cmm_{ij} $i = 1, 2, 3, j = 1, \ldots, m$ (according to equation 5.8)
6. Initialize the Gaussian distributions for sampling the design space for current iteration: $\boldsymbol{\mu} = \boldsymbol{x}^{\mathbf{o}}$, $\boldsymbol{\sigma} = \mathbf{0}$
7. Initialize the fuzzy logic rules counter: $l = 0$
8. Set $l = l + 1$
9. Evaluate the fuzzy rule R_l. Every fuzzy rule returns a rule activity-value ($a \in [0, 1]$), index of affected design variable ($i_a \in \{1, 2, \ldots, nVar\}$) and adjustment level ($\tilde{a}$). Calculate the sampling distribution adjustment value $\bar{a}$ according to:

$$\bar{a} = a.\tilde{a} \qquad \text{(Equation 6.1)}$$

10. Adjust the Gaussian distributions: $\mu_{i_a} = \mu_{i_a} + \bar{a}$, $\sigma_{i_a} = \sigma_{i_a} + |\bar{a}|$
11. If $l < nRules$, goto step #8 (loop until all fuzzy rules have been evaluated)
12. Ensure the standard deviations of the Gaussian distributions are equal to or larger than the minimum: If $\sigma_{i,} < \sigma_{i,min}$, set $\sigma_{i,} = \sigma_{i,min}$, $i = 1, \ldots, nVar$
13. Generate sample designs $\boldsymbol{x}^k$, $k = 1, \ldots, nIterSamples$ from the search space with the Gaussian probability distribution ($\boldsymbol{\mu}, \boldsymbol{\sigma}$)
14. Run the detailed FE model for each sample design, save the FE run results, and evaluate $f(\boldsymbol{x}^k)$ and $(\boldsymbol{g}(\boldsymbol{x}^k)$, $k = 1, \ldots, nIterSamples$
15. Identify $\boldsymbol{x}^\dagger$, as the best among the drawn samples $\boldsymbol{x}^k$, $k = 1, \ldots, nIterSamples$
16. Set $\boldsymbol{x}^{\mathbf{o}} = \boldsymbol{x}^\dagger$ and $iIter = iIter + 1$

17. Goto step #3 (beginning next iteration of the main loop)

18. Return $\boldsymbol{x}^*$, $f(\boldsymbol{x}^*)$ and $(\boldsymbol{g}(\boldsymbol{x}^*)$

6.3.2 Algorithm Details

Implementation of the algorithm into computer code is performed via the C++ programming language. Source code of the program is provided in Appendix D. Details of the main steps of the algorithm are provided as follows:

The proposed algorithm examines a number of sample designs in every iteration then uses the best among the sample as the new current design, but best among the samples may not be better than current design. Step #3 of the algorithm ensures that a separate record is kept for the best encountered design during the stochastic sampling. This is known in Genetic Algorithms literature as “elitism”, which is important for the convergence properties of any optimization algorithm that uses stochastic sampling of the search space.

The criteria for comparison between two designs as to which is “better” which is used in step #3 (and later in step #15) are summarized as:

- If both designs are infeasible (one or some $\boldsymbol{g}(\boldsymbol{x}) > 0$), a penalty function is employed (weighted sum of amount of constraint violation) for the comparison.
- If one design is feasible (all $\boldsymbol{g}(\boldsymbol{x}) \leq 0$), yet the other design is infeasible (one or some $\boldsymbol{g}(\boldsymbol{x}) > 0$), the feasible design is a better design

- If both designs are feasible (all $\boldsymbol{g}(\boldsymbol{x}) \leq 0$), the design with least value of $f(\boldsymbol{x})$ is the better design. Ties are broken randomly. If there are multiple objectives, weighting is employed.

Step #5 uses the saved results of the detailed FE model for current design to evaluate the relaxed crash mode mismatch metric (Equation 5.8).

Step #6 initializes the Gaussian distributions for generating the sample designs in the current iteration. The distributions are initially centered on the current design ($\boldsymbol{\mu} = \boldsymbol{x}^{\mathbf{0}}$) and having zero variance ($\boldsymbol{\sigma} = \mathbf{0}$)

Steps #7 through #11 is a loop over the defined fuzzy rules (Hopgood 2001) for design adjustment (third category of the algorithm inputs in section 6.2). Each rule has a qualitative logical term and a qualitative action term (Fig. 6.2), explained as follows:

- The logical term typically examines part of the relaxed crash mode mismatch metric. For example: "If cmm_{12} is NH".. The logical term is evaluated in the algorithm into the action value a, which is simply the value of the *membership function* (Hopgood 2001) in the qualitative level values {NH, NL, Z, PL, PH}. The membership functions implemented in the algorithm are two-sided sigmoid functions (Mathworks, 2001), shown in Fig. 6.3.

- For logical terms that employ multiple logical expressions, for example "If *cm*$_{13}$ is PH AND *cm*$_{25}$ is Z", the action value a is calculated as the minimum of the membership values in each expression.
- The action term of the fuzzy rule provides an index i_a for the design variable whose sampling distribution is to be modified, and the adjustment value $\tilde{a}$. The value of $\tilde{a}$ is used for the calculation of the total adjustment value $\bar{a}$ via equation 6.1. Finally, the mean and standard deviation values of the design variable in question (μ_{i_a}, σ_{i_a}) are updated in step #10.

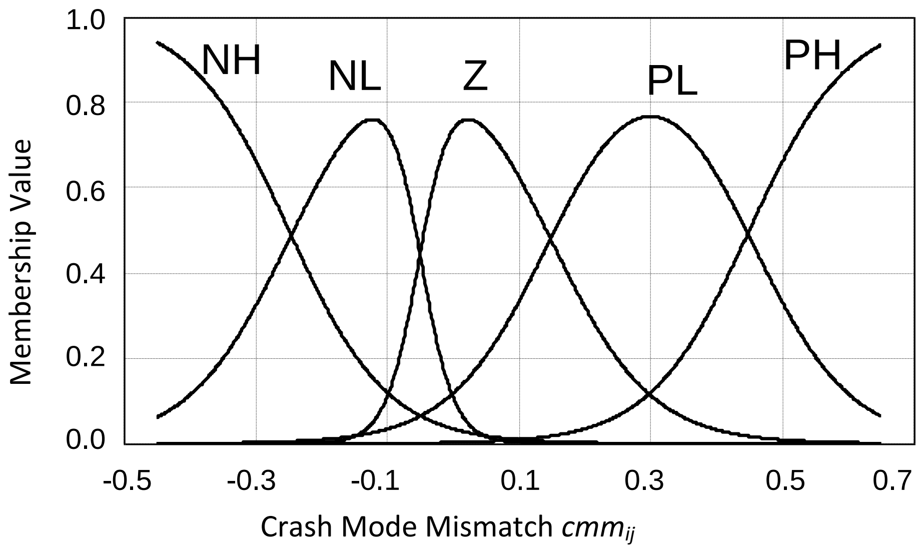


Fig. 6.3. Membership functions for the fuzzy design adjustment rules

Step #12 re-adjusts the standard deviation on the sampling distributions so that they are at least at the minimum values as in the algorithm tuning inputs (fourth category of the algorithm inputs in section 6.2)

Step #13 uses the computer's random number generator to generate a number of sample designs with the design variable values following the Gaussian distribution (μ, σ).

Steps #14 through #16 go through performing the detailed FE runs for each of the generated sample designs. Best design among the samples becomes the new current design and the iteration counter is updated.

The algorithm iterations are continued till the pre-set number of iterations is reached, then the best encountered design (which is being recorded in step #3) is returned.

6.4 Algorithm Convergence

The proposed algorithm for crash mode matching is a stochastic search technique. As with all stochastic search algorithms, there are the following drawbacks:

- No guarantees on producing the same answer to the same problem every time the algorithm is run
- Convergence to the global optimum is not guaranteed unless certain criteria are fulfilled

The discussion in this section follows similar analysis to one of the convergence proofs of genetic algorithms (Coello *et. al.* 2001). There are two sufficient conditions for global convergence of a genetic algorithm, which are summarized as:

- Elitism: This is keeping a separate record of the best member(s) in the GA population until a better one(s) is discovered. In analogy, this condition is met in step #3 of the proposed algorithm.
- Reachability: This is having non-zero probability to reach the optimum from any randomized initial population. In analogy, this condition is met in the proposed algorithm because the Gaussian distributions for the design space sampling have non-zero probability over the entire search space.

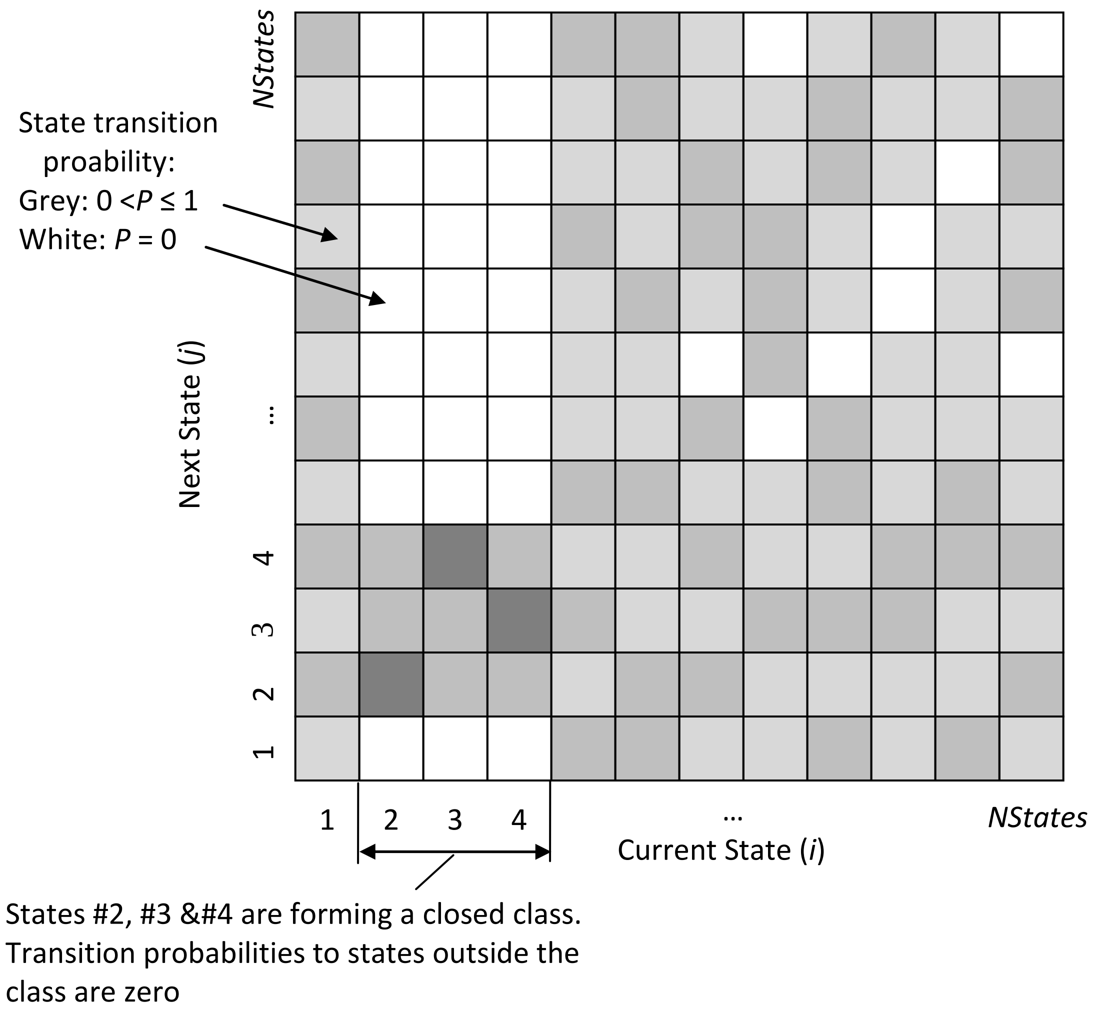
The discussion in this section shows that with the elitism and reachability conditions met, it is possible to establish the proof of convergence for the proposed algorithm as well.

6.4.1 Algorithm Convergence for Discrete Design Variables

The proof in this section considers the case when all the design variables in the vector $\boldsymbol{x}$ are discrete. The proof will employ Markov chains (Resnick 1992) to show that the best encountered design in the algorithm ($\boldsymbol{x}^*$) converges to the global optimum ($\boldsymbol{x}^{**}$) within a finite number of iterations.

Some of the definitions from Discrete Markov Chains (Resnick 1992) are used in the proof and are listed as:

- *Stochastic state*: is some observable quantity which may change over time.
- *Discrete-state*: is a stochastic state which may only take on certain discrete values that may be enumerated $i = 1, 2, \ldots, NStates$
- *Discrete-time-discrete-state*: is a discrete state which changes over time, but only during regular discrete periods of time.
- *State Transition Matrix* (**P)**: is a matrix of probability values for the transition between one state to another in a *discrete-time-discrete-state* stochastic process. $\mathbf{P} = \{P_{ij}\} = P(i \rightarrow j)$. Fig. 6.4 shows an illustration of a state transition matrix. The value P_{ij} inside a cell of the matrix is the probability of transition from the discrete state i (row index) to the discrete state j (column index) in the next time step. A grey colored cell illustrates P_{ij} having a non-zero value, while a white colored cell illustrates $P_{ij} = 0$.
- *Class of States*: are a group of states having some common property. For example, in the observation of people in a queue, with the state being the number of people in the queue. All states that correspond to a queue less than 5 people may be considered a class of states
- *Closed Class of States*: are classes of states that have only zero values in the state transition matrix for transition between any of the states in the closed class to a state outside the closed class. By definition, when the state of a stochastic process reaches a closed class, it may change to other states within the closed class, but cannot change to a state outside the closed class. In Fig. 6.4, the discrete states 2, 3 and 4 form a close class because the probability of transition from any of those states to some other state (besides states 2, 3 and 4) is zero.



State transition proability:
Grey: 0 <P ≤ 1
White: P = 0
NStates
Next State (j)
...
4
3
2
1
1
2
3
4
...
NStates
Current State (i)
States #2, #3 &#4 are forming a closed class. Transition probabilities to states outside the class are zero

Fig. 6.4 Illustrations of some Concepts in Discrete-Time-Discrete-Event Markov Chains

The proposed algorithm for automated crash mode matching is modeled as a Markov Chain:

The state is defined as the pair of vectors ($\boldsymbol{x}^{\mathbf{0}}$, $\boldsymbol{x}^{*}$), *i.e.* current design and best encountered design so far

Since the design variables are discrete and have lower and upper bounds (from algorithm inputs, section 6.2), then the number of all possible values for ($\boldsymbol{x}^{\mathbf{o}}$, $\boldsymbol{x}^{*}$), is finite and can be enumerated 1, …, *NStates*

Thus, the algorithm state ($\boldsymbol{x}^{\mathbf{o}}$, $\boldsymbol{x}^{*}$) is a discrete state

The algorithm state ($\boldsymbol{x}^{\mathbf{o}}$, $\boldsymbol{x}^{*}$) only changes (updates to values of $\boldsymbol{x}^{\mathbf{o}}$ and/or $\boldsymbol{x}^{*}$) once per iteration of the main loop (steps #3 through #17) in the proposed algorithm

Thus, the algorithm state ($\boldsymbol{x}^{\mathbf{o}}$, $\boldsymbol{x}^{*}$) is a discrete-time-discrete-state. With time in this case being the iteration counter *iIter*

Define a class C^{*}: ($\boldsymbol{x}$, $\boldsymbol{x}^{**}$) as all the states that include the global optimum

C^{*} is closed class of states. This is established via step #3 in the proposed algorithm. When/if the optimum design ($\boldsymbol{x}^{**}$) is encountered by the algorithm, the algorithm will set $\boldsymbol{x}^{*} = \boldsymbol{x}^{**}$. Since $\boldsymbol{x}^{*}$ now holds a copy of the optimum design, it will no longer be changed again in step #3 of the algorithm.

Since the distributions for generating the samples in the proposed algorithm are Gaussian distributions thus having non-zero values of the state transition probabilities (P_{ij}) in the state transition matrix are all non-zero as long as $\boldsymbol{x}^{*} \neq \boldsymbol{x}^{**}$.(so there exists a chance for $\boldsymbol{x}^{*}$ .to change in step #3).

Thus, C^* is the only closed class in the Markov Chain.

The absorption property (Resnick 1992) of discrete-time-discrete-state Markov chains that have a finite number of states, indicates that the state of the Markov chain gets absorbed in one of the closed classes in finite time. Note: time in this Markov Chain is the number of iteration of the algorithm *iIter*

Since C^* is the only closed class in the Markov Chain, it is thus established that the algorithm can be started at some ($\boldsymbol{x}^{\mathbf{o}}$, $\boldsymbol{x}^{\mathbf{o}}$) and it converges to ($\boldsymbol{x}$, $\boldsymbol{x}^{**}$) within a finite number of *iIter*

6.4.2 Algorithm Convergence for Continuous Design Variables

The proposed algorithm is meant to be applicable to problems involving discrete, continuous or mixed discrete and continuous variables. While it is not possible to guarantee convergence to the exact optimum in finite time if one or some of the design variables are continuous, this sub-section provides a discussion to show that it is possible to achieve convergence to a design that is at ε-distance (which can be made very a small distance in the space of design variables, but not zero) from the optimum design $\boldsymbol{x}^{**}$ within finite time.

Replace all the design variables x_i , i = 1, …, *nVar* that may take continuous values, with discrete variables that may only take on values:

$$x_i^{\dagger} = x_{i,min} + k \,.\, (x_{i,max} - x_{i,min}) / \, nDiv,\ k = 0, 1, \ldots, nDiv \qquad \text{(Equation 6.2)}$$

The discrete problem ($\boldsymbol{x}^{\dagger}$ replacing $\boldsymbol{x}$) now replaces the original problem. In the discrete problem, there exists an optimal design $\boldsymbol{x}^{\dagger *}$ that is within $\boldsymbol{\varepsilon}$-distance from $\boldsymbol{x}^{**}$. $\boldsymbol{\varepsilon}$ can be made sufficiently small (but still $\boldsymbol{\varepsilon} > \mathbf{0}$) by choosing *nDiv* sufficiently large.

The convergence proof in section 6.4.1 is repeated for the discrete problem. This shows the algorithm started at some ($\boldsymbol{x}^{\mathbf{o}\dagger}$, $\boldsymbol{x}^{\mathbf{o}\dagger}$) and it converges to ($\boldsymbol{x}^{\dagger}$, $\boldsymbol{x}^{\dagger *}$) within a finite number of *iIter*

6.5 Demonstrative Example

This section presents a step by step demonstration of the automated crash mode matching algorithm to vehicle mid-rail problem that was introduced in chapter 4. The model of the problem is shown in Fig. 6.5 and data is summarized in Table 6.1. The sheet metal thicknesses (t_1, t_2) in zone #1 and zone #2 respectively are independent variables, and are assumed to take discrete values between 1.0mm and 4.0mm in steps of 0.2mm.

6.5.1 Inputs to the Algorithm

<u>Category #1 of Inputs: Objectives, Constrains & Design Variables:</u>

The objective is to reduce the total deformation in zone #2:

$$\text{Min.} \quad f(t_1, t_2) = \text{Total deformation in zone \#2} \qquad \text{(Equation 6.3)}$$

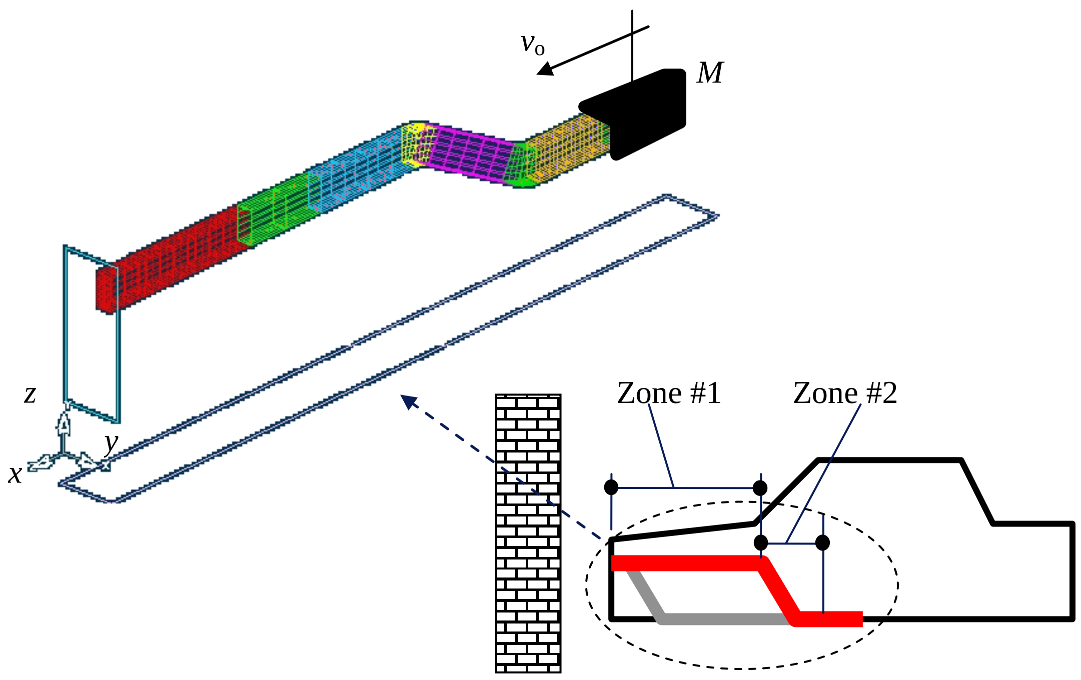


Figure 6.5. Vehicle Mid Rail Model

There are two independent design variables (t_1, t_2) which are the sheet metal thicknesses in zone #1 and zone #2 respectively. Both t_1, t_2 $\in$ {1.0, 1.2, 1.4, …, 4.0}. There are no explicit constraints in this example.

It is noted that there are no constraints on structural weight in this optimization problem. At first glance, this might imply that the optimum design to minimize deformation would be attained by setting both variables t_1, t_2 at maximum range (for maximum structural stiffness). However, this is not true because it leads to an undesirable crash mode (deformation at zone #2 first).

Table 6.1. Parameter values for the Vehicle Mid Rail example

Symbol	Description	Value	Unit
M	Payload mass	200.0	kg
v_o	Initial velocity	10.0	m/s
L_1	Length of Zone #1	650.0	mm
L_2	Length of Zone #2	375.0	mm
h	Box section height	90.0	mm
b	Box section width	60.0	mm
ϕ	Inclination angle	23.5	deg
d	Z-direction rail axis offset	120.0	mm
t_1	Sheet thickness in Zone #1	(Variable)	mm
t_2	Sheet thickness in Zone #2	(Variable)	mm

<u>Category #2 of Inputs: Desirable Crash Mode:</u>

A good estimate of the desirable crash mode for this structure is to have a large amount of axial deformation in zone #1 and a small amount of bending deformation in zone #2. When accurate time history profile for the desired crash mode is not available, it is convenient to use the compact form of the desired crash mode (Equation 5.6). The selected values in this example are:

$cm_{11}^{*} \equiv (t_{01}, d_1) = (0.03, 500)$: 500mm total axial deformation in zone #1, reaching half the deformation value at time 0.03 (30ms after hitting the barrier). 500mm is reasoned to be a large amount of deformation (compared to

650mm, which is the total axial length of zone #1) and 30ms is reasoned to be an early crush (initial crash velocity 10m/s, for the deformation to reach half its final value in zone #1 after 30ms, means crush must initiate in zone #1).

$cm_{22}^{*} \equiv (t_{02}, d_{1}) = (0.06, 0.1)$: 0.1 rad for total bending in zone #2, reaching half the deformation value at time 0.06 (60ms after hitting the barrier). 0.1rad is essentially requiring almost zero deformation in zone #2, however zero values are not allowed as inputs in desired crash mode otherwise it would cause a numerical singularity in Equation 5.8. 60ms is reasoned to be a late crush

Given the configuration of the structure, the rest of zone/deformation types (bending in zone #1 and axial deformations in zone #2) were not deemed important for this example study.

<u>Category #3 of Inputs: Sampling Distribution Adjustment Rules:</u>

No deep insights as to the behavior of the crash mode verses changes in the design variables are assumed available for this study. The only rules defined are simple-logic of: “if a structural zone is deforming too much, then it could use reinforcement, then the design variable affecting the zone should be increased”. A listing of the fuzzy rules is given as:

R_1: If cmm_{11} is NH then adjust t_1 by “-ve large amount”

R_2: If cmm_{11} is NL then adjust t_1 by “-ve small amount”

R_3: If cmm_{11} is PL then adjust t_1 by “+ve small amount”

R_4: If cmm_{11} is PH then adjust t_1 by “+ve large amount”

R_5: If cmm_{22} is NH then adjust t_2 by “-ve large amount”

R_6: If cmm_{22} is NL then adjust t_2 by “-ve small amount”

R_7: If cmm_{22} is PL then adjust t_2 by “+ve small amount”

R_8: If cmm_{22} is PH then adjust t_2 by “+ve large amount”

Category #4 of Inputs: Tuning Parameters for the Algorithm:

The number of samples per iteration was chosen as *nIterSamples* = 8

The total number of iterations was chosen as *nIter* = 20

The minimum value for the standard deviations on the sampling distributions was chosen as $\sigma_{1,min} = \sigma_{2,min} = 0.3$mm (within 5% to 10% of the variables’ minimum to maximum ranges)

6.5.2 Simulation of the Algorithm Steps

The sub-section goes through a simulation of steps of one iteration of the automated crash mode matching algorithm.

The initial design $\boldsymbol{x}^{\mathbf{o}}$ was at (t_1, t_2) = (2.0mm, 3.0mm). This start was a random choice close to the mid ranges of the design variables.

Step 1:

FE model is run for the values of ($\boldsymbol{x}^{\mathbf{o}}$). The objective function value (total deformation in zone #2), $f(\boldsymbol{x}^{\mathbf{o}})$ = 378mm.

Step 3:

Since this is the first iteration in the algorithm, the values of $\boldsymbol{x}^{\mathbf{o}}$ and $f(\boldsymbol{x}^{\mathbf{o}})$ are the best known (so far), so they are copied to $\boldsymbol{x}^{*}$ and $f(\boldsymbol{x}^{*})$

Step 5:

The mean values of the sampling distributions are initialized to coincide on $\boldsymbol{x}^{\mathbf{o}}$:

μ_1 = 2.0mm

μ_2 = 3.0mm

Using the results of the FE run for the values of ($\boldsymbol{x}^{\mathbf{o}}$), the relaxed crash mode mismatch metric (Equation 5.8) is calculated. The values came out to be:

$cmm_{11} = -0.15$

$cmm_{22} = 36.7$

Using the two-sided sigmoid membership functions (Fig. 6.2), membership values are calculated for the crash mismatch metrics:

cmm_{11} has membership values: 0.301, 0.689, 0.015, 0.000 in NH, NL, PL, PH respectively

cmm_{22} has membership values: 0.000, 0.000, 0.000, 1.000, in NH, NL, PL, PH respectively

<u>Steps 9 & 10:</u>

Application of the sampling adjustment rules is performed (step #9 and #10 in the algorithm):

R_1: activity value a = 0.301 (membership of cmm_{11} in NH), action is on the sampling distribution of t_1, by an amount:

$\tilde{a} = -(t_{1,\max} - t_{1,\min})/5 = -0.6$mm.

This adjusts μ_1 by $\bar{a} = 0.301 \times -0.6$, now $\mu_1 = 1.819$mm

This also adjusts σ_1 by $|\bar{a}|$, now $\sigma_1 = 0.181$mm

R_2: activity value is 0.689 (membership of cmm_{11} in NL), action is on the sampling distribution of t_1, by an amount:

$\tilde{a} = -(t_{1,\max} - t_{1,\min})/10 = -0.3$mm.

This adjusts μ_1 by $\bar{a} = 0.689 \times -0.3$, now $\mu_1 = 1.613$mm

This also adjusts σ_1 by $|\bar{a}|$, now $\sigma_1 = 0.387$mm

R_3: activity value is 0.015 (membership of cmm_{11} in PL), action is on the sampling distribution of t_1, by an amount:

$\tilde{a} = (t_{1,\max} - t_{1,\min})/10 = 0.3$mm.

This adjusts μ_1 by $\bar{a} = 0.015\times 0.3$, now $\mu_1 = 1.617$mm

This also adjusts σ_1 by $|\bar{a}|$, now $\sigma_1 = 0.391$mm

R_8: activity value is 1.000 (membership of cmm_{22} in PH), action is on the sampling distribution of t_2, by an amount:

$\tilde{a} = (t_{1,\max} - t_{1,\min})/5 = 0.6$mm.

This adjusts μ_2 by $\bar{a} = 1.000\times 0.6$, now $\mu_2 = 3.6$mm

This also adjusts σ_2 by $|\bar{a}|$, now $\sigma_2 = 0.6$mm

R_4, R_5, R_6, R_7 have activity values of almost zero (membership of cmm_{11} in PH, and memberships of membership of cmm_{22} in NH, NL, PL respectively), thus almost no effect on the sampling distributions.

Standard deviations are checked verses their minimum set values (step #12 in the algorithm), which is 0.3mm in the algorithm inputs. Both σ_1, σ_2 are already sufficiently large.

Step 12:

The sampling distributions are then used to generate 8 sample designs following a Gaussian distribution with the mean values (μ_1, μ_2) = (1.617, 3.6) and standard deviations (σ_1, σ_2) = (0.39, 0.6). Since the design variables may only take on discrete values, the generated samples are rounded off to the nearest discrete value of (t_1, t_2)

The samples generated in this instance of simulation were: (1.8, 3.2), (1.6, 4.0), (1.2, 3.8), (1.4, 3.6), (1.8, 3.2), (1.4, 3.4), (1.6, 3.6), (1.8, 3.4)

Best among the samples in this simulation of the algorithm was at (1.2, 3.8), which is:

t_1 = 1.2mm

t_2 = 3.8mm

$f(1.2, 3.8)$ = 2.04mm

Step 3:

The current design $\boldsymbol{x}^{\mathbf{o}}$ is now replaced by the values (1.2, 3.8), and since the objective function value f (1.2, 3.8) was better than the previously best known. The current best known design is also updated:

$\boldsymbol{x}^{\mathbf{o}} = (1.2, 3.8)$

$\boldsymbol{x}^{*} = \boldsymbol{x}^{\mathbf{o}}$

20 more iterations of the algorithm (similar to the steps described in this sub-section) are repeated, then the values of $\boldsymbol{x}^{*}$ and $f(\boldsymbol{x}^{*})$ are returned.

6.5.3 Performance Assessment of the Algorithm

It is worth noting that the size of the search space (total number of possible designs) for this optimization problem is 256, which is a small enough number to allow for exhaustive search (full enumeration of every design in the search space). While completely impractical for crashworthiness design problems that involve structures of realistic level of complexity, exhaustive search was performed for this problem to ensure the discovery of the global optimum and use it for assessment of the performance of the automated crash mode matching algorithm.

Exhaustive search revealed the global optimum for this problem to be at:

$t_1 = 1.2$mm

$t_2 = 4.0$mm

$f(1.2, 4.0) = 2.01$mm

As with all stochastic search-based algorithms, the global convergence property (section 6.4) only guarantees convergence to the global optimum after running the algorithm for a *sufficiently large* number of iterations. The algorithm does not necessarily produce the same result it is run. In order to assess the performance of the algorithm, 100 independent runs were performed, with different initial state of the random number generator. Summary of the results is provided in Table 6.2.

Out of 100 runs in this study, the automated crash mode matching algorithm was successful in attaining the exact optimum in 85 runs. Furthermore, 99 runs out of the 100 produced a result whose objective function value was within 10% of the optimum. Designs within 10% of the optimum (10% of 2.01mm deformation in zone #2 is approximately 2.2mm) had a crash mode that correctly matches the desirable crash mode from a qualitative sense. Designs encountered during exhaustive search that didn't match the desired crash mode had deformations in zone #2 in range between 300mm and 450mm.

As a comparison, the optimization problem was also addressed via Genetic Algorithm (GA), which is well known and popular stochastic based search (Goldberg 1989, Michalewiz, and Fogel, 2000). Parameter settings for tuning of the GA are listed in Table 6.3. All the parameters are typical values from the literature except the population size of 8, which is a small number for typical GA. It is noted however, that in typical GA application problems, the population size is usually less than 1% of the size of the search space. A population size of 8 is in fact 3.1% of the size of the search space. Also, for an appropriate comparison, the total number of objective function evaluations is set to the same value as the automated crash mode matching algorithm runs: 20 generations × 8 population size is 160 objective evaluations, which is the same as the 20 iterations, with 8 samples in each iteration for the automated crash mode matching algorithm.

Table 6.2. Summary of the Results of 100 Algorithm Runs

	Auto-CMM	**GA**
Number of runs successfully attaining the optimum	85	84
Number of runs successfully attaining a result within 10% of the optimum	99	89

Table 6.3. Genetic Algorithm Parameters used in Study

Number of Generations	20
Population Size	8
Crossover Probability	90%
Mutation Probability	5%
Crossover Type	Arithmetic / Heuristic
Mutation Type	Randomize within upper and lower bound

Results of the GA are displayed in Table 6.2. It is observed that for 100 independent runs, GA was successful in attaining the exact optimum 84 times. . Furthermore, 89 runs out of the 100 produced a result whose objective function value was within 10% of the optimum. Overall, this study shows a slight advantage of the automated crash mode matching algorithm compared to GA. In more complex problems, direct linking of GA to detailed FE models may not be feasible with available computational resources (for example, the case study presented in Chapter 8).

On a side note, exhaustive search of this problem revealed that the objective function doesn't have any local minima except the global optimum, so any hill-climber type optimization algorithm (Michalewiz, and Fogel, 2000) is guaranteed to find the optimum regardless of the starting point. Local minima however are a known issue in crashworthiness design problems even ones involving fairly simple structures (Chen 2001). This is the reason hill-climber or gradient based algorithms are rarely used for crashworthiness problems involving structures of realistic level of complexity, and were not included in this study.

6.6 Summary

This chapter presented the proposed crashworthiness optimization algorithm that employs automated crash mode matching to accelerate the discovery of good designs within a stochastic search framework. Global convergence properties of the algorithm are established, and a simple example is presented to demonstrate the algorithm performance from a practical point of view. The next chapters provide application studies of the proposed crashworthiness design methodology to larger scale vehicle structures.

CHAPTER 7

CASE STUDY 1: FRONT HALF-BODY VEHICLE MODEL

This chapter presents a case study in which the proposed methodology for structural crashworthiness design is examined. The study considers a simplified FE model of a vehicle with only few box-section members, undergoing a full-lap frontal crash against a rigid barrier. The proposed methodology is successful in attaining good designs while requiring moderate computational resources.

7.1 Problem Model

The model of the vehicle (Fig. 7.1) is set to simulate frontal crash conditions against a rigid barrier. The FE model has the following specifications:

- All main structural members are box-section
- The engine and power train are represented as a rigid box of mass 250 kg, connected to the engine mounting points via stiff beams.
- The rest of the vehicle mass (600 kg) is represented as a lumped mass connected to the structure via stiff beams.
- Crash speed is 15.6 m/s (35 mph)
- Coefficient of friction at the rigid barrier is 0.3
- Material model is elastic-plastic for mild steel

There are 4 continuous and 14 discrete design variables:

- h_1, b_1 [mm]: height and width of the box-section of upper rails and cross members (continuous in [50, 150]). These variables govern the dimensions of the box sections in zones 1 through 6, 13 and 14 as indicated in Fig. 7.1
- h_2, b_2 [mm]: height and width of the box-section of lower rails and cross members (continuous in [50, 150]). These variables govern the dimensions of the box sections in zones 9 through 12 as indicated in Fig. 7.1.
- Zones 7 and 8 are connectors between upper and lower members and have tapering cross section between them.
- t_1, …, t_{14} [mm]: sheet metal thickness of the box-section, for structural zones 1 through 14 as indicated in Fig.7.1 (discrete in {0.6, 0.8, 1.0, …, 4.2, 4.4}).

The design objective is to minimize the structural mass, subject to safety constraints on the passenger:

- f [kg]: structural weight, to be minimized
- $g_1 < 100$ [mm]: intrusion into passenger compartment, measured as the maximum relative deformation during the crash event between points A and B in Fig. 7.1
- $g_2 < 30$ [G]: maximum acceleration at passenger compartment during the crash event, measured at point A in Fig. 7.1

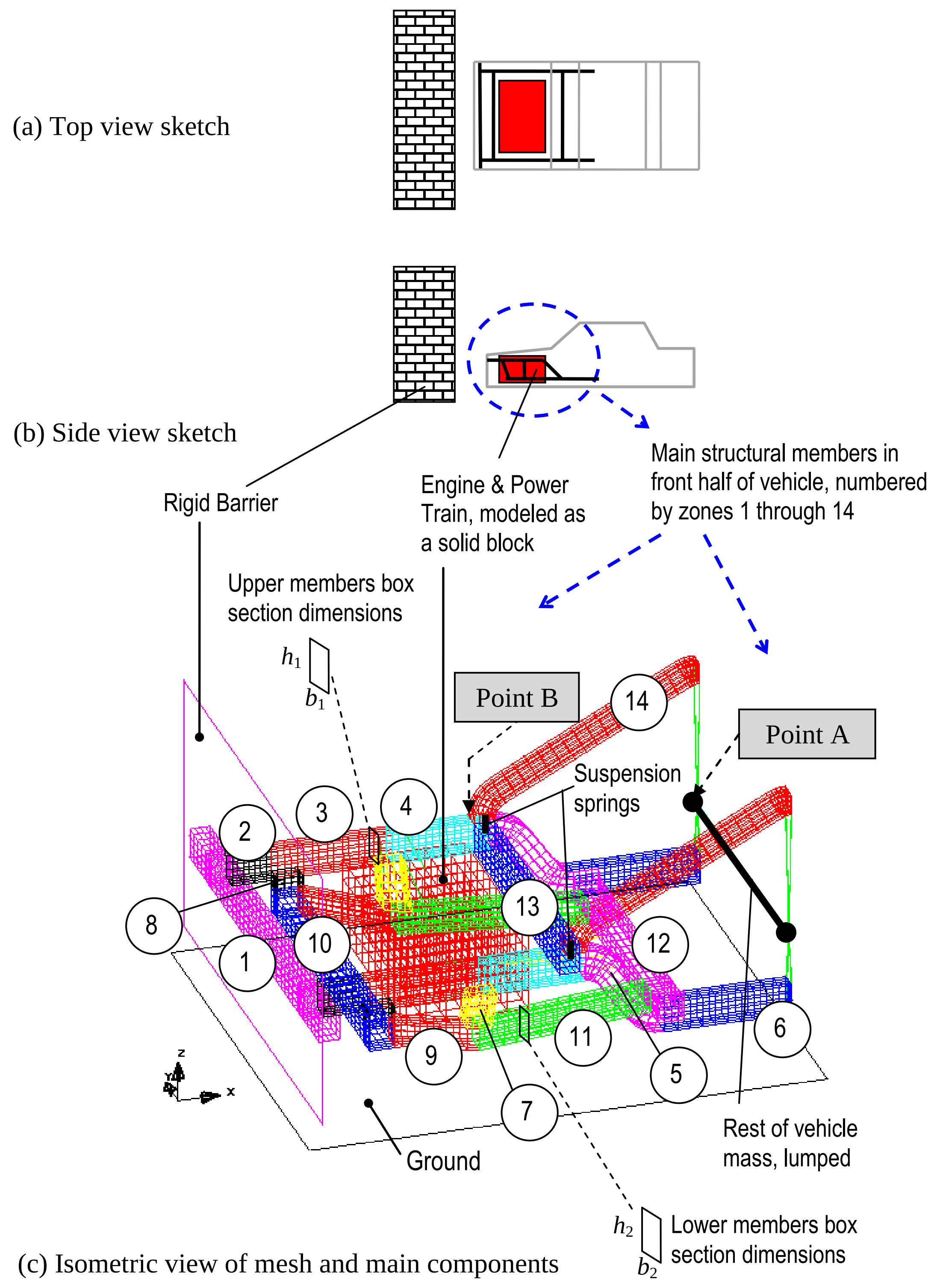

(a) Top view sketch
(b) Side view sketch
Rigid Barrier
Engine & Power Train, modeled as a solid block
Main structural members in front half of vehicle, numbered by zones 1 through 14
Upper members box section dimensions
h_1
b_1
Point B
Point A
14
Suspension springs
2
3
4
8
1
10
13
12
6
9
11
5
7
Ground
Rest of vehicle mass, lumped
h_2
b_2
Lower members box section dimensions
(c) Isometric view of mesh and main components

Fig. 7.1. FE model of front half of a vehicle subjected to full-overlap frontal crash.

7.2 Previous Optimization Attempts

This section summarizes previous optimization attempts for this case study problem. Running optimization over a response surface meta-model which was used to approximate the behavior of the FE model achieved little success. Limited success was achieved in in (Hamza and Saitou 2004c) via application of genetic algorithm (GA) (Goldberg 1989, Michalewicz 1996) directly to the FE model. The study in (Hamza and Saitou 2004c) also examined manual (non-automated) crash mode matching (with the desired crash mode identified via an EM model). The manual crash mode matching produced the best (so far known) result to the problem.

7.2.1 Optimization via Response Surface Models

As discussed in chapter 2, structural crashworthiness optimization via response surface models seems to be the most popular approach for automated design in practice. The approach follows 3 main steps:

1. Sampling of the design space via detailed FE model runs
2. Fitting of a response surface model over the sampled designs to form a meta-model
3. Running an optimization algorithm while using the meta model for the objectives and constraints estimation (instead of the detailed FE model)

In step #1, a standard orthogonal array L54 (Phadke 1993) was used for design of experiments (DOE) drawing of samples in design space. The L54 array allows for data fitting of up to 25 variables, with each variable being sampled at 3 different levels. Use of this array for sampling only requires running the detailed FE model for 54 sample designs (compared to a full factorial DOE of 3^{18} = 387420489 samples), but the higher order interactions between the variables cannot be observed.

Depending on the variables to column assignments (Phadke 1993) in the standard orthogonal array, there can be multiple instances of an orthogonal array (*i.e*. different values of the design variables in the set of 54 design samples). In this study, 2 different instances of the L54 orthogonal array were used. Full listing of the sample designs and the values of objective and constraints obtained from the detailed FE model is provided in Appendix B. The generated design samples are labeled sample set #1 and sample set #2

In step #2, 2nd order polynomial and Kriging were used to fit the samples data of each sample set, as well as all the generated samples. This allowed the construction of 6 different meta-models, listed in Table 7.1. The meta-models were labeled RSM1 through RSM6.

In step #3, genetic algorithm (GA) is used for the design optimization by running it while using the constructed meta-models for estimation of the objective and constraints. Since the objective and constraints estimation via the meta-models is very fast, this allowed a thorough GA run until full convergence of the GA. Parameters for the GA runs are listed in Table 7.2.

Table 7.1 Listing of the Constructed Meta-Models for Case Study 1

Meta-Model	Description
RSM1	2nd Order polynomial fitted over sample set #1
RSM2	2nd Order polynomial fitted over sample set #2
RSM3	2nd Order polynomial fitted over both sample set #1 and #2
RSM4	Kriging fitted over sample set #1
RSM5	Kriging fitted over sample set #2
RSM6	Kriging fitted over both sample set #1 and #2

Table 7.2. GA Parameters used in Optimization via Response Surface Models

Number of Generations	150
Population Size	80
Crossover Probability	90%
Mutation Probability	5%
Crossover Type	Arithmetic / Heuristic
Mutation Type	Randomize within upper and lower bound

GA runs are performed for each of the constructed meta-models. The optimum values for the design variables for each run are listed in Table 7.3. The detailed FE model is then used to check the objective and constraint values of the obtained designs. It is observed that none of the runs were successful in producing a feasible design (g_1 always violated). This result highlights the main drawback in optimization via response surface methods, which is the risk of obtaining invalid results if the constructed meta-model(s) is not sufficiently accurate.

Table 7.3 Results of Optimization via Response Surface Models

	RSM1	RSM2	RSM3	RSM4	RSM5	RSM6
h_1 [mm]	90.0	90.0	90.0	90.0	90.0	90.0
b_1 [mm]	54.3	58.4	57.9	50.0	55.3	56.3
h_2 [mm]	50.0	51.6	50.0	50.0	50.0	50.2
b_2 [mm]	70.1	70.0	76.1	70.0	70.0	70.8
t_1 [mm]	2.0	2.0	2.0	2.0	2.0	2.0
t_2 [mm]	1.8	2.0	2.0	1.8	1.8	1.8
t_3 [mm]	1.6	1.8	1.8	1.6	1.6	1.6
t_4 [mm]	2.6	3.0	2.6	2.6	3.0	2.6
t_5 [mm]	3.2	3.2	3.2	3.2	3.2	3.2
t_6 [mm]	3.0	3.0	3.0	2.8	3.0	3.2
t_7 [mm]	1.2	1.2	1.2	1.0	1.0	1.0
t_8 [mm]	2.2	2.2	2.2	2.0	2.2	2.2
t_9 [mm]	2.6	2.4	2.6	2.8	2.4	2.4
t_{10} [mm]	2.6	2.6	2.6	2.6	2.6	2.6
t_{11} [mm]	2.8	2.8	3.2	2.8	2.8	3.0
t_{12} [mm]	2.4	2.4	2.4	2.6	2.6	2.6
t_{13} [mm]	1.8	1.8	1.8	1.8	1.8	1.8
t_{14} [mm]	2.4	2.4	2.4	2.4	2.4	2.4
f [kg] (RSM estimate)	53.7	56.1	56.8	53.1	54.9	55.3
g_1 [mm] (≤ 100) (RSM estimate)	100.0	99.6	99.8	99.7	99.9	99.6
g_2 [G] (≤ 30) (RSM estimate)	26.3	29.2	28.3	26.5	27.9	28.3
f [kg] (detailed FE)	54.2	56.3	57.1	53.3	55.1	55.5
g_1 [mm] (≤ 100) (detailed FE)	148.5	161.6	146.4	158.6	172.4	154.6
g_2 [G] (≤ 30) (detailed FE)	27.1	30.4	29.9	26.7	28.1	26.8

Improving the prediction accuracy of a meta-model may be achieved by restricting the ranges of variation for the design variables, or by increasing the number of sample designs used to construct the meta-model. However, both options seem infeasible for this case study. Restricting the ranges of the variables eliminates regions of the design space and should not be done unless enough knowledge of the problem at hand suggests that the excluded regions do not contain optimal designs. Increasing the number of the sample designs to construct the model is impractical beyond a certain point. The meta-models that were fitted using 108 design samples (RSM3 and RSM6) had only slightly better accuracy (Table 7.3) than the models that were fitted using 54 samples (RSM1, RSM2, RSM5 and RSM5). With the number of design samples to construct the meta-model becoming as many as a few hundreds, the computational resources requirement would be fairly similar to linking an optimization algorithm directly to the detailed FE model, which is examined in the next sub-section.

7.2.2 Optimization via Genetic Algorithm

In this optimization attempt, genetic algorithm is directly linked to the detailed FE model. The GA parameters are listed in Table 7.4. It is noted that due to limitations on the available computational resources, the population size and number of generations for this GA run is smaller than what would typically be used. This GA run is essentially an attempt to discover a good design while using available computational resources for 500 detailed FE runs (population size of 50 × 10 generations).

The best obtained design by the GA run is listed in Table 7.5. The design is feasible (both constraints g_1 and g_2 are less than their maximum allowed values). However, neither constraint is close to its maximum allowed value, which would suggest the existence of better designs than the one discovered. However, the available computational resources (500 detailed FE runs) did not allow for finding better designs.

Table 7.4. GA Parameters for directly linked GA with the detailed FE model

Number of Generations	10
Population Size	50
Crossover Probability	90%
Mutation Probability	5%
Crossover Type	Arithmetic / Heuristic
Mutation Type	Randomize within upper and lower bound

Table 7.5 Best obtained design by GA

h_1 [mm]	114.0
b_1 [mm]	67.0
h_2 [mm]	69.0
b_2 [mm]	95.0
t_1 [mm]	2.2
t_2 [mm]	2.0
t_3 [mm]	1.8
t_4 [mm]	2.8
t_5 [mm]	3.0
t_6 [mm]	3.0
t_7 [mm]	1.2
t_8 [mm]	2.2
t_9 [mm]	2.6
t_{10} [mm]	2.8
t_{11} [mm]	3.0
t_{12} [mm]	2.4
t_{13} [mm]	2.0
t_{14} [mm]	2.2
f [kg]	73.1
g_1 [mm] ($\leq$ 100)	62.0
g_2 [G] ($\leq$ 30)	25.9

7.2.3 Optimization via Manual Crash Mode Matching

A study was performed in (Hamza and Saitou 2004c) which was aimed at testing the hypothesis that manipulating the crash mode of the structure can be an effective way of attaining good designs. In this study, a thorough optimization run via GA was performed on an EM model in order to identify the desirable crash mode in the qualitative sense. The design variables were then adjusted manually (non-automated) in order to adjust the crash mode to qualitatively match the desirable crash mode. This led to the discovery of the currently best known design for the problem in this case study. Summary of the study in (Hamza and Saitou 2004c) is provided in this sub-section.

Construction of the EM model of the problem (Fig. 7.2) was performed using the developed computer software. Optimization is then performed by running GA while using the EM model (instead of the detailed FE) for the estimation of objectives and constraints. The GA parameters are listed in Table 7.6. The result of the GA run on EM model converged to the values of the design variables listed in Table 7.7. The crash mode of the obtained design is shown in Fig. 7.3 and is designated as the desirable crash mode.

Table 7.6. GA Parameters for GA linked with EM model

Number of Generations	50
Population Size	100
Crossover Probability	90%
Mutation Probability	5%
Crossover Type	Arithmetic / Heuristic
Mutation Type	Randomize within upper and lower bound

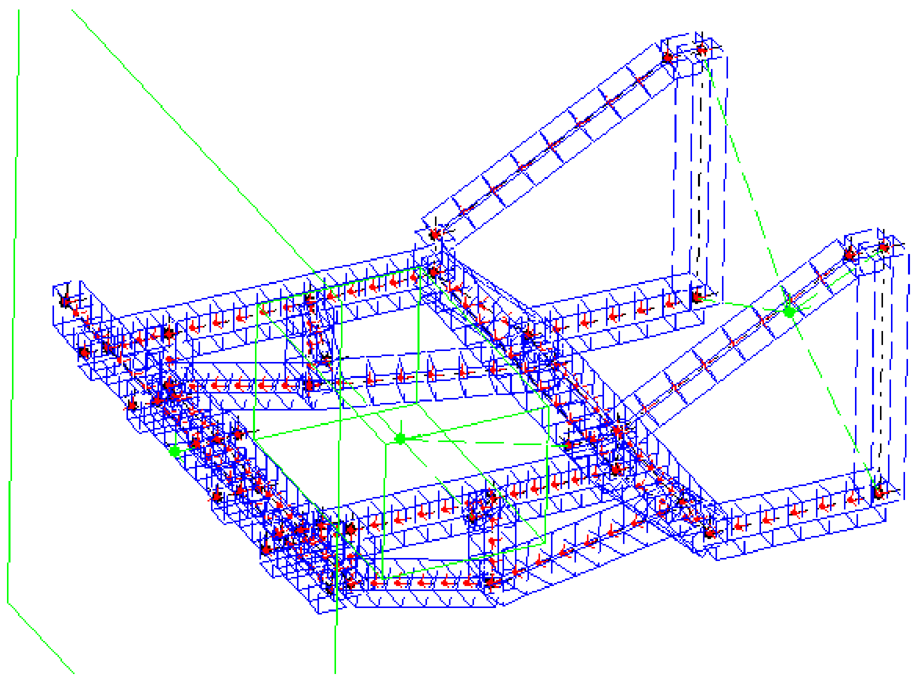
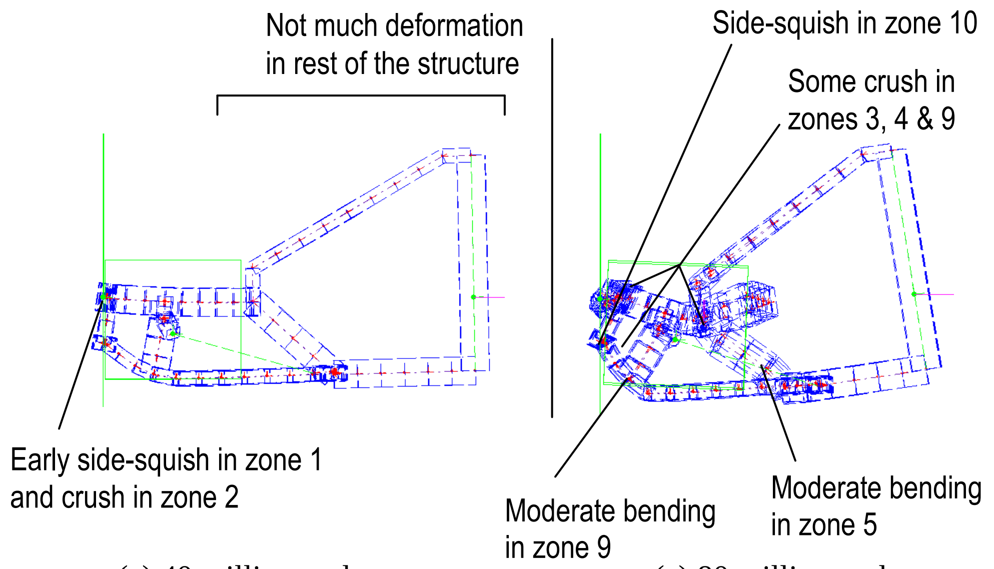


Fig. 7.2. EM model of front half of a vehicle subjected to full-overlap frontal crash


Not much deformation in rest of the structure
Side-squish in zone 10
Some crush in zones 3, 4 & 9
Early side-squish in zone 1 and crush in zone 2
Moderate bending in zone 9
Moderate bending in zone 5

(a) 40 millisecond

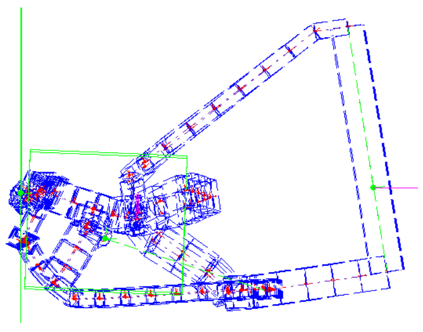
(a) 80 millisecond

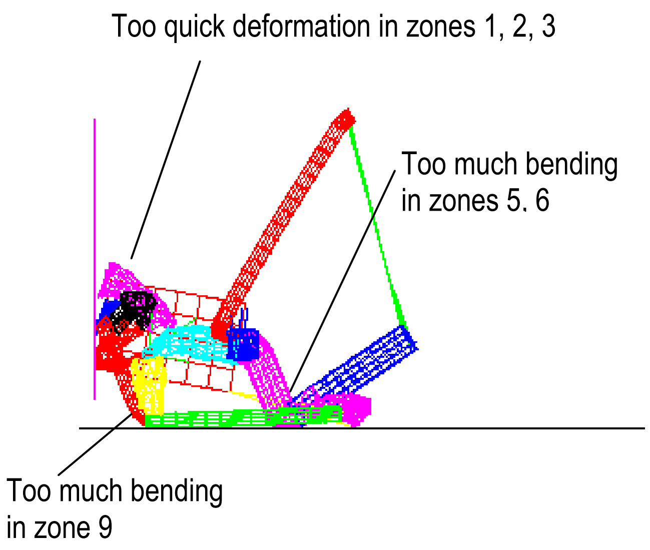
Fig. 7.3. Desirable crash mode as identified via GA linked with EM model

Table 7.7 Design obtained by running GA linked with EM model

h_1 [mm]	99.0
b_1 [mm]	50.0
h_2 [mm]	50.0
b_2 [mm]	77.0
t_1 [mm]	0.8
t_2 [mm]	2.6
t_3 [mm]	2.2
t_4 [mm]	1.8
t_5 [mm]	3.0
t_6 [mm]	3.0
t_7 [mm]	1.0
t_8 [mm]	2.0
t_9 [mm]	2.6
t_{10} [mm]	4.4
t_{11} [mm]	3.2
t_{12} [mm]	2.0
t_{13} [mm]	3.0
t_{14} [mm]	2.0
f [kg] – EM estimate	62.0
g_1 [mm] (≤ 100) – EM estimate	95.0
g_2 [G] (≤ 30) – EM estimate	24.0
f [kg] – detailed FE	58.0
g_1 [mm] (≤ 100) – detailed FE	254.0
g_2 [G] (≤ 30) – detailed FE	25.0

Highlights of the qualitatively observed crash mode are shown in Fig. 7.3. Within the first 40 milliseconds of the crash event, most of the structural deformation occurs in the form of side-squish in the bumper (zone 1) and axial crush in the front crush module (zone 2). Towards the end of the crash event (80 milliseconds), the engine block squeezes the front cross bar (zone 10), and moderate amounts of axial crush and bending occur in 3, 4, 5 and 9.




Too quick deformation in zones 1, 2, 3
Too much bending in zones 5, 6
Too much bending in zone 9

(a) EM Model (b) Detailed FE Model

Fig. 7.4. Crash Mode of design identified via GA linked with EM model: (a) EM Model, (b) Detailed FE Model

However, testing the values of the design variables via a detailed FE model produced an infeasible design (constraint g_1 violated), with a qualitatively-different crash mode (Fig. 7.4) as observed from the animation time history in FE model compared with the EM. It was hypotheses that adjusting the design variables in order to attain the desirable crash mode should improve the crashworthiness performance. This design variables adjustment (Table 7.8, adjustments shown as a shaded cells in the table) was performed using qualitative observation of the crash mode in the detailed FE model (Fig. 7.5).

Steps for manually adjusting the crash mode are listed as follows:

- Iteration 1: it was observed in the CM of the EM that the frontal zones 1, 2, 3 were deforming too quickly, so the sheet thickness was increased in these zones. Also, excessive bending was occurring at the rear, so thickness was increased in zones 4, 5, 6, 14. The front cross bar was too stiff and was not collapsing properly, so the thickness in zone 10 and the width of lower member were reduced. Also, thickness was increased at the vertical member connecting the two rails (zone 7). The resulting design of iteration 1 had a much better cabin intrusion performance of 135 mm compared to 254 mm, but was still short of the target of 100 mm.

- Iteration 2: in an attempt to further prevent too early collapse of the front zones, the thickness in zones 2, 3, 4, 7, 9 was increased, but this resulted in zone 3 not deforming at all, and the resulting crash mode had bad performance, so iteration 2 was abandoned.

- Iteration 3: Prevention of too early collapse of the front structural zones was next attempted by increasing the width of upper structural members and height of lower structural members. Height of upper structural members was slightly reduced. Sheet thickness in zones 1, 4, 7, 9, 12 was increased, while sheet thickness in zones 10, 11 was reduced. The resulting design did achieve an acceptable cabin intrusion of 81 mm, but the maximum acceleration became 35G, which was slightly above the target of 30G.

Table 7.8 Steps for Manual Crash Mode Matching

	Iterations of CM Matching						
	1	2	3	4	5	6	7
h_1 [mm]	100.0	100.0	90.0	90.0	90.0	90.0	90.0
b_1 [mm]	50.0	50.0	80.0	80.0	80.0	80.0	80.0
h_2 [mm]	50.0	50.0	60.0	60.0	60.0	60.0	60.0
b_2 [mm]	70.0	70.0	70.0	70.0	50.0	50.0	50.0
t_1 [mm]	2.8	2.8	3.2	3.2	3.2	3.2	3.2
t_2 [mm]	2.8	3.2	2.8	2.8	2.8	2.4	2.4
t_3 [mm]	2.4	2.8	2.4	2.4	2.0	2.0	2.0
t_4 [mm]	2.4	2.8	2.8	2.8	2.8	2.8	2.8
t_5 [mm]	4.2	4.2	4.2	4.2	4.2	4.2	4.2
t_6 [mm]	3.2	3.6	3.2	3.2	3.2	3.2	2.8
t_7 [mm]	2.0	2.4	2.4	2.8	2.8	2.8	3.2
t_8 [mm]	2.0	2.0	2.0	2.0	2.0	2.0	2.0
t_9 [mm]	2.6	2.4	2.4	2.4	2.4	2.4	2.4
t_{10} [mm]	3.6	3.6	3.2	3.2	2.8	2.6	2.6
t_{11} [mm]	3.2	3.2	2.8	2.8	1.6	1.6	1.2
t_{12} [mm]	2.0	2.0	2.8	2.8	2.8	2.8	2.8
t_{13}[mm]	3.0	3.0	3.0	1.8	1.8	1.8	1.8
t_{14} [mm]	2.2	2.2	2.2	2.2	2.2	2.2	2.2
f [kg]	70.0	70.0	72.0	72.0	70.0	68.0	66.9
g_1 [mm]	135.0	300.0	81.0	74.0	72.0	74.0	76.0
g_2 [G]	29.0	31.0	35.0	34.0	32.0	30.6	29.4

- Iteration 4: the slight CM mismatch between iteration 3 and the target CM identified by the optimal EM (Fig. 7.3) seemed to be apparent at zone 9, which was not axially deforming but got bent at the connection to zones 7 and 11. An attempted remedy was to increase the sheet thickness in zone 7. Also, it was observed that Zone 13 was hardly deforming at all, so sheet thickness in it was reduced. The resulting design more closely resembled the target CM and performed better than iteration 3. However, the acceleration level was at 34 g was still above the target of 30 g.

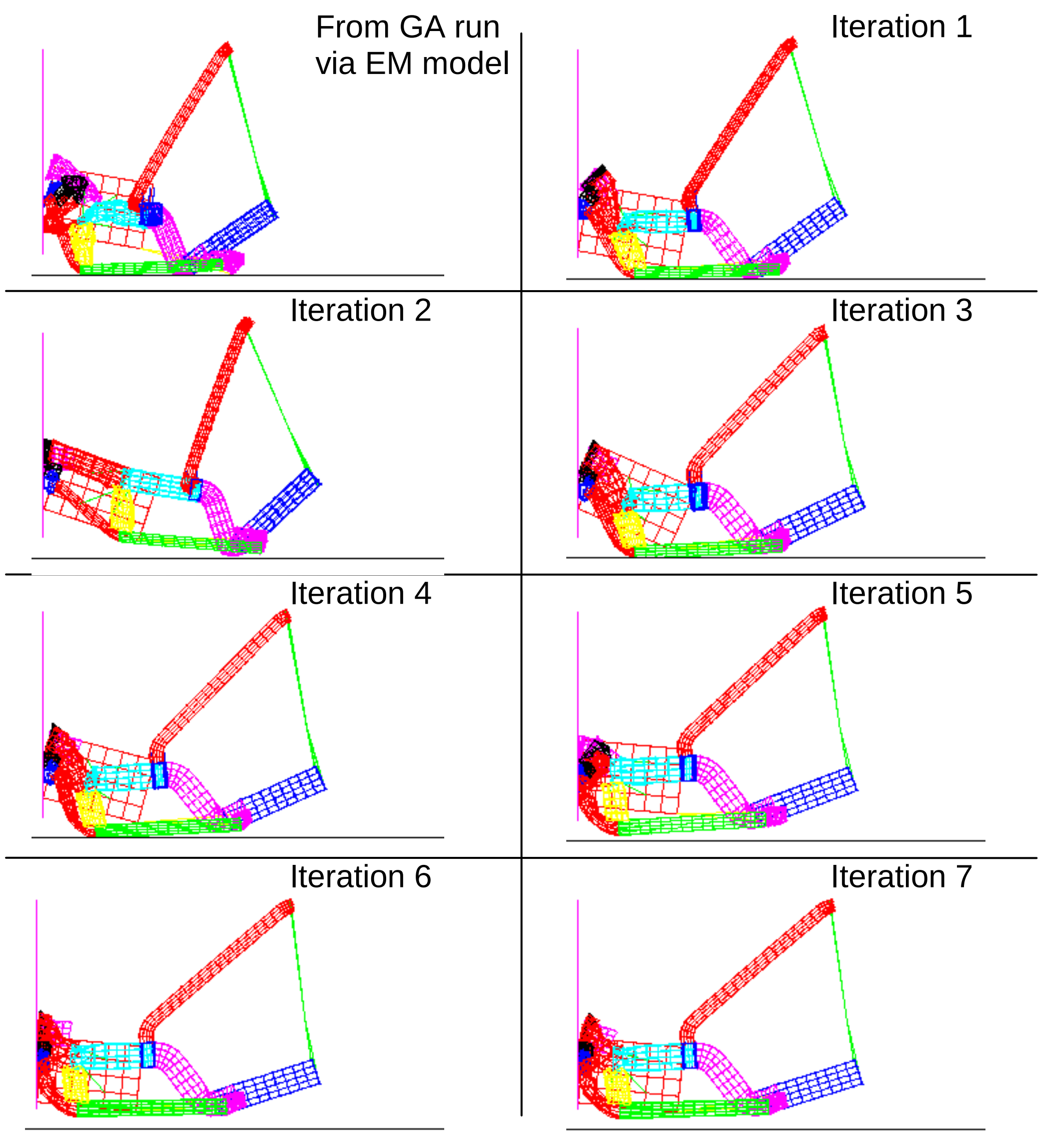


Fig. 7.5 Steps for Manual Crash Mode Matching

- Iterations 5, 6 and 7: The crash mode of iteration 4 was qualitatively close to the desired. This suggested that the acceleration levels that were higher than the target values may be due some zones in the line(s) of force being excessively stiff. Selective reductions in the sheet thickness in zones 2, 3, 6, 7, 10, 11 successfully brought the acceleration level within the desired limits without violating the constraint on allowable deformation.

The final design obtained via manual crash mode matching is currently the best known design for the problem in this case study.

7.3 Optimization via Proposed Methodology

This section discusses the application of the proposed methodology for automated crashworthiness design to the case study problem. Steps for applying the automated crash mode matching algorithm are presented.

7.3.1 Inputs to the Automated Crash Mode Matching Algorithm

Category #1 of Inputs: Objectives, Constrains & Design Variables:

This is the same as the problem formulation discussed in section 7.1. The object f is to minimize the structural weight, subject to the constraints g_1, g_2 on deformation and acceleration, by adjusting the values of the design variables h_1, b_1, h_2, b_2, t_1, …, t_{14}

Category #2 of Inputs: Desirable Crash Mode:

With 14 defined structural zones, with up to 3 types of deformation (axial, bending and side-squish) in each zone, there could be up to 42 crash mismatch metrics cmm_{ij}. However, not all zones experience significant amounts of all deformation types. Furthermore deformation (and energy absorption) in some zones can have more significant influence on the crashworthiness performance than in other zones. For example, the upper and lower rails (zones 2, 3, 9 and 11) have more influence on the crashworthiness than the connectors and cross-members (zones 7, 8, 12, 13, 14). Hence, selection of the significant crash mismatch metrics cmm_{ij} was performed. The select mismatch metrics, which will be used in the fuzzy rules for adjusting the design space sampling, were termed (δ_1, δ_2, …, δ_{19}) and are listed in Table 7.9. Reasoning for the selection is provided as follows:

- All zones except zone 1 (bumper) and zone 10 (front lower cross bar) do not experience any significant side-squishing. Zones 1 and 10 (δ_1, δ_2) do not experience any significant axial or bending deformation.
- Very little deformation occurs in zones 4 and 6 (δ_{13}, δ_{14}), so the axial and bending values of the crash mismatch metric were combined
- Zones 7, 8, 12, 13 and 14 (δ_{15}, δ_{16}, δ_{17}, δ_{18}, δ_{19}) are cross bars and connectors with less contribution to absorbing the crash energy, so the axial and bending values of the crash mismatch metric were combined

Values for the compact representation of the desirable crash mode (t_0, d) were obtained from the EM model that was optimized via GA in section 7.2.3. Estimation of (t_0, d) is illustrated in Fig. 7.6. The d value is taken as the steady state value averaged over the last 5% of time of the crash event. The t_0 value is taken as the time at which the deformation in the zone reaches 50% of the d value. The values for this case study are listed in Table 7.10.

Table 7.9 Crash Mode Mismatch Metrics in Case Study 1

Symbol	Value	Description
δ_1	cmm_{31}	Side-squish in zone 1
δ_2	cmm_{310}	Side-squish in zone 10
δ_3	cmm_{12}	Axial crush in zone 2
δ_4	cmm_{22}	Bending in zone 2
δ_5	cmm_{13}	Axial crush in zone 3
δ_6	cmm_{23}	Bending in zone 3
δ_7	cmm_{15}	Axial crush in zone 5
δ_8	cmm_{25}	Bending in zone 5
δ_9	cmm_{19}	Axial crush in zone 9
δ_{10}	cmm_{29}	Bending in zone 9
δ_{11}	cmm_{111}	Axial crush in zone 11
δ_{12}	cmm_{211}	Bending in zone 11
δ_{13}	$0.5\times(cmm_{14}+cmm_{24})$	Combined axial crush and bending in zone 4
δ_{14}	$0.5\times(cmm_{16}+cmm_{26})$	Combined axial crush and bending in zone 6
δ_{15}	$0.5\times(cmm_{17}+cmm_{27})$	Combined axial crush and bending in zone 7
δ_{16}	$0.5\times(cmm_{18}+cmm_{28})$	Combined axial crush and bending in zone 8
δ_{17}	$0.5\times(cmm_{112}+cmm_{212})$	Combined axial crush and bending in zone 12
δ_{18}	$0.5\times(cmm_{113}+cmm_{213})$	Combined axial crush and bending in zone 13
δ_{19}	$0.5\times(cmm_{114}+cmm_{214})$	Combined axial crush and bending in zone 14

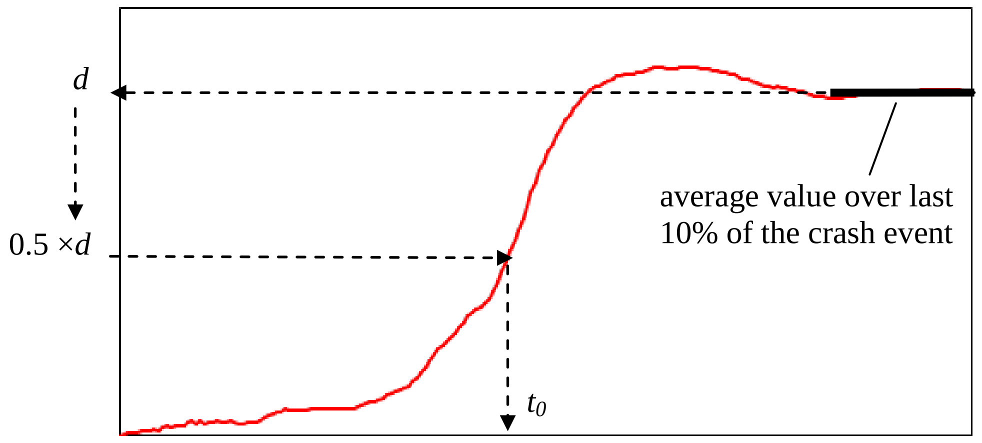


Fig. 7.6. Estimation of (t_0, d) values for a crash mode

Table 7.10 Compact representation values for the desired crash mode

Crash Mode	Desired Values	
	t_0 (ms)	d (mm or rad)
cm^*_{31}	14.0	85.0 mm
cm^*_{310}	16.5	70.0 mm
cm^*_{12}	12.0	250.0 mm
cm^*_{22}	28.0	1.2 rad
cm^*_{13}	30.0	350.0 mm
cm^*_{23}	30.0	1.2 rad
cm^*_{15}	48.0	100.0 mm
cm^*_{25}	48.0	0.8 rad
cm^*_{19}	30.0	100.0 mm
cm^*_{29}	30.0	1.4 rad
cm^*_{111}	25.0	200.0 mm
cm^*_{211}	36.0	0.7 rad
cm^*_{14}	60.0	5.0 mm
cm^*_{24}	60.0	0.1 rad
cm^*_{16}	60.0	5.0 mm
cm^*_{26}	60.0	0.1 rad
cm^*_{17}	50.0	5.0 mm
cm^*_{27}	50.0	0.1 rad
cm^*_{18}	27.0	5.0 mm
cm^*_{28}	27.0	0.1 rad
cm^*_{112}	40.0	5.0 mm
cm^*_{212}	40.0	0.2 rad
cm^*_{113}	48.0	5.0 mm
cm^*_{213}	48.0	0.2 rad
cm^*_{114}	52.0	10.0 mm
cm^*_{214}	52.0	0.7 rad

Category #3 of Inputs: Sampling Distribution Adjustment Rules:

Full listing of all the fuzzy rules in this case study is provided in tabulated form in Appendix C. A total of 238 fuzzy rules were defined . The rules were defined based on simple heuristics:

- If the crash mode mismatch metric for a zone/deformation type is positive (more deformation than desired), then the sampling distribution of the design variables that are perceived to affect the zone in question is adjusted to favor increasing the value of the design variable (make the zone stronger)
- If the crash mode mismatch metric for a zone/deformation type is negative (less deformation than desired), then the sampling distribution of the design variables that are perceived to affect the zone in question is adjusted to favor decreasing the value of the design variables (make the zone less stiff)

Example fuzzy rules in this study are:

If δ_1 is NH, **then** $\tilde{a}$ = -0.4, i_a = 5

(i_a = 5 means this rule adjusts sampling distributions on t_1)

If δ_1 is PL, **then** $\tilde{a}$ = +0.2, i_a = 5

(i_a = 5 means this rule adjusts sampling distributions on t_1)

If δ_3 is NL and δ_4 is PL, **then** $\tilde{a}$ = +6.0, i_a = 1

(i_a = 1 means this rule adjusts sampling distributions on h_1)

<u>Category #4 of Inputs: Tuning Parameters for the Algorithm:</u>

The number of iterations and number of samples per iteration were chosen such that the total number of detailed FE runs would be within 50 runs:

The number of samples per iteration was chosen as *nIterSamples* = 6

The total number of iterations was chosen as *nIter* = 8

The minimum value for standard deviation of the sampling distributions was set within 5% to 10% of the ranges of variation for each variable:

Minimum value for the standard deviations on the sampling distributions was chosen as σ_{min} = 0.2mm for the variables (t_1, …, t_{14})

Minimum value for the standard deviations on the sampling distributions was chosen as σ_{min} = 2.0mm for the variables (h_1, b_1, h_2, b_2)

7.3.2 Results of the Automated Crash Mode Matching Algorithm

Four independent runs of the automated crash mode matching algorithm were performed. In two of the runs, the initial design was the same starting point as manual crash mode matching (Table 7.7). This starting point (Start Pt#1) represents a light weight structure that violates the constraint on allowed deformation (g_1) The two other runs of the algorithm were started at some overly stiff design. . The second starting point (Start Pt#2) represents a heavy weight structure that violates the constraint on allowed acceleration (g_2). Both starting points are listed in Table 7.11. Crash mode plots for each start point are shown in Fig. 7.7 and Fig 7.8 respectively.

All of the four runs of the automated crash mode matching algorithm were successful in attaining feasible designs. The best and worst designs (in terms of objective function value) among the four runs are listed in Table 7.11, and their crash mode plots are shown in 7.9 and Fig 7.10 respectively.

Table 7.11 Summary of Automated Crash Mode Matching Algorithm Runs

	Start Pt#1	Start Pt#2	Auto CM – Best	Auto CM – Worst
h_1 [mm]	99.0	120.0	83.0	94.0
b_1 [mm]	50.0	80.0	77.0	77.0
h_2 [mm]	50.0	90.0	54.0	68.0
b_2 [mm]	77.0	100.0	56.0	105.0
t_1 [mm]	0.8	3.2	2.6	3.2
t_2 [mm]	2.6	1.4	1.6	2.2
t_3 [mm]	2.2	2.0	2.2	2.0
t_4 [mm]	1.8	3.0	4.6	3.4
t_5 [mm]	3.0	3.0	4.4	3.0
t_6 [mm]	3.0	3.0	3.2	2.6
t_7 [mm]	1.0	2.0	3.4	2.0
t_8 [mm]	2.0	2.4	1.6	1.6
t_9 [mm]	2.6	2.4	2.2	1.8
t_{10} [mm]	4.4	2.4	2.6	3.2
t_{11} [mm]	3.2	3.2	1.8	1.8
t_{12} [mm]	2.0	3.2	3.0	3.0
t_{13} [mm]	3.0	2.0	2.0	0.6
t_{14} [mm]	2.0	2.0	2.2	2.0
f [kg]	58.0	89.0	68.8	72.6
g_1 [mm]	254.0	87.1	97.0	89.0
g_2 [G]	25.0	35.8	26.8	27.4

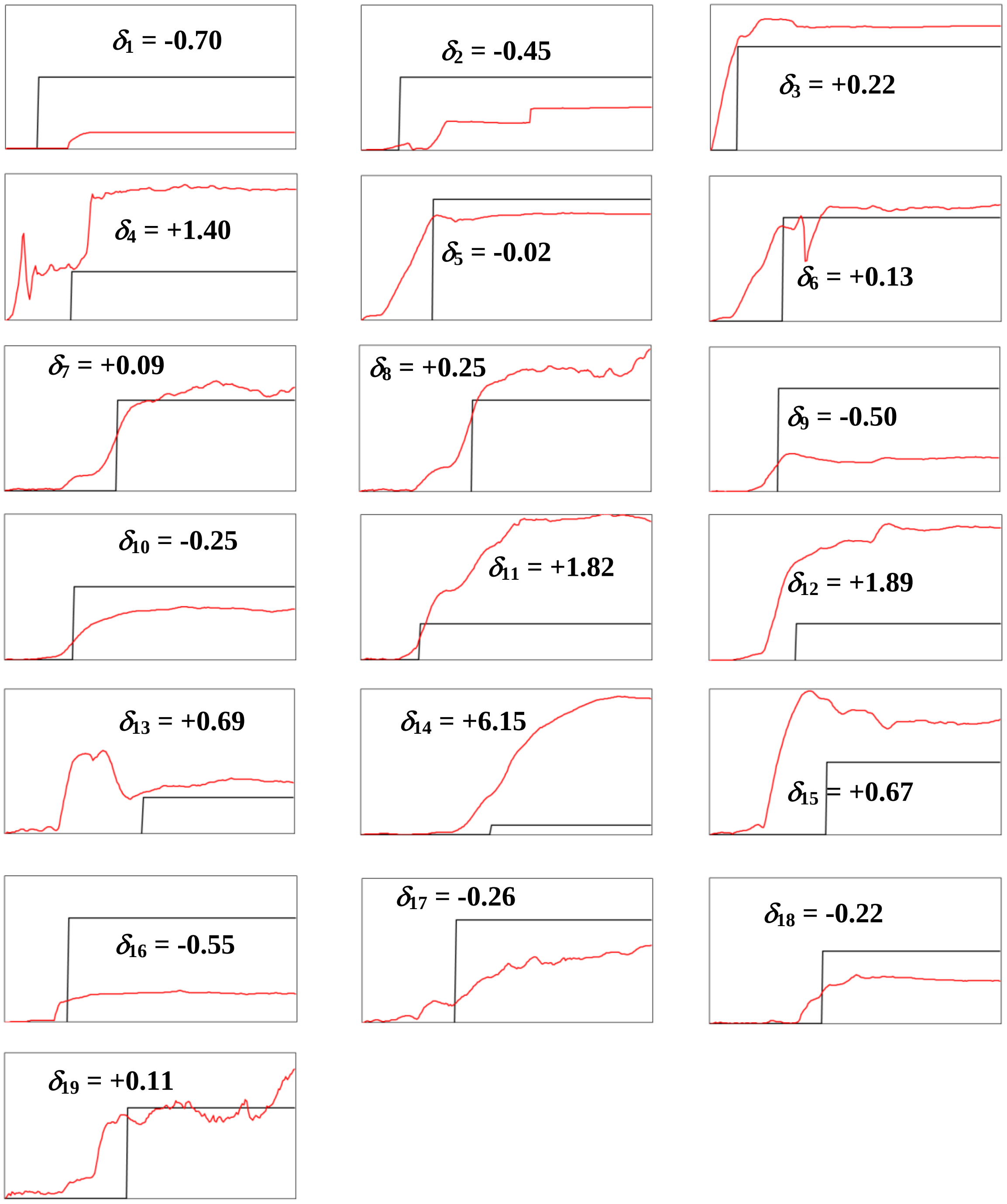


Fig. 7.7 Crash Mode Plot for Start Pt#1: Step function curves represent the desirable crash mode. The actual recorded crash mode time series plot is superimposed, and the normalized error integral value (δ) is stated

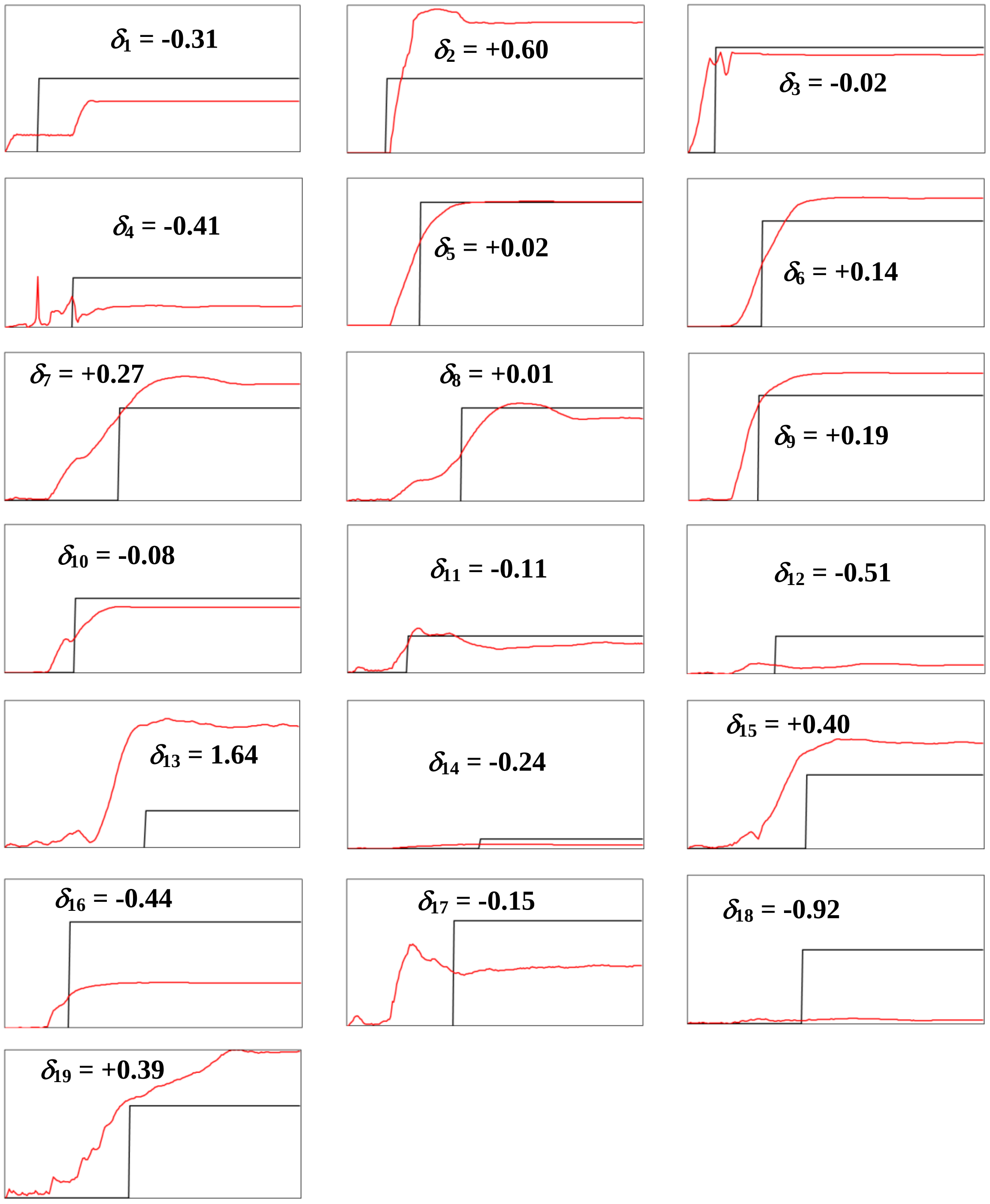


Fig. 7.8 Crash Mode Plot for Start Pt#2: Step function curves represent the desirable crash mode. The actual recorded crash mode time series plot is superimposed, and the normalized error integral value (δ) is stated

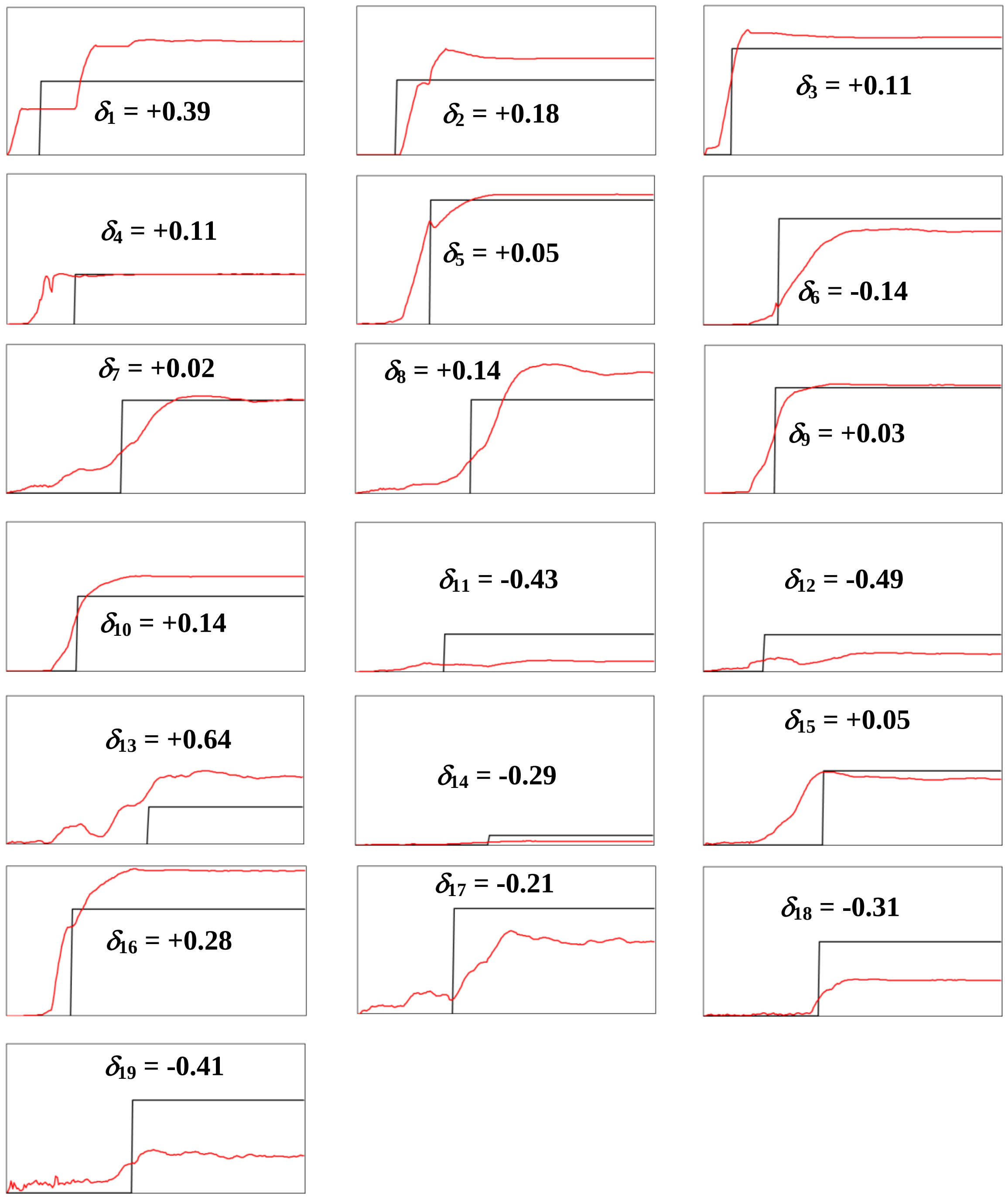


Fig. 7.9 Crash Mode Plot for best run of the automated crash mode matching algorithm. Step function curves represent the desirable crash mode. The actual recorded crash mode time series plot is superimposed, and the normalized error integral value (*δ*) is stated

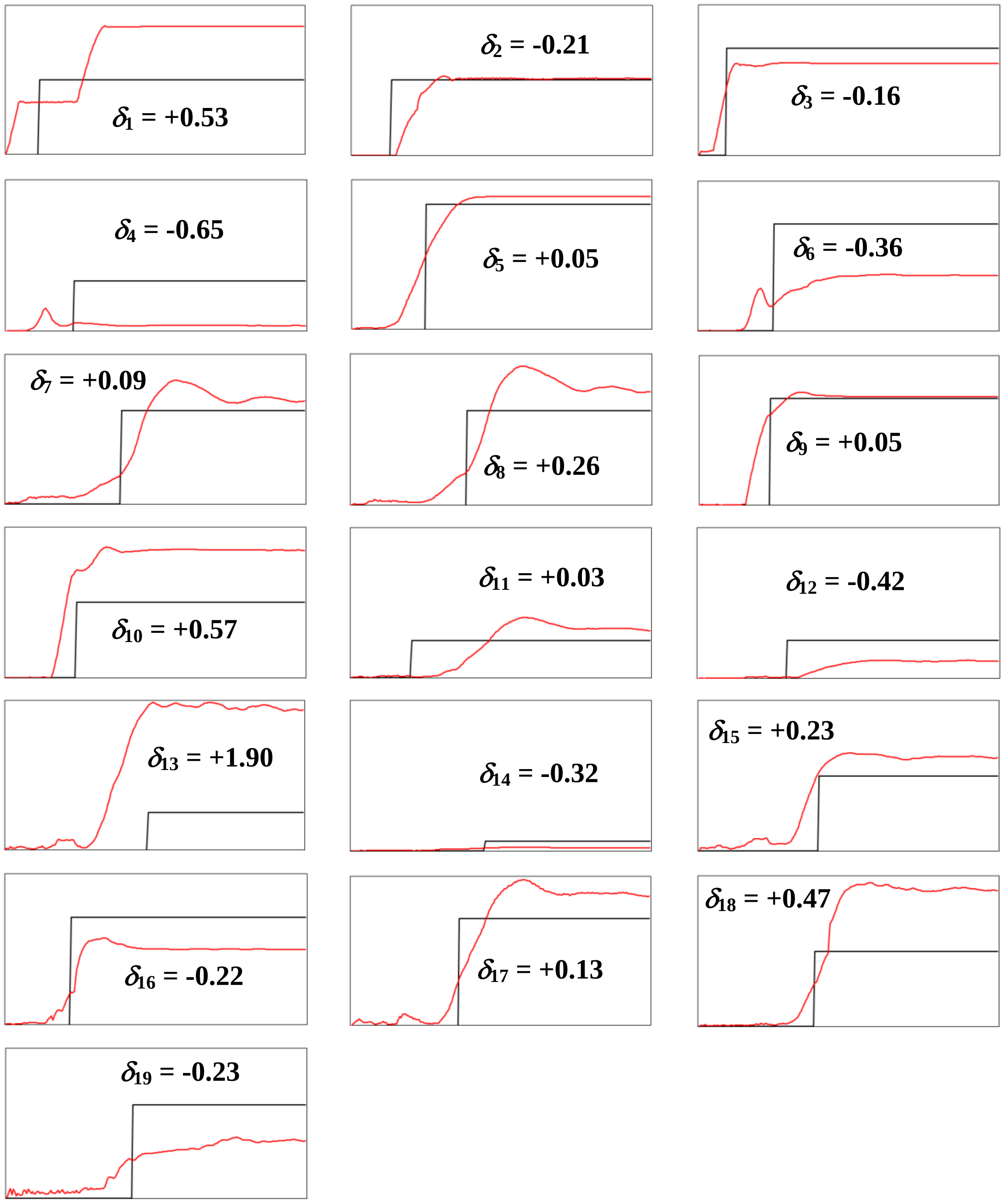


Fig. 7.10 Crash Mode Plot for worst run of the automated crash mode matching algorithm. Step function curves represent the desirable crash mode. The actual recorded crash mode time series plot is superimposed, and the normalized error integral value (δ) is stated

7.4 Summary of Results

Comparison of results for various optimization attempts for this case study is provided in Table 7.12, and graphically in Fig. 7.11. Optimization via response surface models performed in section 7.2.1 (Table 7.3) did not yield any feasible designs and thus is not included in the comparison. The best known design for this problem is the one attained via manual crash mode matching (convenient practice in automotive industry), however the manual crash mode matching is more of an art rather than an automated design approach.

All of the four runs that were performed via the automated crash mode matching algorithm were successful in attaining feasible designs. The best among the results of the automated crash mode matching algorithm is within 2% in objective function value from the best know result to this case study problem. The worst among the results of the automated crash mode matching algorithm was still better than the result obtained via GA. Total computational resources for the automated crash mode matching algorithm, however is less the 25% of that required for GA.

Table 7.12. Case Study 1: Summary of Results

	GA	Manual CM	Auto CM – Best	Auto CM – Worst
h_1 [mm]	114.0	88.0	83.0	94.0
b_1 [mm]	67.0	80.0	77.0	77.0
h_2 [mm]	69.0	60.0	54.0	68.0
b_2 [mm]	95.0	50.0	56.0	105.0
t_1 [mm]	2.2	3.2	2.6	3.2
t_2 [mm]	2.0	2.4	1.6	2.2
t_3 [mm]	1.8	2.0	2.2	2.0
t_4 [mm]	2.8	2.8	4.6	3.4
t_5 [mm]	3.0	4.2	4.4	3.0
t_6 [mm]	3.0	2.8	3.2	2.6
t_7 [mm]	1.2	3.2	3.4	2.0
t_8 [mm]	2.2	2.0	1.6	1.6
t_9 [mm]	2.6	2.4	2.2	1.8
t_{10} [mm]	2.8	2.6	2.6	3.2
t_{11} [mm]	3.0	1.2	1.8	1.8
t_{12} [mm]	2.4	2.8	3.0	3.0
t_{13} [mm]	2.0	1.8	2.0	0.6
t_{14} [mm]	2.2	2.2	2.2	2.0
f [kg]	73.1	66.9	68.8	72.6
g_1 [mm]	62.0	76.0	97.0	89.0
g_2 [G]	25.9	29.4	26.8	27.4
# FE runs	500	10	50	50
# EM runs	–	500	500	500
Comp. time [hour]	350	55	75	75

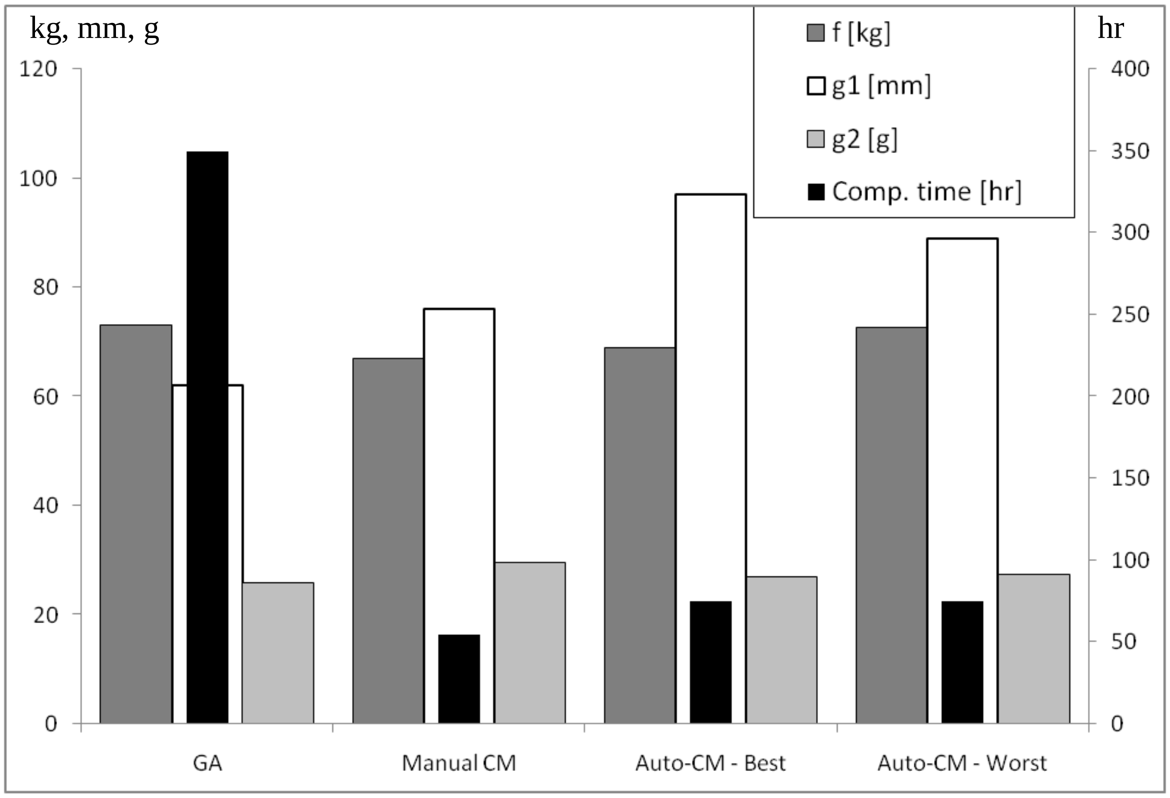


Fig. 7.11 Case Study 1: Summary of Results

CHAPTER 8

CASE STUDY 2: FULL VEHICLE MODEL

This chapter presents a case study of a detailed FE vehicle model subjected to offset frontal crash against a deformable barrier. A baseline model of a real vehicle is used in the study. Data about dimensions, deformations and acceleration is only listed in normalized form throughout this study[†]. The objective of this case study is mainly to show the success of the automated crash mode matching algorithm works when linked to a fully detailed vehicle model.

8.1 Problem Model

A detailed FE model of the vehicle and offset deformable barrier is shown in Fig. 8.1. The model has a total of more than half-million elements, mostly shell elements to model the sheet metal structural components. The model also includes solid elements for the bulky/rigid components such as the power-train components.

The problem is formulated as an unconstrained, multi-objective problem with the objectives being the minimization of the maximum deformation and acceleration.

[†] Due to propriety issues, the normalization scaling constants cannot be disclosed

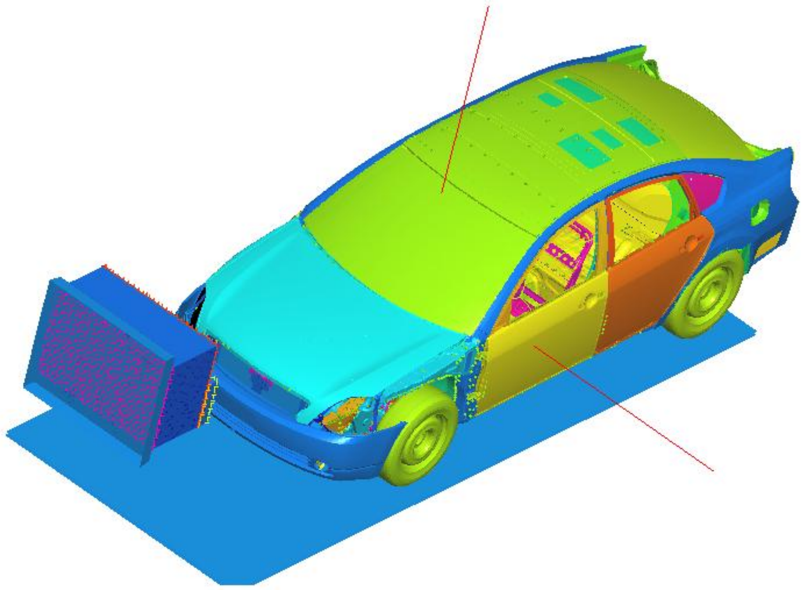


Fig. 8.1 Detailed FE model of a vehicle subjected to offset frontal crash against a deformable barrier

- f_1 is the normalized maximum deformation, to be minimized
- f_2 is the normalized maximum acceleration, to be minimized

A total of 49 sheet metal components were selected for exploring the effects of design adjustments. Some sheet metal components that build up the same structural member (or part of it) were grouped together and set to change according to one design variable. Thus, the thickness of the 49 sheet metal components was governed by 12 design variables (x_1, …, x_{12}). All the variables are discrete since they correspond to the sheet metal thickness. Structural weight was not considered a constraint or an objective in this study

Due to propriety issues, the values of the design variables will only be listed in some normalized form, with a value of 1.0 being the baseline. Each variable has up to 7 possible values, which would be: $x_1, \ldots, x_{12} \in \{0.85, 0.9, 0.95, 1.0, 1.05, 1.1, 1.15\}$.

8.2 Exploration of the Design Space

Computational resources required for performing one detailed FE run for this problem are at an average of 200 hours of CPU time. Even with available parallel computing resources at the University of Michigan Center for Advanced computing, the lengthy computation time for running the FE model made it prohibitive to use GA directly linked with the detailed FE model (as was done in case study 1 in section 7.2.2). Instead, only orthogonal arrays sampling and response surface models were used to explore the design space.

The standard orthogonal array L27 (Phadke 1993) was used for design of experiments (DOE) drawing of samples in design space. The L27 array allows for data fitting of up to 13 variables, with each variable being sampled at 3 different levels. Two independent L27 arrays (generated by randomizing the column order of the array) were used. Each of the arrays required 27 detailed FE runs, for a total of 54 runs.

Results of the design space sampling ($f_1 - f_2$ values) are shown in Fig. 8.2. It is observed that the baseline design is already Pareto-optimal within the 54 samples of the two orthogonal arrays. When performing a clustering analysis on the $f_1 - f_2$ values, the sample designs seem to have one of four behavior categories:

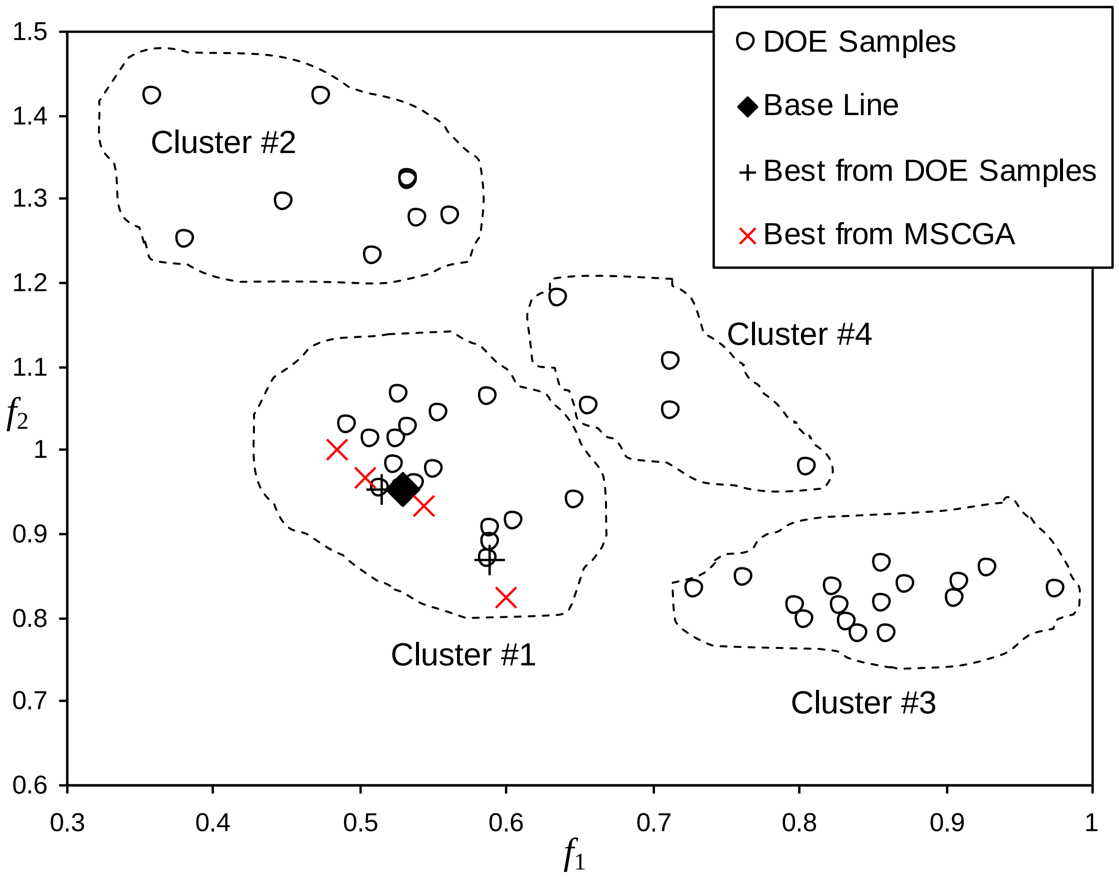


Fig. 8.2 Baseline, DOE samples and MSCGA results for Case Study 2

- Cluster #1: maintains a good balance between f_1 and f_2
- Cluster #2: has very good f_1 but bad f_2
- Cluster #3: has bad f_1 yet very good f_2
- Cluster #4: is bad in both f_1 and f_2

Further exploration of the design space was also performed through optimization using a GA running on response surface models constructed via the DOE samples. The algorithm used for this task is called Multi-Scenario Co-evolutionary Genetic Algorithm

(MSCGA). Details of the algorithm are in (Hamza and Saitou 2005). A brief summary of the algorithm is provided as:

- Multiple response surface models are constructed for the problem. 3 RSM were constructed in this problem: one RSM with data fitting on samples of the first orthogonal array, one RSM with data fitting on samples of the second orthogonal array and one RSM with data fitting on all the samples. The type of RSM used in this study was 2nd order polynomial.
- All objectives and constraints (if any) are estimated during the algorithm run via the constructed RSMs. No detailed FE simulations are needed until the end of the MSCGA run
- The algorithm co-evolves multiple populations (illustration shown in Fig. 8.3), each population is *tied* with one of the constructed RSMs. So, for this problem, 3 populations were co-evolved.
- The *fitness* in each of the evolving populations is treated in a Pareto-sense. Set to favor designs that have good performance estimate on the RSM for which it is tied, as well as good performance estimate among all the other RSMs.
- At the end of the MSCGA run, each population would include designs that are *best as estimated by the RSM the population is tied to*. This is the same result one would obtain from simply running a regular GA linked with that RSM. These designs are then checked via detailed FE
- At the end of the MSCGA run, each population would also include designs that seem to have good performance as estimated by all the constructed RSMs. Those designs are also checked via detailed FE

After running MSCGA linked to the constructed RSMs, there were 7 new sample designs that were checked via detailed FE (2 designs from each sub-population, plus an additional design that had the best estimated performance in a weighted average from all 3 constructed RSMs). Out of the 7 new sample designs, 4 were Pareto-optimal and in cluster #1 (Fig. 8.2) of the values of $f_1 - f_2$

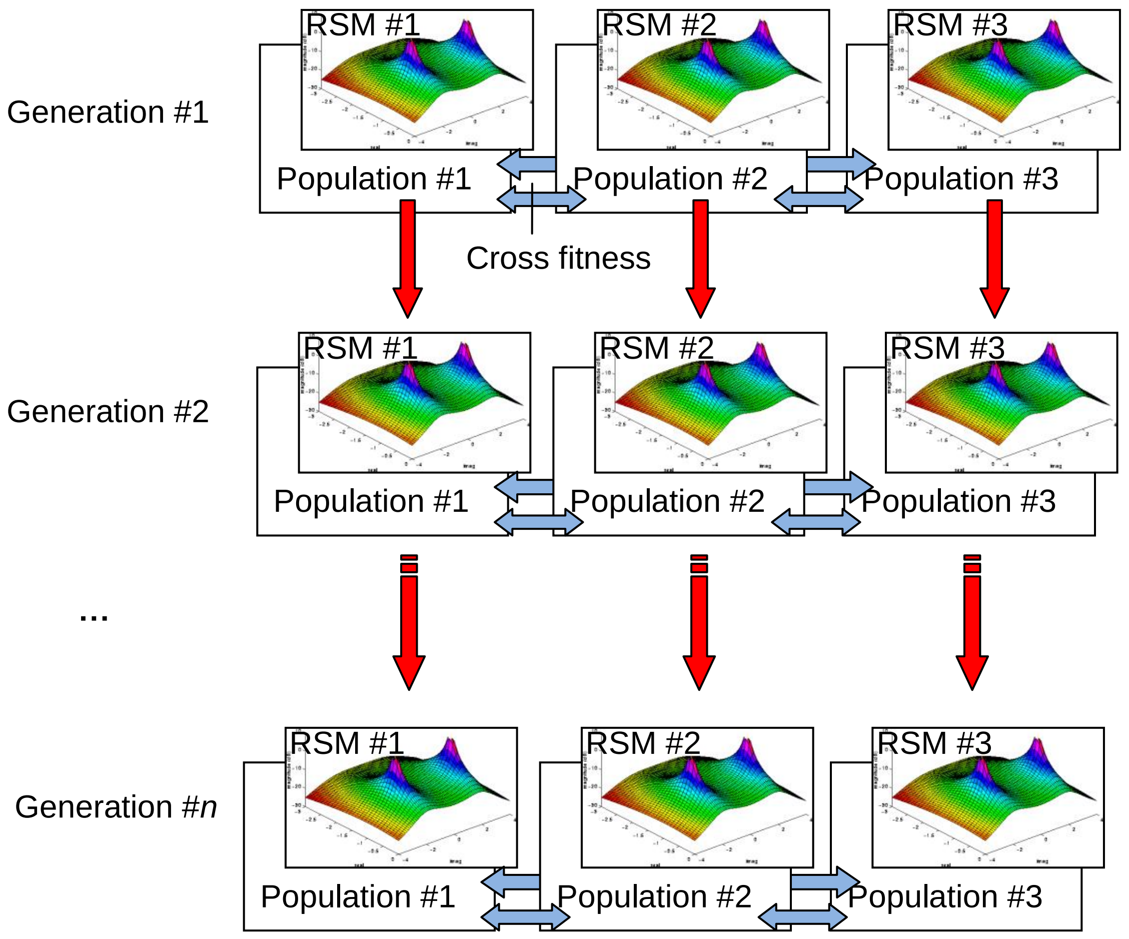


Fig. 8.3 Illustration of MSCGA algorithm

8.3 Automated Crash Mode Matching

Testing of the automated crash mode matching algorithm is performed in this section. A total of 10 structural zones are defined for the crash mode matching. Normalized values from the baseline design are set as the desirable crash mode. The crash mode mismatch metric for axial and bending deformations were combined into one crash mismatch metric for each of the 10 zones (δ_1, δ_2, …, δ_{10}) in a similar manner to case study 1. A total of 52 fuzzy rules for adjustment of the design space sampling were defined in a similar manner to the rules defined for case study 1.

The automated crash mode matching algorithm was tested starting from 2 different starting points, shown in Fig. 8.4. The first starting point (SP#1) is a representative sample from cluster #2 (low deformation, but higher acceleration), while the second starting point (SP#2) is a representative sample from cluster #3 (low acceleration, but higher deformation). Normalized values for design variables and objectives for the baseline, SP#1 and SP#2 are listed in Table 8.1. Crash mode plots for SP#1 and SP#2 are shown in Fig. 8.5 and Fig. 8.6 respectively.

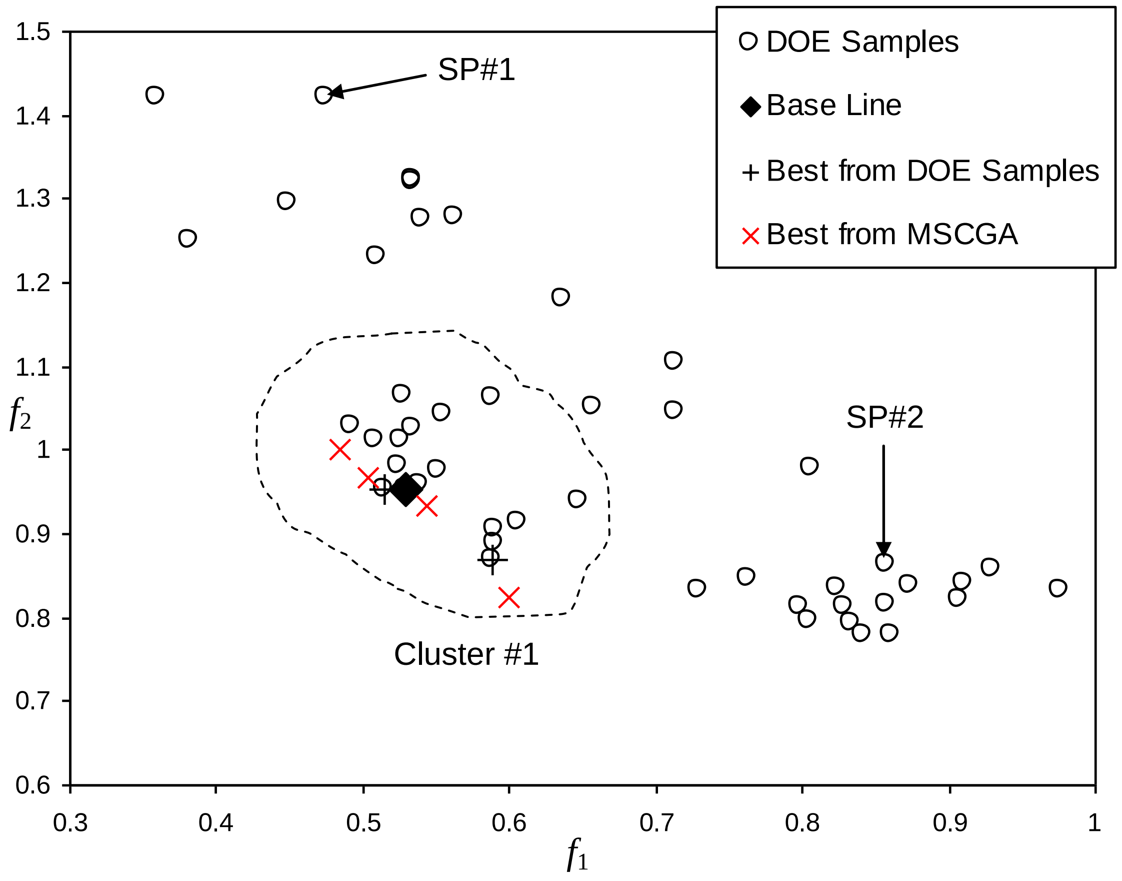


Fig. 8.4 Starting points for the automated crash mode matching algorithm

A total of 4 runs of the automated crash mode matching algorithm were performed (2 starting at each starting point). The number of samples per iteration of the algorithm *nIter* was set to 4 (in accordance with available computational resources).

Table 8.1 Normalized values for design variables and objectives of baseline design and starting points for the automated crash mode matching algorithm

	Baseline	SP#1	SP#2
x_1	1.00	1.00	1.00
x_2	1.00	1.00	0.90
x_3	1.00	0.90	0.90
x_4	1.00	0.90	1.00
x_5	1.00	1.10	1.00
x_6	1.00	1.00	1.10
x_7	1.00	1.15	1.15
x_8	1.00	0.90	1.15
x_9	1.00	1.10	1.10
x_{10}	1.00	1.15	0.90
x_{11}	1.00	0.90	1.15
x_{12}	1.00	1.00	1.10
f_1	0.53	0.47	0.85
f_2	0.95	1.42	0.86

Table 8.2 Normalized values for design variables and objectives of baseline design and final results of the automated crash mode matching algorithm

	Baseline	Run #1	Run #2	Run #3	Run #4
x_1	1.00	1.15	1.10	0.95	1.00
x_2	1.00	1.10	1.05	0.90	0.90
x_3	1.00	0.95	0.95	0.90	0.90
x_4	1.00	0.90	0.90	1.00	0.95
x_5	1.00	1.10	1.10	1.10	1.10
x_6	1.00	1.00	0.95	1.00	1.05
x_7	1.00	1.10	1.15	1.00	1.15
x_8	1.00	0.90	0.90	1.10	1.05
x_9	1.00	0.90	1.15	1.10	1.10
x_{10}	1.00	1.10	1.05	0.90	0.90
x_{11}	1.00	1.15	1.10	1.10	1.10
x_{12}	1.00	0.85	1.00	1.05	0.90
f_1	0.53	0.45	0.54	0.65	0.48
f_2	0.95	1.09	1.10	0.89	1.08

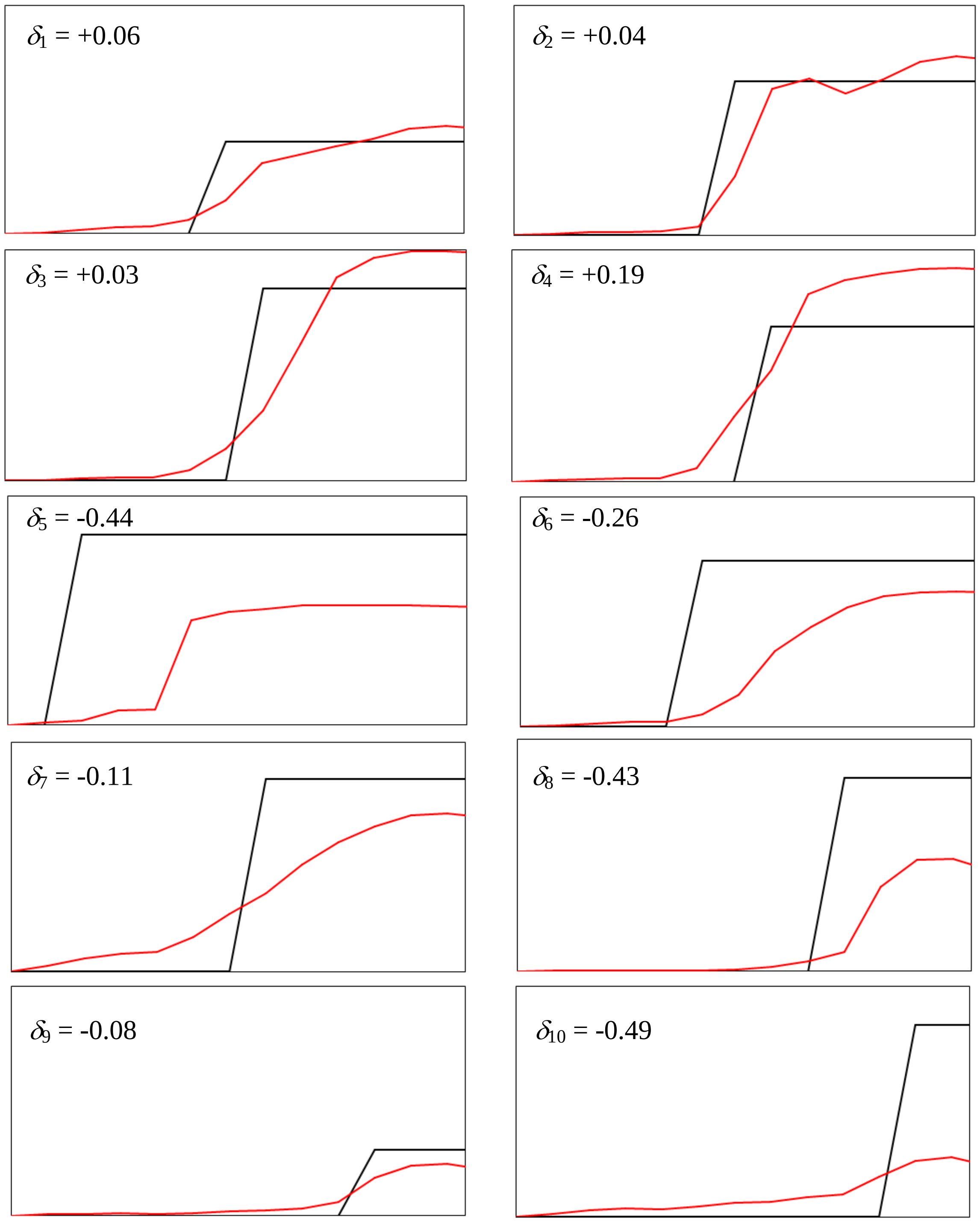


Fig. 8.5 Crash Mode Plot for Start Pt#1

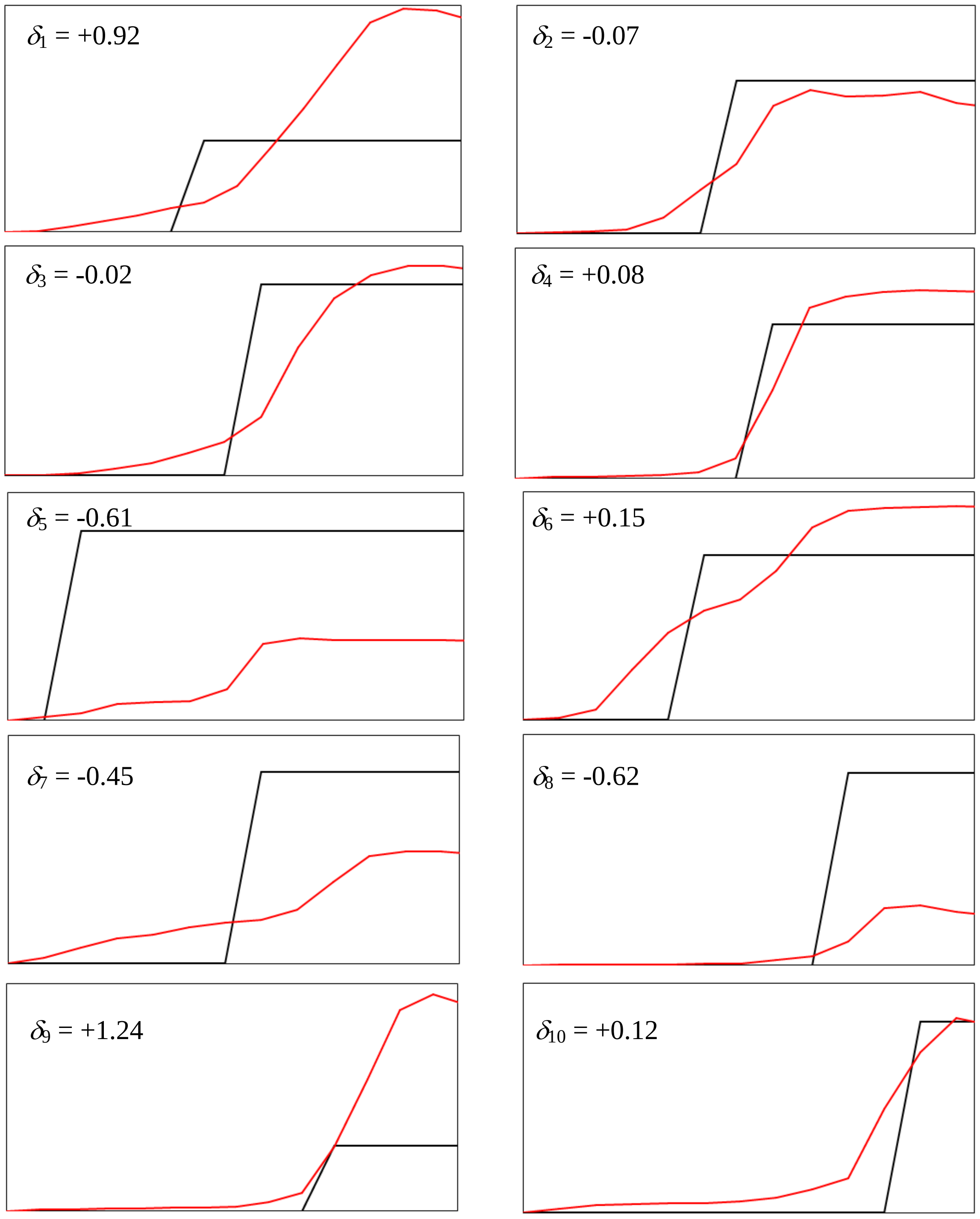


Fig. 8.6 Crash Mode Plot for Start Pt#2

All runs of the automated crash mode matching algorithm were successful in producing a design in cluster #1 in at most 2 iterations of the algorithm (Fig. 8.7). Furthermore, the designs discovered by run #1 and run #4 are Pareto-optimal with respect to the current known designs. Normalized values for design variables and objectives for the baseline and run #1 through #4 are listed in Table 8.2. Crash mode plots for run #1 through #4 are shown in Fig. 8.8 through Fig. 8.11 respectively.

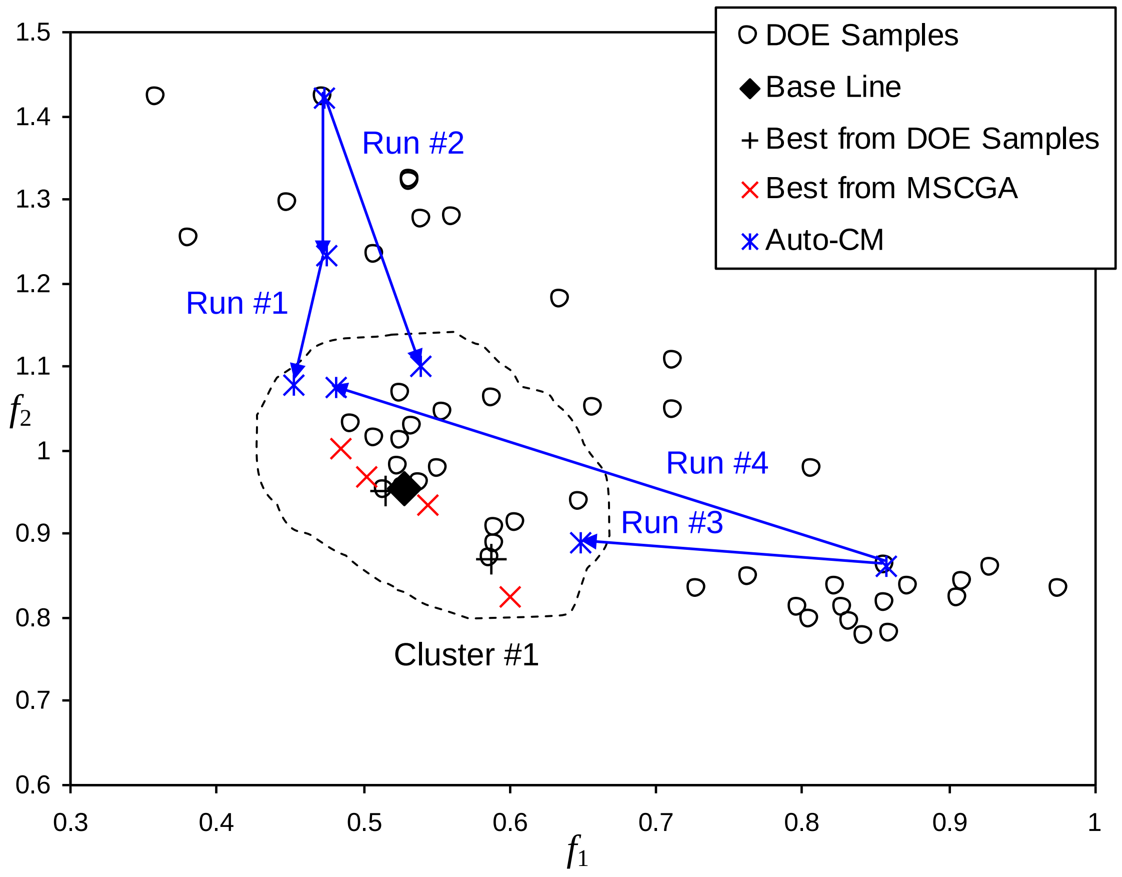


Fig. 8.7 Progress of the Runs of the Automated Crash Mode Matching Algorithm

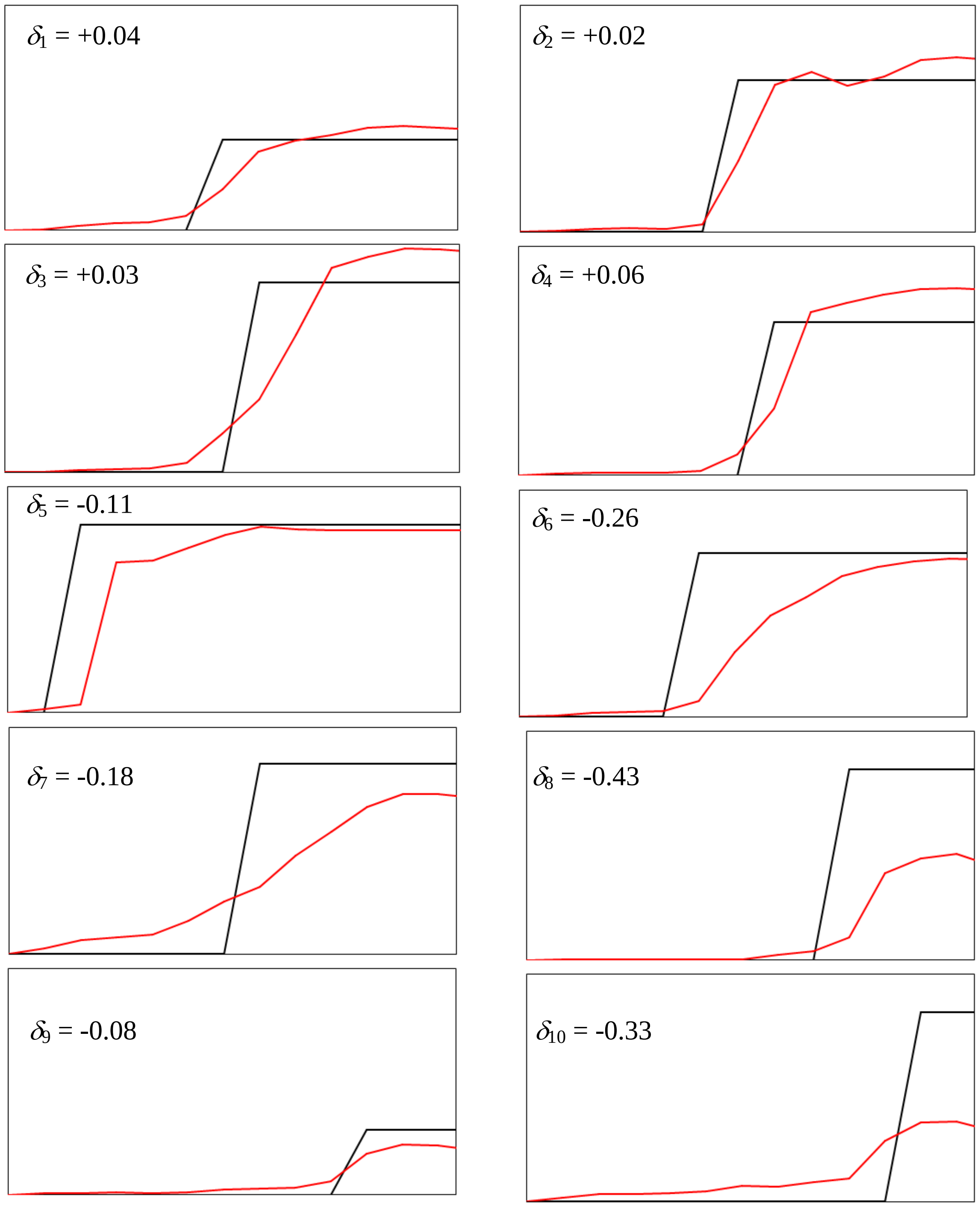


Fig. 8.8 Crash Mode Plot for Final Design in Run#1

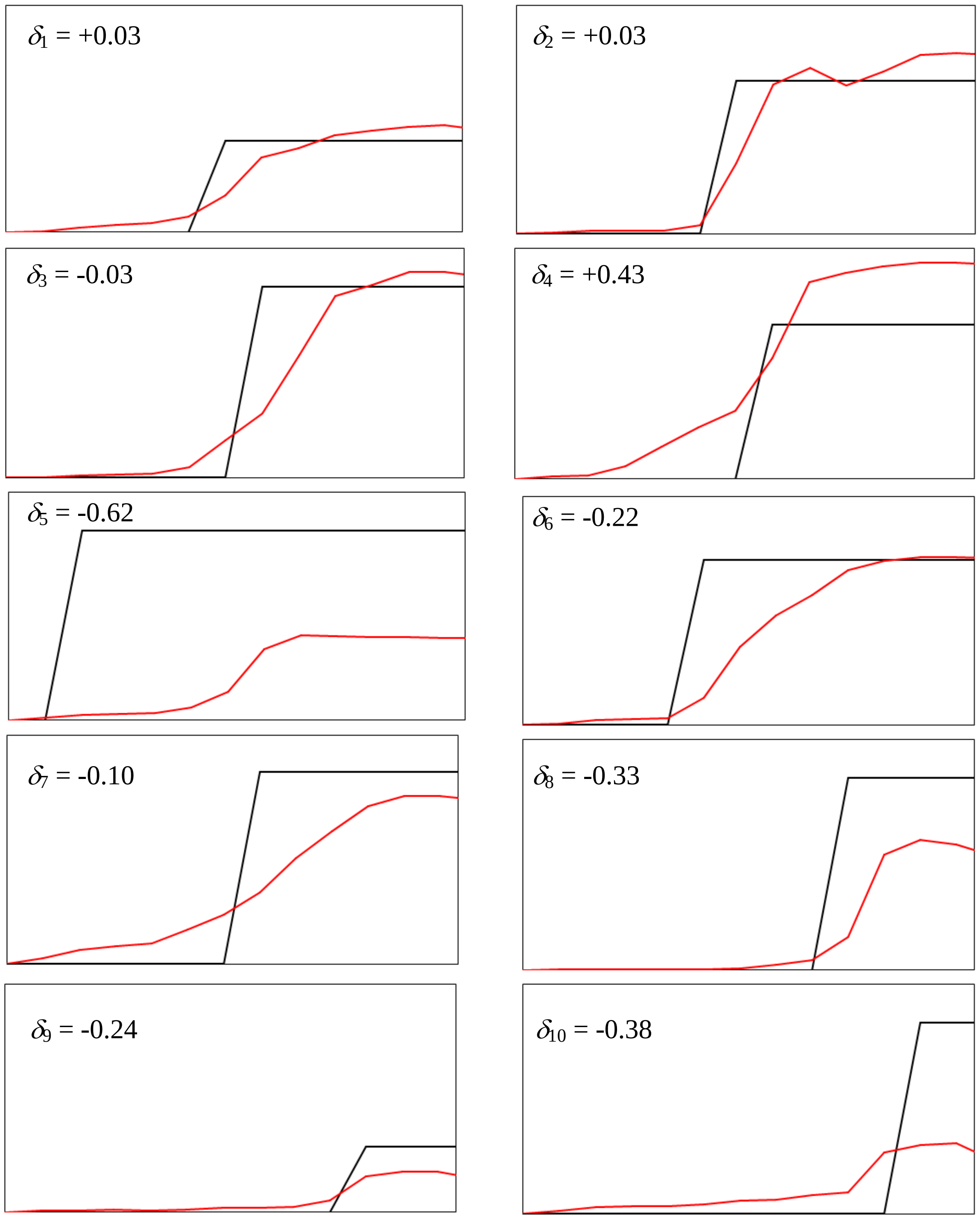


Fig. 8.9 Crash Mode Plot for Final Design in Run#2

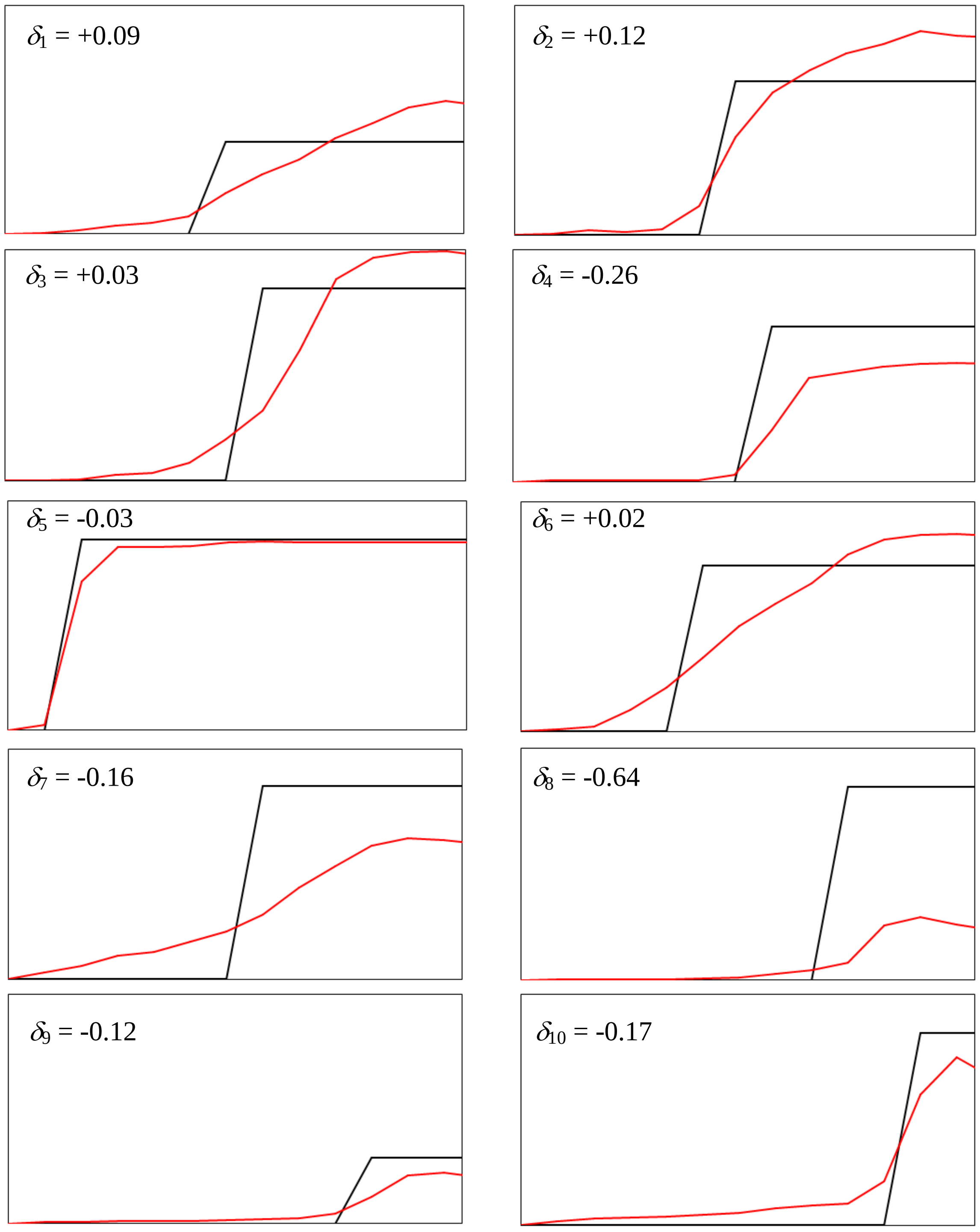


Fig. 8.10 Crash Mode Plot for Final Design in Run#3

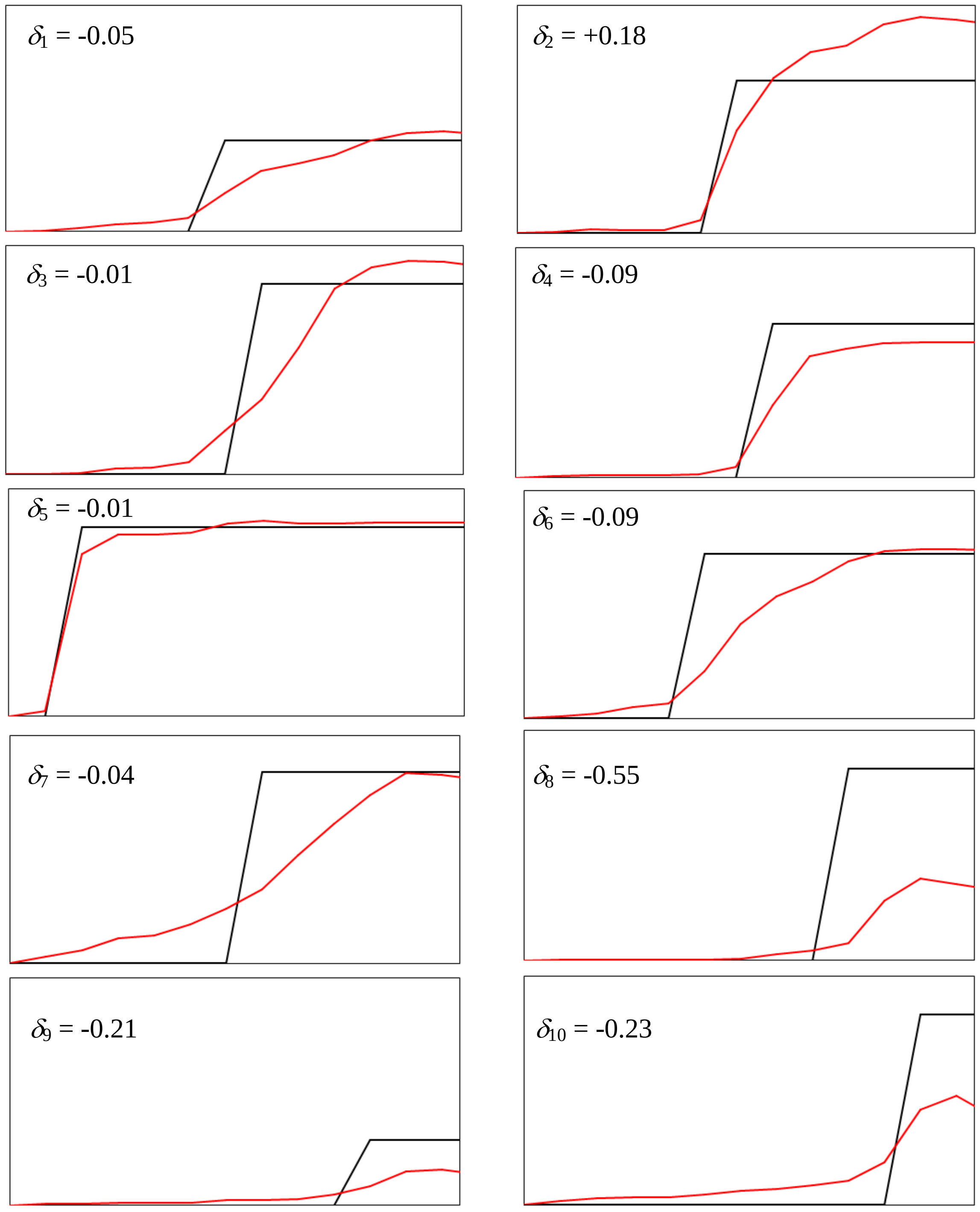


Fig. 8.11 Crash Mode Plot for Final Design in Run#4

8.4 Summary

This chapter presented application of the automated crash mode matching algorithm to a case study of a detailed full vehicle model subjected to frontal crash against an offset deformable barrier. Four runs of the proposed algorithm were started at designs that mismatched the desirable crash mode and were successful in improving the design performance by adjusting the crash mode to the desired one. This is a demonstration of the effectiveness of the proposed methodology when applied to vehicle models of realistic level of detail.

CHAPTER 9
CONCLUSION

This chapter concludes the dissertation. It includes a summary of the work, list of contributions and expected future extensions.

9.1 Dissertation Conclusion

The research performed in this dissertation targeted the development of a design methodology for parametric structural crashworthiness by formalizing the crash mode matching approach. A quantitative representation of the crash mode is introduced, as well as comparison metrics for the degree of mismatch in crash mode between different designs. An algorithm is then design for automated crash mode matching. The algorithm heuristically directs stochastic sampling of the design space based on Fuzzy logic rules that are defined in analogy to the type of decisions that an experienced designer would make for crash mode matching.

Two case studies were presented to demonstrate that the proposed methodology. The first study considered a frontal half body box-section structure vehicle model subjected to frontal impact conditions against a rigid barrier. The problem had one objective and two constraints. The proposed methodology was successful in attaining feasible, good performance designs at a reasonable amount of computational requirement.

In the second study, a detailed finite element model of a full vehicle subjected to frontal crash against an offset deformable barrier was considered. The problem was formulated as a two-objective problem. Within the limited available computational resources, several tested runs of the proposed methodology were successful in adjusting the design variables started at non-favorable designs and attain an improved performance.

9.2 Contributions

The main contribution of this dissertation is the *formalization* of crash mode matching as a methodology for structural crashworthiness design. This formalization included:

- Introduction of quantitative metrics for crash modes assessment and comparison based on the recorded time history of deformation in different structural zones

- Development of Equivalent mechanism models, which are reduced order dynamic models, to assist in exploration and discovery of desirable crash modes

- Development of a stochastic design-space sampling algorithm for parametric structural crashworthiness optimization via automated crash mode matching

9.3 Future Work

While there could be several extensions to the research presented in this dissertation, the following are perceived attractive:

- Hybridizing features from response surface methods and crash modes matching approaches. During some of the performed studies, the input-output relations that are fitted by the RSM remain well conditioned as long as there is no change in the crash mode. While RSM alone cannot detect a change in the crash mode, elements from the crash mode matching approach may help overcome this difficulty.

- Developing of further systematic approaches to explore and discover desirable crash modes for vehicle structures. This could follow several routes of reduced order mass-spring type models, pure kinematic models and/or coarse mesh finite element models

- Developing better metrics for assessment of the degree of crash mode mismatch. The developed automated crash mode matching algorithm requires a metric that is able to indicate whether the deformation in a zone is less or more than the desired amount. The relaxed mismatch metric used in this thesis, which is a simple normalized error integral, was sufficient in the considered case studies. The relaxed metric however has a drawback that in some occasions, a mismatching crash mode that has both positive and negative portions in its time history might not be well detected.

APPENDICES

APPENDIX A
Tutorial: Construction of an EM Model

Sketches of the main components (complete symmetry about global YZ plane is assumed), which are to be included in the EM model are provided in Figures A.1 – Fig. A.5.

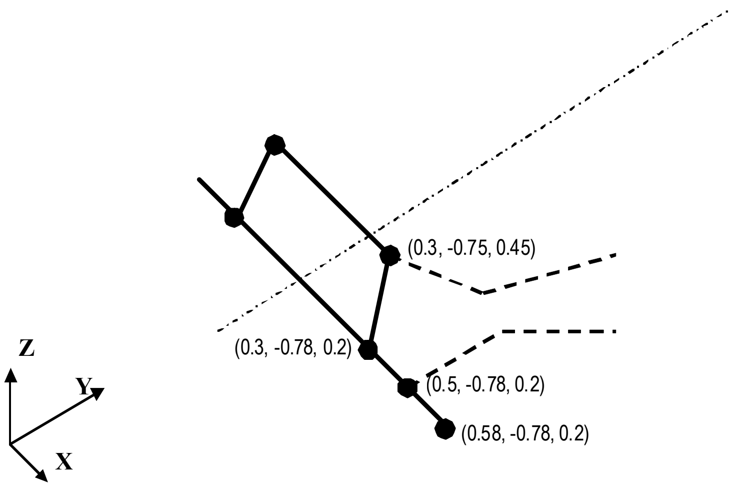


Fig. A.1 Sketch of key point locations

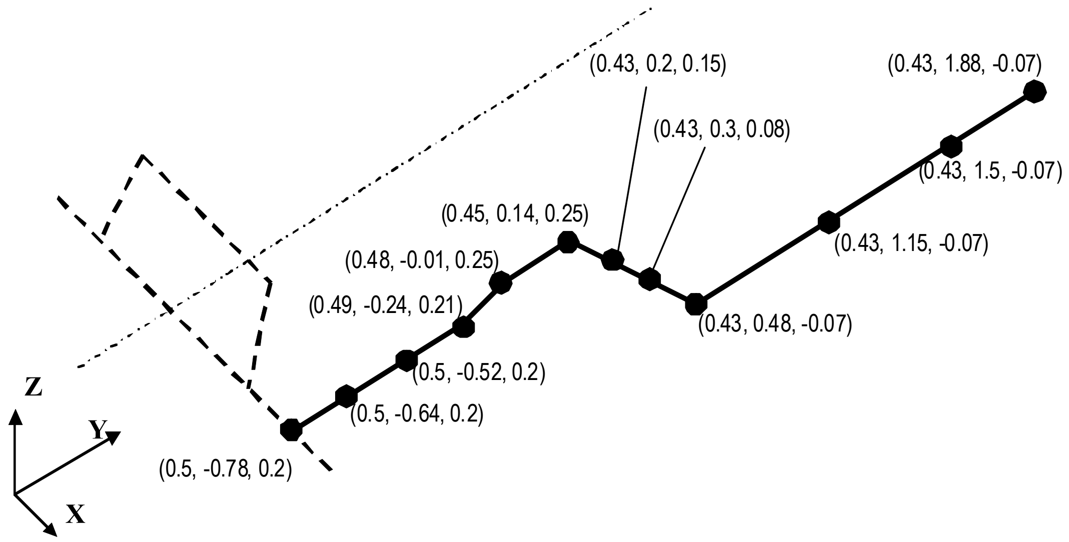


Fig. A.2 Sketch of key point locations

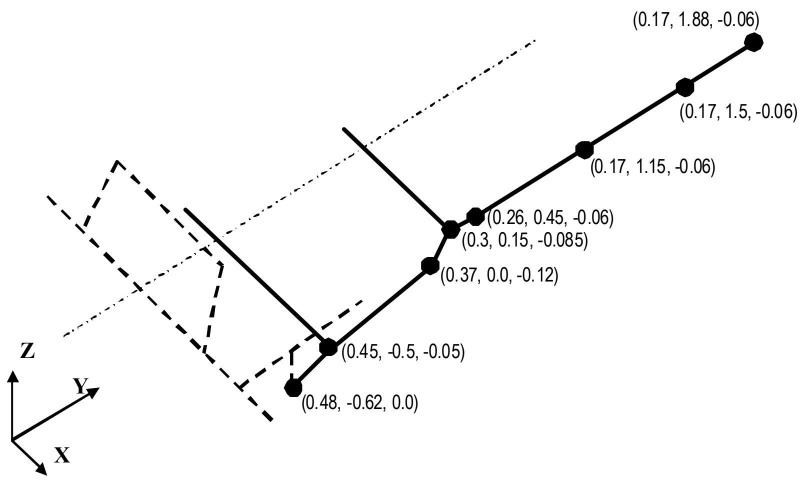


Fig. A.3 Sketch of key point locations

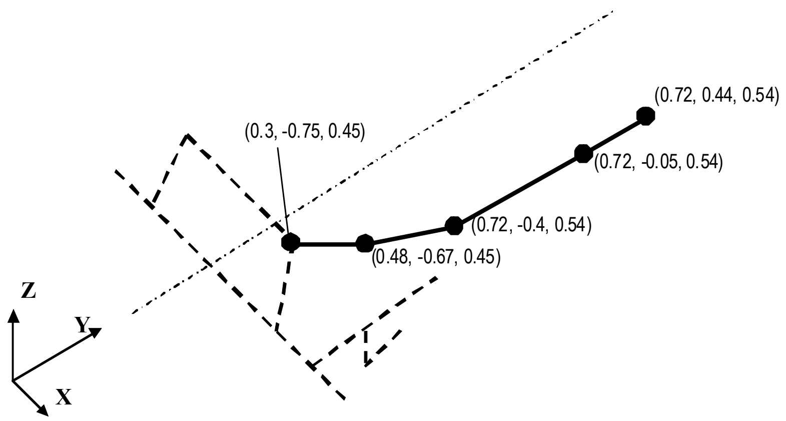


Fig. A.4 Sketch of key point locations

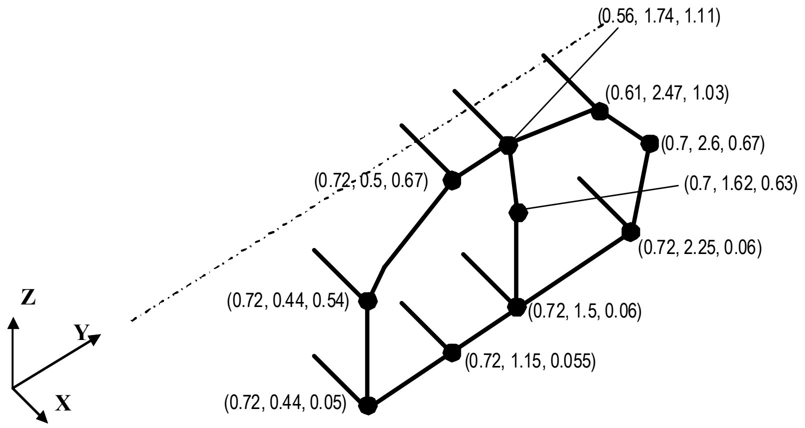


Fig. A.5 Sketch of key point locations

The main screen of the dODE Crash Designer software looks like in Fig. A.6. The main modeling commands are accessible through the menu system:

Modeling → Key Point → Add…
Modeling → Key Point → Edit…
Modeling → Key Point → Delete
Modeling → Key Point → Identify

Modeling → EM Link → Add…
Modeling → EM Link → Edit…
Modeling → EM Link → Delete
Modeling → EM Link → Identify

Modeling → Side Squisher → Add…
Modeling → Side Squisher → Edit…
Modeling → Side Squisher → Delete
Modeling → Side Squisher → Identify

Modeling → Rigid Body → Add…
Modeling → Rigid Body → Edit…
Modeling → Rigid Body → Delete
Modeling → Rigid Body → Identify

Modeling → Panel → Add…
Modeling → Panel → Edit…
Modeling → Panel → Delete
Modeling → Panel → Identify

Modeling → Force Curve → Add…
Modeling → Force Curve → Edit…
Modeling → Force Curve → Delete
Modeling → Force Curve → Identify

Alternatively, the modeling commands are also available through the modeling toolbar (Fig. A.6)

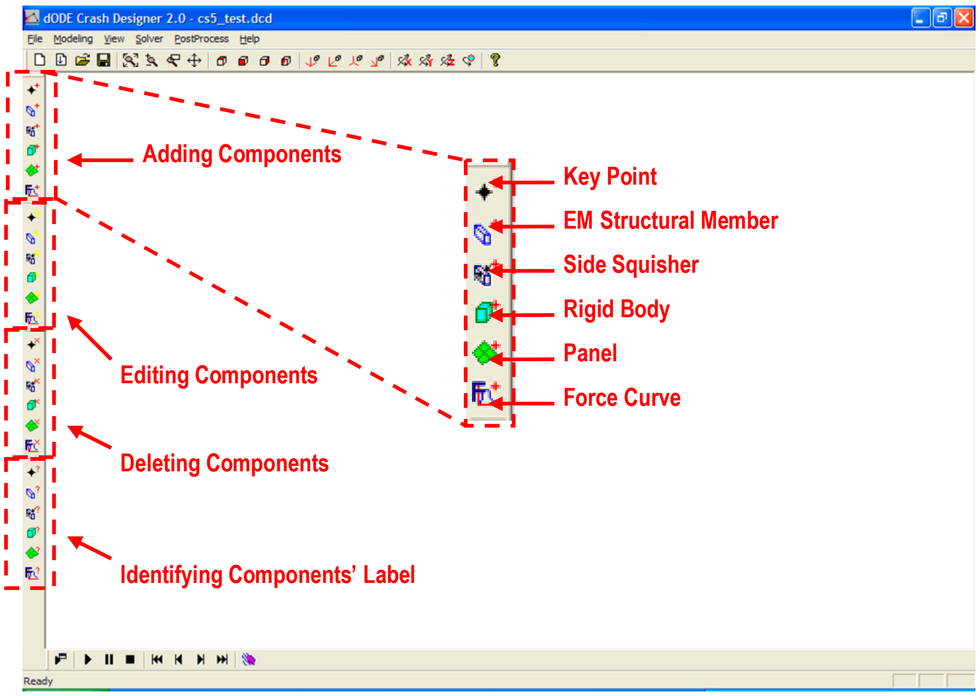


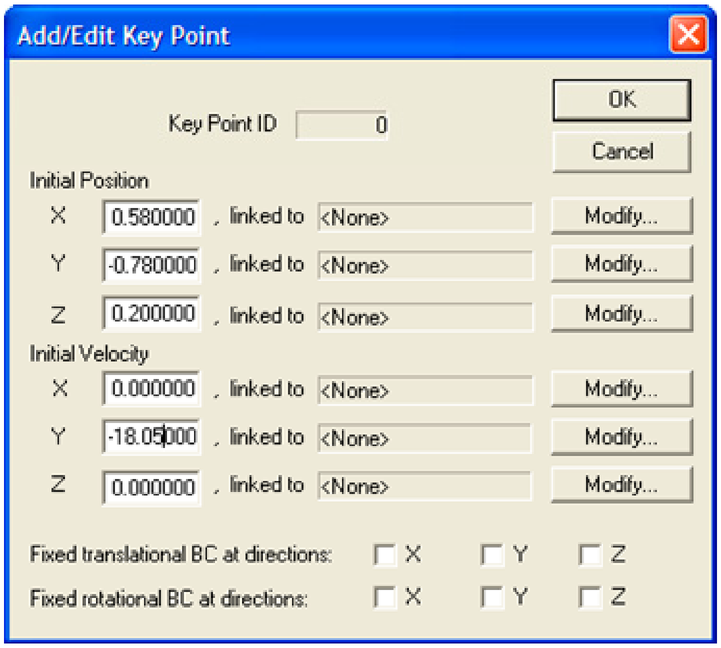
Fig. A.6 Main Screen of dODE Crash Designer and Modeling Toolbar

EM modeling starts by adding Key Points, upon which the rest of the EM modeling components are mounted. Key points also specify the initial and boundary (fixed/free) conditions. Unless otherwise specified, the boundary conditions are set to fully free along all directions and rotations, and the initial conditions are set as initial velocity equal to -18.05 m/s along the global Y-direction. All units in model are assumed to follow SI system (kg, m, sec).

Start by adding Key points (as per the sketches in Fig. A.1-A.5). Fig. A.7 shows the dialog for Adding (and/or editing a Key point).

Next create EM structural links. The Add/Edit dialog for EM structural links is shown in Fig. A.8. Among the main inputs of the EM link are the connectivity key points, which may be entered directly (if their labels are know), or by clicking the "Select…" button to open a selection screen. In the selection screen, left clicking selects the nearest entity, right clicking selects the "next-nearest" entity, pressing the "enter" or "space" key accepts the selection, while pressing the "esc" key cancels the selection. The Add/Edit dialog for EM structural links also allows selection of the available databases for the cross-section and the interpolating surrogate. Selecting the section type automatically determines the number of dimensions which can be entered (3 for box sections: height, width and thickness). When done click the "OK" button.

When done with creating the EM links, the screen should look similar to Fig. A.9



Key Point ID 0

Initial Position

X 0.580000 , linked to <None>

Y -0.780000 , linked to <None>

Z 0.200000 , linked to <None>

Initial Velocity

X 0.000000 , linked to <None>

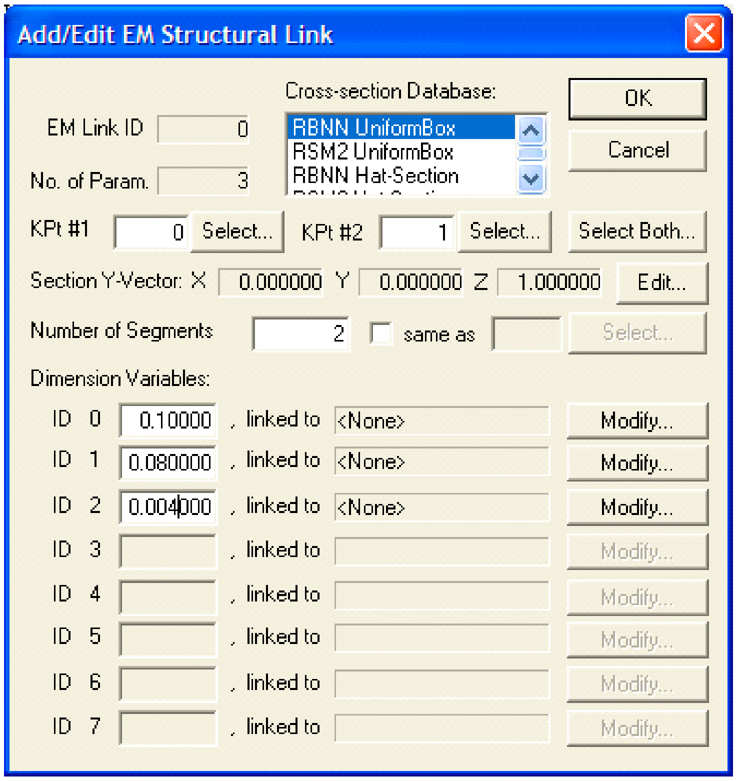
Y -18.050000 , linked to <None>

Z 0.000000 , linked to <None>

Fixed translational BC at directions: ☐ X ☐ Y ☐ Z

Fixed rotational BC at directions: ☐ X ☐ Y ☐ Z

Fig. A.7 Add/Edit Key Point Dialog



EM Link ID 0

No. of Param. 3

Cross-section Database: RBNN UniformBox, RSM2 UniformBox, RBNN Hat-Section

KPt #1 0 KPt #2 1

Section Y-Vector: X 0.000000 Y 0.000000 Z 1.000000

Number of Segments 2 ☐ same as

Dimension Variables:

ID 0 0.10000 , linked to <None>

ID 1 0.080000 , linked to <None>

ID 2 0.004000 , linked to <None>

ID 3 , linked to

ID 4 , linked to

ID 5 , linked to

ID 6 , linked to

ID 7 , linked to

Fig. A.8 Add/Edit EM-link Dialog

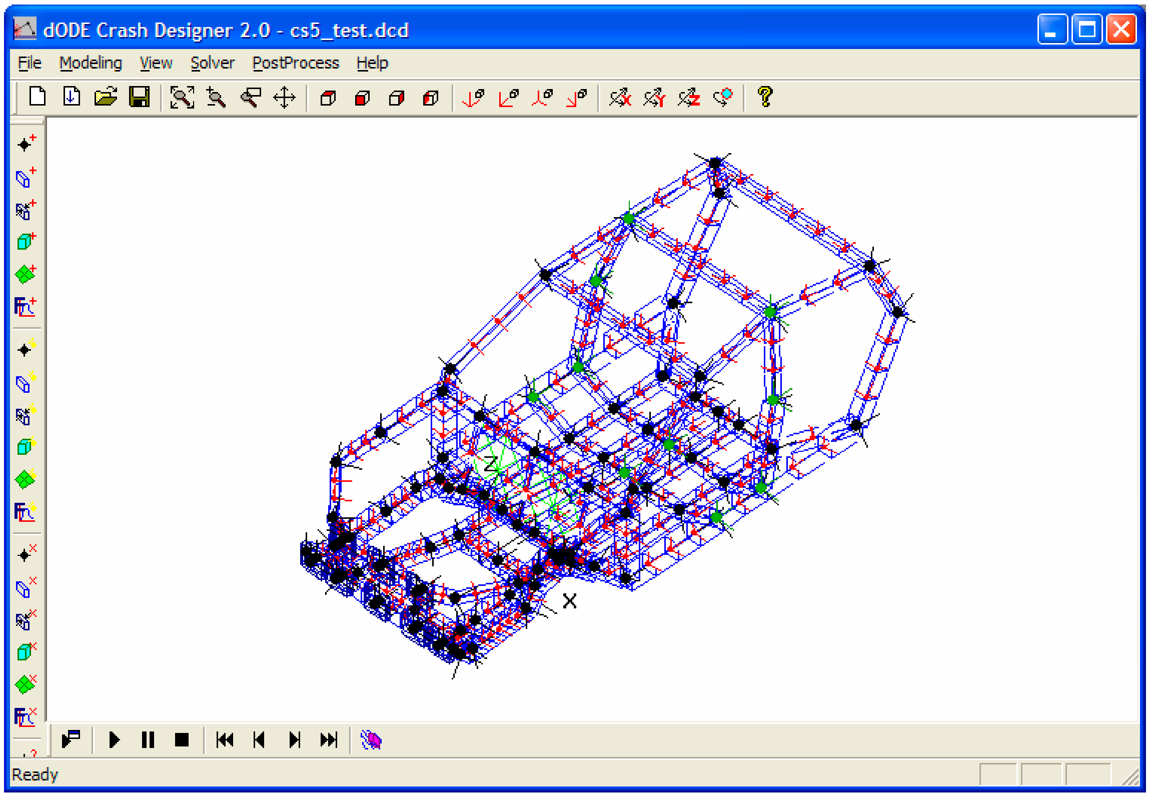

dODE Crash Designer 2.0 - cs5_test.dcd
File Modeling View Solver PostProcess Help
Ready

Fig. A.9 Main Screen of dODE Crash Designer and Modeling, EM model constructed

Additional features offered by the software include options to add a representation for panels (Fig. A.10), rigid bodies (Fig. A.11) and Force curves (Fig. A.12).

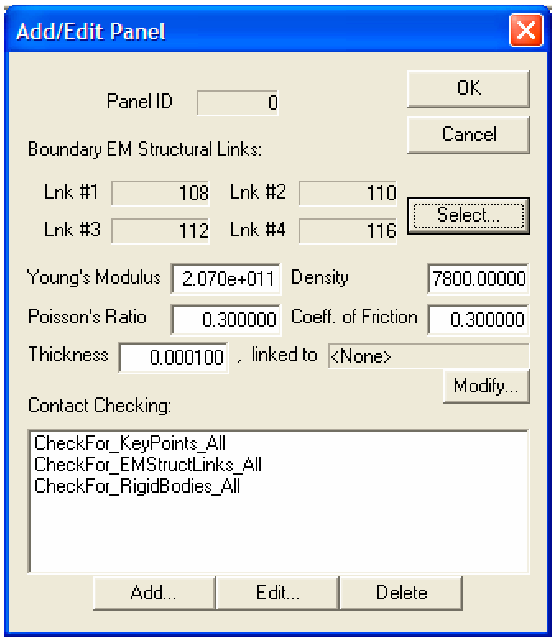
To solve the EM model. From the menu system, select:

Solver → Parameters…

Adjust the solver parameters as shown in Fig. A.13, then select the menu command:

Solver → Run…

The software automatically saves the model before attempting to solve. Then, a progress bar indicates the solver progress.


Add/Edit Panel

Panel ID 0

Boundary EM Structural Links:

Lnk #1 108 Lnk #2 110

Lnk #3 112 Lnk #4 116

Young's Modulus 2.070e+011 Density 7800.00000

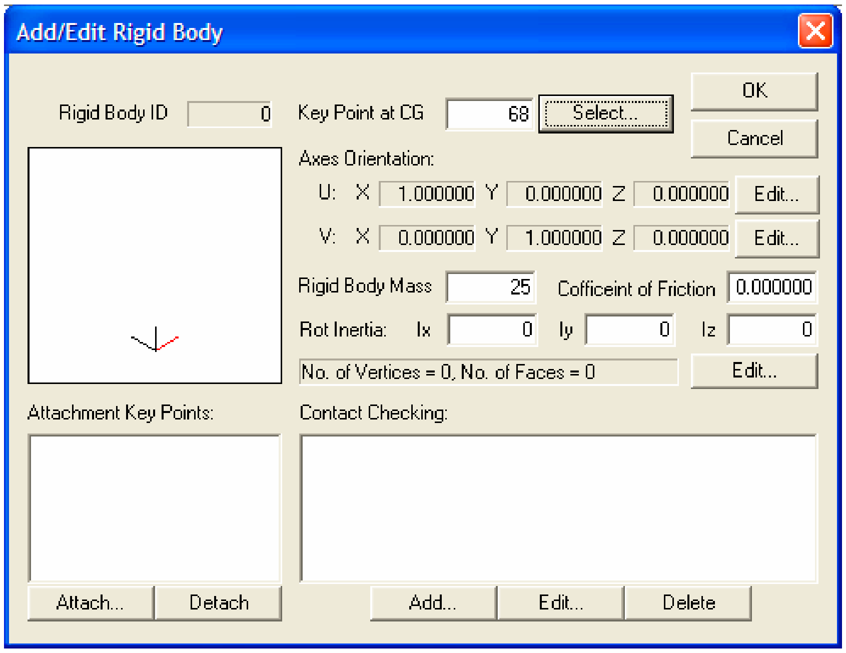
Poisson's Ratio 0.300000 Coeff. of Friction 0.300000

Thickness 0.000100 , linked to <None>

Contact Checking:

```
CheckFor_KeyPoints_All
CheckFor_EMStructLinks_All
CheckFor_RigidBodies_All
```

Fig. A.10 Add/Edit Panel Dialog


Add/Edit Rigid Body

Rigid Body ID 0 Key Point at CG 68

Axes Orientation:

U: X 1.000000 Y 0.000000 Z 0.000000

V: X 0.000000 Y 1.000000 Z 0.000000

Rigid Body Mass 25 Cofficeint of Friction 0.000000

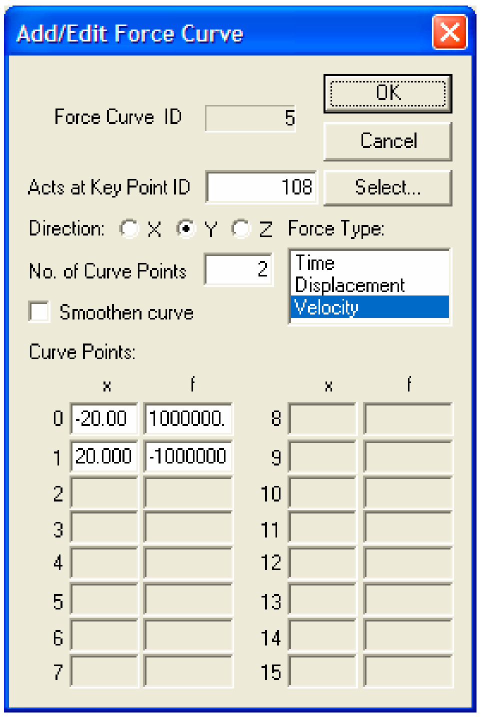
Rot Inertia: Ix 0 Iy 0 Iz 0

No. of Vertices = 0, No. of Faces = 0

Attachment Key Points:

Contact Checking:

Fig. A.11 Add/Edit Rigid Body Dialog


Add/Edit Force Curve

Force Curve ID: 5

Acts at Key Point ID: 108

Direction: X [✓ Y] Z

Force Type: Time / Displacement / [✓ Velocity]

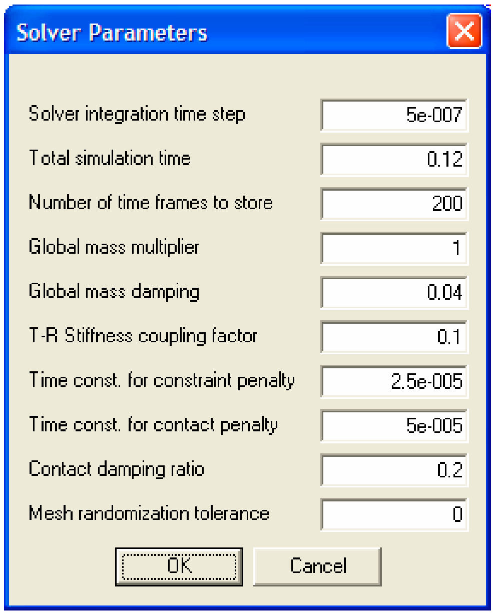
No. of Curve Points: 2

Smoothen curve: ☐

Curve Points:

	x	f		x	f
0	-20.00	1000000.	8		
1	20.000	-1000000	9		
2			10		
3			11		
4			12		
5			13		
6			14		
7			15		

Fig. A.12 Add/Edit Force Curve Dialog


Solver Parameters

Solver integration time step	5e-007
Total simulation time	0.12
Number of time frames to store	200
Global mass multiplier	1
Global mass damping	0.04
T-R Stiffness coupling factor	0.1
Time const. for constraint penalty	2.5e-005
Time const. for contact penalty	5e-005
Contact damping ratio	0.2
Mesh randomization tolerance	0

Fig. A.13 Solver Parameters Dialog

APPENDIX B
DOE Samples for Construction of RSM for Case Study 1

Table B.1 Sample Set #1

	Sample Number					
	1	2	3	4	5	6
h_1 (mm)	90.0	110	130	90	110	130
b_1 (mm)	50.0	50	50	70	70	70
h_2 (mm)	50.0	70	90	70	90	50
b_2 (mm)	70.0	70	70	90	90	90
t_1 (mm)	2.0	2.2	2.4	2.0	2.2	2.4
t_2 (mm)	1.8	1.8	1.8	2.0	2.0	2.0
t_3 (mm)	1.6	1.8	2.0	2.0	1.6	1.8
t_4 (mm)	2.6	2.8	3.0	2.6	2.8	3.0
t_5 (mm)	2.8	3.0	3.2	2.8	3.0	3.2
t_6 (mm)	2.8	2.8	2.8	3.0	3.0	3.0
t_7 (mm)	1.0	1.2	1.4	1.0	1.2	1.4
t_8 (mm)	2.0	2.2	2.4	2.2	2.4	2.0
t_9 (mm)	2.4	2.6	2.8	2.8	2.4	2.6
t_{10} (mm)	2.6	2.6	2.6	2.6	2.6	2.6
t_{11} (mm)	2.8	2.8	3.0	3.0	3.0	3.0
t_{12} (mm)	2.2	2.4	2.4	2.4	2.6	2.2
t_{13} (mm)	1.8	2.0	1.8	1.8	2.0	2.2
t_{14} (mm)	2.0	2.0	2.2	2.2	2.2	2.2
f (kg)	51.2	62.6	75.1	64.8	74.4	75.6
g_1 (mm)	276.5	154.4	182.1	152.3	81.2	55.5
g_2 (g)	23.7	28.3	36.6	29.4	27.8	29.1

Table B.2 Sample Set #1 – continued

	Sample Number					
	7	8	9	10	11	12
h_1 (mm)	90	110	130	90	110	130
b_1 (mm)	90	90	90	50	50	50
h_2 (mm)	90	50	70	50	70	90
b_2 (mm)	110	110	110	110	110	110
t_1 (mm)	2.0	2.2	2.4	2.0	2.2	2.4
t_2 (mm)	2.2	2.2	2.2	2.2	2.2	2.2
t_3 (mm)	1.8	2	1.6	1.6	1.8	2
t_4 (mm)	2.6	2.8	3.0	3.0	2.6	2.8
t_5 (mm)	2.8	3	3.2	3.2	2.8	3.0
t_6 (mm)	3.2	3.2	3.2	2.8	2.8	2.8
t_7 (mm)	1.0	1.2	1.4	1.2	1.4	1.0
t_8 (mm)	2.4	2.0	2.2	2.0	2.2	2.4
t_9 (mm)	2.6	2.8	2.4	2.4	2.6	2.8
t_{10} (mm)	2.6	2.6	2.6	2.8	2.8	2.8
t_{11} (mm)	3.2	3.2	3.2	3.0	3.0	3.0
t_{12} (mm)	2.6	2.2	2.4	2.4	2.6	2.2
t_{13} (mm)	1.8	2.0	2.2	2.0	2.2	1.8
t_{14} (mm)	2.4	2.4	2.4	2.2	2.2	2.2
f (kg)	76.4	77.3	87.6	60.9	70.6	79.0
g_1 (mm)	68.6	103.0	52.9	164.7	152.2	237.5
g_2 (g)	29.1	26.6	37.9	25.1	26.2	29.9

Table B.3 Sample Set #1 – continued

	Sample Number					
	13	14	15	16	17	18
h_1 (mm)	110	130	90	110	130	90
b_1 (mm)	70	70	90	90	90	70
h_2 (mm)	90	50	90	50	70	50
b_2 (mm)	70	70	90	90	90	110
t_1 (mm)	2.2	2.4	2	2.2	2.4	2.2
t_2 (mm)	1.8	1.8	2.0	2.0	2.0	2.0
t_3 (mm)	1.6	1.8	1.8	2.0	1.6	2.0
t_4 (mm)	2.6	2.8	3.0	2.6	2.8	3.0
t_5 (mm)	2.8	3.0	3.2	2.8	3.0	3.0
t_6 (mm)	3.0	3.0	3.2	3.2	3.2	2.8
t_7 (mm)	1.4	1.0	1.2	1.4	1.0	1.4
t_8 (mm)	2.4	2.0	2.4	2.0	2.2	2.2
t_9 (mm)	2.4	2.6	2.6	2.8	2.4	2.4
t_{10} (mm)	2.8	2.8	2.8	2.8	2.8	3.0
t_{11} (mm)	3.2	3.2	2.8	2.8	2.8	3.2
t_{12} (mm)	2.2	2.4	2.2	2.4	2.6	2.2
t_{13} (mm)	2.2	1.8	2.0	2.2	1.8	1.8
t_{14} (mm)	2.4	2.4	2.0	2.0	2.0	2.0
f (kg)	71.0	71.0	74.3	74.2	81.9	66.7
g_1 (mm)	124.8	72.9.	94.5	117.5	51.2	185.4
g_2 (g)	30.1	30.9	31.3	33.3	29.0	31.3

Table B.4 Sample Set #1 – continued

	Sample Number					
	19	20	21	22	23	24
h_1 (mm)	110	130	90	110	130	90
b_1 (mm)	70	70	90	90	90	50
h_2 (mm)	70	90	70	90	50	90
b_2 (mm)	110	110	70	70	70	90
t_1 (mm)	2.4	2.0	2.2	2.4	2.0	2.2
t_2 (mm)	2.0	2.0	2.2	2.2	2.2	1.8
t_3 (mm)	1.6	1.8	1.8	2.0	1.6	1.6
t_4 (mm)	2.6	2.8	3.0	2.6	2.8	3.0
t_5 (mm)	3.2	2.8	3.0	3.2	2.8	3.0
t_6 (mm)	2.8	2.8	3.0	3.0	3.0	3.2
t_7 (mm)	1.0	1.2	1.4	1.0	1.2	1.4
t_8 (mm)	2.4	2.0	2.4	2	2.2	2.0
t_9 (mm)	2.6	2.8	2.8	2.4	2.6	2.6
t_{10} (mm)	3.0	3.0	3.0	3.0	3.0	3.0
t_{11} (mm)	3.2	3.2	2.8	2.8	2.8	3.0
t_{12} (mm)	2.4	2.6	2.4	2.6	2.2	2.6
t_{13} (mm)	2.0	2.2	1.8	2.0	2.2	1.8
t_{14} (mm)	2.0	2.0	2.2	2.2	2.2	2.4
f (kg)	76.0	84.3	69.2	78.8	73.8	66.7
g_1 (mm)	91.8	89.1	138.4	100.0	152.4	130.1
g_2 (g)	25.5	32.3	30.9	33.6	27.5	29.5

Table B.5 Sample Set #1 – continued

	Sample Number					
	25	26	27	28	29	30
h_1 (mm)	110	130	90	110	130	90
b_1 (mm)	50	50	90	90	90	50
h_2 (mm)	50	70	50	70	90	70
b_2 (mm)	90	90	70	70	70	90
t_1 (mm)	2.4	2.0	2.2	2.4	2.0	2.4
t_2 (mm)	1.8	1.8	2	2.0	2.0	2.2
t_3 (mm)	1.8	2.0	1.8	2.0	1.6	1.6
t_4 (mm)	2.6	2.8	3.0	2.6	2.8	2.6
t_5 (mm)	3.2	2.8	3.0	3.2	2.8	3.0
t_6 (mm)	3.2	3.2	2.8	2.8	2.8	3.0
t_7 (mm)	1.0	1.2	1.4	1.0	1.2	1.2
t_8 (mm)	2.2	2.4	2.4	2.0	2.2	2.0
t_9 (mm)	2.8	2.4	2.4	2.6	2.8	2.8
t_{10} (mm)	3.0	3.0	2.6	2.6	2.6	2.6
t_{11} (mm)	3.0	3.0	3.0	3.0	3	3.2
t_{12} (mm)	2.2	2.4	2.6	2.2	2.4	2.2
t_{13} (mm)	2.0	2.2	1.8	2.0	2.2	2.2
t_{14} (mm)	2.4	2.4	2.4	2.4	2.4	2.0
f (kg)	65.3	73.0	64.3	73.3	79.5	62.4
g_1 (mm)	100.0	119.5	157.2	66.9	79.4	246.1
g_2 (g)	27.0	30.0	27.6	33.9	33.3	28.0

Table B.6 Sample Set #1 – continued

	Sample Number					
	31	32	33	34	35	36
h_1 (mm)	110	130	90	130	90	110
b_1 (mm)	50	50	70	70	70	70
h_2 (mm)	90	50	90	70	50	70
b_2 (mm)	90	90	110	110	90	90
t_1 (mm)	2.0	2.2	2.4	2.2	2.2	2.4
t_2 (mm)	2.2	2.2	1.8	1.8	2.2	2.2
t_3 (mm)	1.8	2.0	2.0	1.8	2.0	1.6
t_4 (mm)	2.8	3.0	2.6	3.0	2.8	3.0
t_5 (mm)	3.3	2.8	3.0	2.8	3.2	2.8
t_6 (mm)	3.0	3.0	3.2	3.2	2.8	2.8
t_7 (mm)	1.4	1.0	1.2	1.0	1.0	1.2
t_8 (mm)	2.2	2.4	2.2	2.0	2.2	2.4
t_9 (mm)	2.4	2.6	2.6	2.4	2.4	2.6
t_{10} (mm)	2.6	2.6	2.6	2.6	2.8	2.8
t_{11} (mm)	3.2	3.2	2.8	2.8	2.8	2.8
t_{12} (mm)	2.4	2.6	2.4	2.2	2.6	2.2
t_{13} (mm)	1.8	2.0	2.2	2.0	2.2	1.8
t_{14} (mm)	2.0	2.0	2.2	2.2	2.4	2.4
f (kg)	69.3	70.1	73.7	78.2	64.3	70.6
g_1 (mm)	155.4	152.6	113.4	82.4	137.4	112.2
g_2 (g)	30.0	28.2	28.7	28.4	31.9	24.8

Table B.7 Sample Set #1 – continued

	Sample Number					
	37	38	39	40	41	42
h_1 (mm)	130	90	110	130	90	110
b_1 (mm)	70	90	90	90	50	50
h_2 (mm)	90	70	90	50	90	50
b_2 (mm)	90	110	110	110	70	70
t_1 (mm)	2.0	2.2	2.4	2.0	2.2	2.4
t_2 (mm)	2.2	1.8	1.8	1.8	2.0	2.0
t_3 (mm)	1.8	1.8	2.0	1.6	1.6	1.8
t_4 (mm)	2.6	2.8	3.0	2.6	2.8	3.0
t_5 (mm)	3.0	3.2	2.8	3.0	3.2	2.8
t_6 (mm)	2.8	3.0	3.0	3.0	3.2	3.2
t_7 (mm)	1.4	1.0	1.2	1.4	1.0	1.2
t_8 (mm)	2.0	2.4	2.0	2.2	2.0	2.2
t_9 (mm)	2.8	2.8	2.4	2.6	2.6	2.8
t_{10} (mm)	2.8	2.8	2.8	2.8	2.8	2.8
t_{11} (mm)	2.8	3.0	3.0	3.0	3.2	3.2
t_{12} (mm)	2.4	2.2	2.4	2.6	2.4	2.6
t_{13} (mm)	2.0	2.2	1.8	2.0	2.2	1.8
t_{14} (mm)	2.4	2.0	2.0	2.0	2.2	2.2
f (kg)	78.6	75.0	83.1	79.7	63.6	62.6
g_1 (mm)	58.2	106.3	132.3	55.5	142.0	164.9
g_2 (g)	31.4	30.6	31.0	28.4	27.6	26.3

Table B.8 Sample Set #1 – continued

	Sample Number					
	43	44	45	46	47	48
h_1 (mm)	130	90	110	130	90	110
b_1 (mm)	50	90	90	90	50	50
h_2 (mm)	70	50	70	90	70	90
b_2 (mm)	70	90	90	90	110	110
t_1 (mm)	2.0	2.2	2.4	2.0	2.4	2.0
t_2 (mm)	2.0	1.8	1.8	1.8	2.2	2.2
t_3 (mm)	2.0	1.8	2.0	1.6	1.6	1.8
t_4 (mm)	2.6	2.8	3.0	2.6	2.8	3.0
t_5 (mm)	3.0	3.2	2.8	3.0	2.8	3.0
t_6 (mm)	3.2	2.8	2.8	2.8	3.0	3.0
t_7 (mm)	1.4	1.0	1.2	1.4	1.4	1.0
t_8 (mm)	2.4	2.4	2.0	2.2	2.0	2.2
t_9 (mm)	2.4	2.4	2.6	2.8	2.8	2.4
t_{10} (mm)	2.8	3.0	3.0	3.0	3.0	3.0
t_{11} (mm)	3.2	3.2	3.2	3.2	2.8	2.8
t_{12} (mm)	2.2	2.4	2.6	2.2	2.6	2.2
t_{13} (mm)	2.0	2.2	1.8	2.0	2.0	2.2
t_{14} (mm)	2.2	2.2	2.2	2.2	2.4	2.4
f (kg)	68.8	69.0	78.1	82.9	66.8	73.0
g_1 (mm)	172.0	142.3	80.1	60.3	148.8	128.9
g_2 (g)	31.3	27.6	31.8	34.2	30.6	27.9

Table B.9 Sample Set #1 – continued

	Sample Number					
	49	50	51	52	53	54
h_1 (mm)	130	90	110	130	110	130
b_1 (mm)	50	70	70	70	70	90
h_2 (mm)	50	90	50	70	50	50
b_2 (mm)	110	70	70	70	90	110
t_1 (mm)	2.2	2.4	2.0	2.2	2.2	2.4
t_2 (mm)	2.2	2.2	2.2	2.2	2.0	2.2
t_3 (mm)	2.0	2.0	1.6	1.8	1.8	2.0
t_4 (mm)	2.6	2.8	3.0	2.6	2.8	3.0
t_5 (mm)	3.2	2.8	3.0	3.2	3.0	3.2
t_6 (mm)	3.0	3.2	3.2	3.2	2.8	2.8
t_7 (mm)	1.2	1.4	1.0	1.2	1.0	1.0
t_8 (mm)	2.4	2.2	2.4	2.0	2.2	2.4
t_9 (mm)	2.6	2.6	2.8	2.4	2.6	2.8
t_{10} (mm)	3.0	3.0	3.0	3.0	2.6	2.6
t_{11} (mm)	2.8	3.0	3.0	3.0	2.8	2.8
t_{12} (mm)	2.4	2.2	2.6	2.6	2.2	2.2
t_{13} (mm)	1.8	2.0	1.8	1.8	1.8	1.8
t_{14} (mm)	2.4	2.0	2.0	2.0	2.2	2.4
f (kg)	72.6	68.9	66.6	74.9	66.0	81.9
g_1 (mm)	75.5	254.4	185.9	67.8	123.9	41.5
g_2 (g)	26.0	29.9	30.7	31.5	28.0	32.2

Table B.10 Sample Set #2

	Sample Number					
	1	2	3	4	5	6
h_1 (mm)	90	110	130	110	130	90
b_1 (mm)	50	70	70	70	90	70
h_2 (mm)	50	50	70	50	50	70
b_2 (mm)	70	70	70	90	110	70
t_1 (mm)	2.0	2.0	2.2	2.2	2.4	2.4
t_2 (mm)	1.8	2.2	2.2	2.0	2.2	1.8
t_3 (mm)	1.6	1.6	1.8	1.8	2.0	2.0
t_4 (mm)	2.6	3.0	2.6	2.8	3.0	2.6
t_5 (mm)	2.8	3.0	3.2	3.0	3.2	3.0
t_6 (mm)	2.8	3.2	3.2	2.8	2.8	3.0
t_7 (mm)	1.0	1.0	1.2	1.0	1.0	1.2
t_8 (mm)	2.0	2.4	2.0	2.2	2.4	2.2
t_9 (mm)	2.4	2.8	2.4	2.6	2.8	2.4
t_{10} (mm)	2.6	3.0	3.0	2.6	2.6	2.6
t_{11} (mm)	2.8	3.0	3.0	2.8	2.8	3.0
t_{12} (mm)	2.2	2.4	2.6	2.2	2.2	2.4
t_{13} (mm)	1.8	2.2	1.8	1.8	1.8	2.0
t_{14} (mm)	2.0	2.0	2.0	2.2	2.4	2.0
f (kg)	51.1	66.6	74.9	66.0	81.9	63.6
g_1 (mm)	276.4	185.9	67.8	123.9	41.4	202.2
g_2 (g)	23.7	30.7	31.5	28	32.2	35.9

Table B.11 Sample Set #2 – continued

	Sample Number					
	7	8	9	10	11	12
h_1 (mm)	110	130	90	110	130	90
b_1 (mm)	90	50	90	50	70	50
h_2 (mm)	70	70	90	90	90	50
b_2 (mm)	90	110	70	90	110	70
t_1 (mm)	2.0	2.2	2.2	2.4	2.0	2.0
t_2 (mm)	2.0	2.2	1.8	2.0	2.2	2.0
t_3 (mm)	1.6	1.8	1.8	2.0	1.6	1.6
t_4 (mm)	2.8	3.0	2.6	2.8	3.0	3.0
t_5 (mm)	3.2	2.8	3.2	2.8	3.0	3.0
t_6 (mm)	3.0	3.0	3.2	3.2	3.2	2.8
t_7 (mm)	1.2	1.2	1.4	1.4	1.4	1.2
t_8 (mm)	2.4	2.0	2.4	2.0	2.2	2.0
t_9 (mm)	2.6	2.8	2.4	2.6	2.8	2.6
t_{10} (mm)	2.6	2.6	2.6	2.6	2.6	2.8
t_{11} (mm)	3.0	3.0	3.2	3.2	3.2	3.2
t_{12} (mm)	2.4	2.4	2.6	2.6	2.6	2.6
t_{13} (mm)	2.0	2.0	2.2	2.2	2.0	2.0
t_{14} (mm)	2.2	2.4	2.2	2.2	2.4	2.4
f (kg)	75.0	75.4	72.9	73.1	84.9	55.2
g_1 (mm)	62.0	94.4	95.3	202.2	38.0	152.5
g_2 (g)	32.0	27.2	34.4	27.7	36.4	28.4

Table B.12 Sample Set #2 – continued

	Sample Number					
	13	14	15	16	17	18
h_1 (mm)	110	130	90	110	130	90
b_1 (mm)	70	90	70	90	50	90
h_2 (mm)	50	50	70	70	70	90
b_2 (mm)	90	110	70	90	110	70
t_1 (mm)	2.2	2.4	2.4	2.0	2.2	2.2
t_2 (mm)	2.2	1.8	2.0	2.2	1.8	2.0
t_3 (mm)	1.8	2.0	2.0	1.6	1.8	2.6
t_4 (mm)	2.6	2.8	3.0	2.6	2.8	3.0
t_5 (mm)	3.2	2.8	3.2	2.8	3.0	2.8
t_6 (mm)	2.8	2.8	3.0	3.0	3.0	3.2
t_7 (mm)	1.2	1.2	1.4	1.4	1.4	1.0
t_8 (mm)	2.2	2.4	2.2	2.4	2.0	2.4
t_9 (mm)	2.8	2.4	2.6	2.8	2.4	2.6
t_{10} (mm)	2.8	2.8	2.8	2.8	2.8	2.8
t_{11} (mm)	3.2	3.2	2.8	2.8	2.8	3.0
t_{12} (mm)	2.6	2.6	2.2	2.2	2.2	2.4
t_{13} (mm)	2.0	2.0	2.2	2.2	2.2	1.8
t_{14} (mm)	2.0	2.2	2.4	2.0	2.2	2.4
f (kg)	69.5	82.5	65.7	74.6	74.6	71.7
g_1 (mm)	99.7	94.6	174.7	171.6	96.0	98.0
g_2 (g)	26.0	30.7	33.9	30.9	28.7	30.4

Table B.13 Sample Set #2 – continued

	Sample Number					
	19	20	21	22	23	24
h_1 (mm)	110	130	90	110	130	90
b_1 (mm)	50	70	50	70	90	70
h_2 (mm)	90	90	50	50	50	70
b_2 (mm)	90	110	90	110	70	90
t_1 (mm)	2.4	2.0	2.0	2.2	2.4	2.4
t_2 (mm)	2.2	1.8	1.8	2.0	2.2	1.8
t_3 (mm)	2.0	1.6	2.0	1.6	1.8	1.8
t_4 (mm)	2.6	2.8	3.0	2.6	2.8	3.0
t_5 (mm)	3.0	3.2	2.8	3.0	3.2	3.0
t_6 (mm)	3.2	3.2	3.0	3.0	3.0	3.2
t_7 (mm)	1.0	1.0	1.0	1.0	1.0	1.2
t_8 (mm)	2.0	2.2	2.2	2.4	2.0	2.4
t_9 (mm)	2.8	2.4	2.8	2.4	2.6	2.8
t_{10} (mm)	2.8	2.8	3.0	3.0	3.0	3.0
t_{11} (mm)	3.0	3.0	3.0	3.0	3.0	3.2
t_{12} (mm)	2.4	2.4	2.6	2.6	2.6	2.2
t_{13} (mm)	1.8	1.8	2.2	2.2	2.2	1.8
t_{14} (mm)	2.0	2.2	2.2	2.4	2.0	2.2
f (kg)	71.9	81.6	60.1	72.2	78.8	68.8
g_1 (mm)	264.8	35.1	185.9	117.1	54.2	119.4
g_2 (g)	26.0	32.3	27.2	26.7	31.4	28.8

Table B.14 Sample Set #2 – continued

	Sample Number					
	25	26	27	28	29	30
h_1 (mm)	110	130	90	110	130	90
b_1 (mm)	90	50	90	50	70	50
h_2 (mm)	70	70	90	90	90	50
b_2 (mm)	110	70	90	110	70	90
t_1 (mm)	2.0	2.2	2.2	2.4	2.0	2.0
t_2 (mm)	2.0	2.2	1.8	2.0	2.2	1.8
t_3 (mm)	2.0	1.6	1.6	1.8	2.0	1.8
t_4 (mm)	2.6	2.8	3.0	2.6	2.8	3.0
t_5 (mm)	3.2	2.8	3.2	2.8	3.0	3.2
t_6 (mm)	3.2	3.2	2.8	2.8	2.8	3.2
t_7 (mm)	1.2	1.2	1.4	1.4	1.4	1.4
t_8 (mm)	2.0	2.2	2.0	2.2	2.4	2.4
t_9 (mm)	2.4	2.6	2.8	2.4	2.6	2.8
t_{10} (mm)	3.0	3.0	3.0	3.0	3.0	2.6
t_{11} (mm)	3.2	3.2	2.8	2.8	2.8	3.0
t_{12} (mm)	2.2	2.2	2.4	2.4	2.4	2.2
t_{13} (mm)	1.8	1.8	2.0	2.0	2.0	2.0
t_{14} (mm)	2.4	2.0	2.2	2.4	2.0	2.2
f (kg)	79.2	68.9	75.	73.2	77.0	58.6
g_1 (mm)	67.2	214.7	88.0	143.6	136.1	189.8
g_2 (g)	29.3	24.3	29.5	26.0	37.5	27.7

Table B.15 Sample Set #2 – continued

	Sample Number					
	31	32	33	34	35	36
h_1 (mm)	110	130	90	110	130	90
b_1 (mm)	70	90	70	90	50	90
h_2 (mm)	50	50	70	70	70	90
b_2 (mm)	110	70	110	70	90	110
t_1 (mm)	2.2	2.4	2.4	2.0	2.2	2.2
t_2 (mm)	2.0	2.2	2.2	1.8	2.0	2.2
t_3 (mm)	2.0	1.6	1.6	1.8	2.0	2.0
t_4 (mm)	2.6	2.8	2.6	2.8	3.0	2.6
t_5 (mm)	2.8	3.0	2.8	3.0	3.2	3.0
t_6 (mm)	3.2	3.2	2.8	2.8	2.8	3.0
t_7 (mm)	1.4	1.4	1.0	1.0	1.0	1.2
t_8 (mm)	2.0	2.2	2.0	2.2	2.4	2.2
t_9 (mm)	2.4	2.6	2.6	2.8	2.4	2.6
t_{10} (mm)	2.6	2.6	2.6	2.6	2.6	2.6
t_{11} (mm)	3.0	3.0	3.2	3.2	3.2	2.8
t_{12} (mm)	2.2	2.2	2.4	2.4	2.4	2.6
t_{13} (mm)	2.0	2.0	2.2	2.2	2.2	1.8
t_{14} (mm)	2.4	2.0	2.2	2.4	2.0	2.2
f (kg)	70.5	76.1	69.3	72.3	73.4	76.8
g_1 (mm)	152.3	123.4	139.8	68.0	106.5	116.6
g_2 (g)	31.1	28.2	33.3	32.2	29.2	29.8

Table B.16 Sample Set #2 – continued

	Sample Number					
	37	38	39	40	41	42
h_1 (mm)	130	90	110	130	90	110
b_1 (mm)	70	50	70	90	70	90
h_2 (mm)	90	50	50	50	70	70
b_2 (mm)	90	90	110	70	90	110
t_1 (mm)	2.0	2.0	2.2	2.4	2.4	2.0
t_2 (mm)	2.0	2.2	1.8	2.0	2.2	1.8
t_3 (mm)	1.8	2.0	1.6	1.8	1.8	2.0
t_4 (mm)	3.0	2.8	3.0	2.6	2.8	3.0
t_5 (mm)	2.8	3.2	2.8	3.0	2.8	3.0
t_6 (mm)	3.0	3.0	3.0	3.0	3.2	3.2
t_7 (mm)	1.2	1.4	1.4	1.4	1.0	1.0
t_8 (mm)	2.0	2.2	2.4	2	2.4	2.0
t_9 (mm)	2.4	2.4	2.6	2.8	2.4	2.6
t_{10} (mm)	2.6	2.8	2.8	2.8	2.8	2.8
t_{11} (mm)	2.8	3.2	3.2	3.2	2.8	2.8
t_{12} (mm)	2.6	2.4	2.4	2.4	2.6	2.6
t_{13} (mm)	1.8	1.8	1.8	1.8	2.0	2.0
t_{14} (mm)	2.0	2.4	2.0	2.2	2.4	2.0
f (kg)	77.7	58.7	71.0	75.9	67.7	80.0
g_1 (mm)	69.4	151.5	122.2	61.1	147.5	88.9
g_2 (g)	28.0	27.9	25.0	32.5	32.4	30.4

Table B.17 Sample Set #2 – continued

	Sample Number					
	43	44	45	46	47	48
h_1 (mm)	130	90	110	130	90	110
b_1 (mm)	50	90	50	70	50	70
h_2 (mm)	70	90	90	90	50	50
b_2 (mm)	70	90	110	70	90	110
t_1 (mm)	2.2	2.2	2.4	2.0	2.0	2.2
t_2 (mm)	2.0	2.2	1.8	2.0	2.2	1.8
t_3 (mm)	1.6	1.6	1.8	2.0	1.8	2.0
t_4 (mm)	2.6	2.8	3.0	2.6	2.8	3.0
t_5 (mm)	3.2	3.0	3.2	2.8	3.0	3.2
t_6 (mm)	3.2	2.8	2.8	2.8	3.2	3.2
t_7 (mm)	1.0	1.2	1.2	1.2	1.2	1.2
t_8 (mm)	2.2	2.0	2.2	2.4	2.4	2.0
t_9 (mm)	2.8	2.4	2.6	2.8	2.4	2.6
t_{10} (mm)	2.8	2.8	2.8	2.8	3.0	3.0
t_{11} (mm)	2.8	3.0	3.0	3.0	2.8	2.8
t_{12} (mm)	2.6	2.2	2.2	2.2	2.4	2.4
t_{13} (mm)	2.0	2.2	2.2	2.2	2.2	2.2
t_{14} (mm)	2.2	2.4	2.0	2.2	2.4	2.0
f (kg)	69.8	73.4	74.5	75.9	59.2	74.3
g_1 (mm)	75.4	152.8	152.7	167.7	137.0	113.1
g_2 (g)	29.2	26.4	24.7	36.0	29.5	28.3

Table B.18 Sample Set #2 – continued

	Sample Number					
	49	50	51	52	53	54
h_1 (mm)	130	90	110	130	90	110
b_1 (mm)	90	70	90	50	90	50
h_2 (mm)	50	70	70	70	90	90
b_2 (mm)	70	110	70	90	110	70
t_1 (mm)	2.4	2.4	2.0	2.2	2.2	2.4
t_2 (mm)	2.0	2.0	2.2	1.8	2.0	2.2
t_3 (mm)	1.6	1.6	1.8	2.0	2.0	1.6
t_4 (mm)	2.6	2.8	3.0	2.6	2.8	3.0
t_5 (mm)	2.8	3.2	2.8	3.0	2.8	3.0
t_6 (mm)	3.2	2.8	2.8	2.8	3.0	3.0
t_7 (mm)	1.2	1.4	1.4	1.4	1.0	1.0
t_8 (mm)	2.2	2.0	2.2	2.4	2.2	2.4
t_9 (mm)	2.8	2.8	2.4	2.6	2.8	2.4
t_{10} (mm)	3.0	3.0	3.0	3.0	3.0	3.0
t_{11} (mm)	2.8	3.2	3.2	3.2	3.2	3.2
t_{12} (mm)	2.4	2.6	2.6	2.6	2.2	2.2
t_{13} (mm)	2.2	1.8	1.8	1.8	2.0	2.0
t_{14} (mm)	2.2	2.0	2.2	2.4	2.0	2.2
f (kg)	76.7	71.9	72.6	72.9	78.7	68.6
g_1 (mm)	72.1	127.0	101.7	97.2	129.1	143.9
g_2 (g)	32.3	26.7	29.0	32.5	32.7	23.6

APPENDIX C
Fuzzy Logic Sampling Adjustment Rules for Case Study 1

The following tables summarize the 238 fuzzy rules used in the case study 1. Values in a table indicate adjustments to the respective design variable. For example, Table C.1 represents the following five fuzzy rules for adjusting t_1:

If δ_1 is NH, **then** $\tilde{a} = -0.4$, $i_a = 5$

If δ_1 is NL, **then** $\tilde{a} = -0.2$, $i_a = 5$

If δ_1 is Z, **then** $\tilde{a} = +0.1$, $i_a = 5$

If δ_1 is PL, **then** $\tilde{a} = +0.2$, $i_a = 5$

If δ_1 is PH, **then** $\tilde{a} = +0.4$, $i_a = 5$

Table C.1 Fuzzy rules for adjusting t_1

Membership of δ_1				
NH	NL	Z	PL	PH
-0.4	-0.2	+0.1	+0.2	+0.4

Table C.2 Fuzzy rules for adjusting t_2

Membership of δ_3				
NH	NL	Z	PL	PH
-0.2	-0.1	+0.05	+0.1	+0.2

Membership of δ_4				
NH	NL	Z	PL	PH
			+0.1	+0.2

		Membership of δ_3				
		NH	NL	Z	PL	PH
Memb. of δ_4	NH					
	NL					
	Z					
	PL	+0.15	+0.10			
	PH	+0.20	+0.15			

Table C.3 Fuzzy rules for adjusting t_3

Membership of δ_5				
NH	NL	Z	PL	PH
-0.2	-0.1	+0.05	+0.1	+0.2

Membership of δ_6				
NH	NL	Z	PL	PH
			+0.1	+0.2

		Membership of δ_5				
		NH	NL	Z	PL	PH
Memb. of δ_6	NH					
	NL					
	Z					
	PL	+0.15	+0.10			
	PH	+0.20	+0.15			

Table C.4 Fuzzy rules for adjusting t_4

Membership of δ_{13}				
NH	NL	Z	PL	PH
-0.4	-0.2		+0.2	+0.4

Table C.5 Fuzzy rules for adjusting t_5

Membership of δ_7				
NH	NL	Z	PL	PH
-0.2	-0.1		+0.1	+0.2
Membership of δ_8				
NH	NL	Z	PL	PH
-0.2	-0.1		+0.1	+0.2

Table C.6 Fuzzy rules for adjusting t_6

Membership of δ_{14}				
NH	NL	Z	PL	PH
-0.4	-0.2		+0.2	+0.4

Table C.7 Fuzzy rules for adjusting t_7

Membership of δ_{15}				
NH	NL	Z	PL	PH
-0.4	-0.2		+0.2	+0.4

Table C.8 Fuzzy rules for adjusting t_8

Membership of δ_{16}				
NH	NL	Z	PL	PH
-0.4	-0.2		+0.2	+0.4

Table C.9 Fuzzy rules for adjusting t_9

Membership of δ_9				
NH	NL	Z	PL	PH
-0.2	-0.1		+0.1	+0.2
Membership of δ_{10}				
NH	NL	Z	PL	PH
-0.2	-0.1		+0.1	+0.2

Table C.10 Fuzzy rules for adjusting t_{10}

Membership of δ_2				
NH	NL	Z	PL	PH
-0.4	-0.2	+0.1	+0.2	+0.4

Table C.11 Fuzzy rules for adjusting t_{11}

Membership of δ_{11}				
NH	NL	Z	PL	PH
-0.2	-0.1		+0.1	+0.2
Membership of δ_{12}				
NH	NL	Z	PL	PH
-0.2	-0.1		+0.1	+0.2

Table C.12 Fuzzy rules for adjusting t_{12}

Membership of δ_{17}				
NH	NL	Z	PL	PH
-0.4	-0.2	-0.1	+0.2	+0.4

Table C.13 Fuzzy rules for adjusting t_{13}

Membership of δ_{18}				
NH	NL	Z	PL	PH
-0.4	-0.2	-0.1	+0.2	+0.4

Table C.14 Fuzzy rules for adjusting t_{14}

Membership of δ_{19}				
NH	NL	Z	PL	PH
-0.4	-0.2	-0.1	+0.2	+0.4

Table C.15 Fuzzy rules for adjusting h_1

		Membership of δ_3				
		NH	NL	Z	PL	PH
Memb. of δ_4	NH					
	NL					
	Z	+6.0	+3.0			
	PL	+9.0	+6.0	+3.0		
	PH	+12.0	+9.0	+6.0		

		Membership of δ_5				
		NH	NL	Z	PL	PH
Memb. of δ_6	NH					
	NL					
	Z	+6.0	+3.0			
	PL	+9.0	+6.0	+3.0		
	PH	+12.0	+9.0	+6.0		

		Membership of δ_7				
		NH	NL	Z	PL	PH
Memb. of δ_8	NH		-3.0	-6.0	-9.0	-12.0
	NL	+3.0		-3.0	-6.0	-9.0
	Z	+6.0	+3.0		-3.0	-6.0
	PL	+9.0	+6.0	+3.0		-3.0
	PH	+12.0	+9.0	+6.0	+3.0	

Table C.16 Fuzzy rules for adjusting b_1

		Membership of δ_3				
		NH	NL	Z	PL	PH
Memb. of δ_4	NH					
	NL					
	Z	-4.0	-2.0			
	PL	-6.0	-4.0	-2.0		
	PH	-8.0	-6.0	-4.0		

		Membership of δ_5				
		NH	NL	Z	PL	PH
Memb. of δ_6	NH					
	NL					
	Z	-4.0	-2.0			
	PL	-6.0	-4.0	-2.0		
	PH	-8.0	-6.0	-4.0		

		Membership of δ_7				
		NH	NL	Z	PL	PH
Memb. of δ_8	NH		+2.0	+4.0	+6.0	+8.0
	NL	-2.0		+2.0	+4.0	+6.0
	Z	-4.0	-2.0		+2.0	+4.0
	PL	-6.0	-4.0	-2.0		+2.0
	PH	-8.0	-6.0	-4.0	-2.0	

Table C.17 Fuzzy rules for adjusting h_2

		Membership of δ_9				
		NH	NL	Z	PL	PH
Memb. of δ_{10}	NH		-3.0	-6.0	-9.0	-12.0
	NL	+3.0		-3.0	-6.0	-9.0
	Z	+6.0	+3.0		-3.0	-6.0
	PL	+9.0	+6.0	+3.0		-3.0
	PH	+12.0	+9.0	+6.0	+3.0	

		Membership of δ_{11}				
		NH	NL	Z	PL	PH
Memb. of δ_{12}	NH		-3.0	-6.0	-9.0	-12.0
	NL	+3.0		-3.0	-6.0	-9.0
	Z	+6.0	+3.0		-3.0	-6.0
	PL	+9.0	+6.0	+3.0		-3.0
	PH	+12.0	+9.0	+6.0	+3.0	

Table C.18 Fuzzy rules for adjusting b_2

		Membership of δ_9				
		NH	NL	Z	PL	PH
Memb. of δ_{10}	NH		+2.0	+4.0	+6.0	+8.0
	NL	-2.0		+2.0	+4.0	+6.0
	Z	-4.0	-2.0		+2.0	+4.0
	PL	-6.0	-4.0	-2.0		+2.0
	PH	-8.0	-6.0	-4.0	-2.0	

		Membership of δ_{11}				
		NH	NL	Z	PL	PH
Memb. of δ_{12}	NH		+2.0	+4.0	+6.0	+8.0
	NL	-2.0		+2.0	+4.0	+6.0
	Z	-4.0	-2.0		+2.0	+4.0
	PL	-6.0	-4.0	-2.0		+2.0
	PH	-8.0	-6.0	-4.0	-2.0	

APPENDIX D
Source Code Header Files for the Automated Crash Mode Matching Algorithm

File: **RandGen.h**

```
//Includes
#ifndef H_CPP_RandGenRand
#define H_CPP_RandGenRand

#include <fstream>
using namespace std;
#include <cmath>



class RandGen
{
protected:


public:           //C++ class initializations

      //Constructor
      RandGen() {}

      //Destructor
      ~RandGen() {}

protected:

      void init_genrand(unsigned long s) {srand(s);}

      // generates a random number on [0,0xffffffff]-interval
      unsigned long genrand_int32() {return rand();}

      // generates a random number on (0,1)-real-interval
      double genrand_real3()
      {
            return (((double)genrand_int32()) + 0.5)*(1.0/32768.0);
      }

public:

      //Seeding Funcions
      void seed(unsigned long s) {init_genrand(s);}

      //Function to Generate a Uniformy distributed Random Number on the
(0,1)-real-interval
      double rand01() {return genrand_real3();}
      //Function to Generate a random number uniformaly distributed
within a range
      double randrng(double drmin, double drmax) {return drmin+(drmax-
drmin)*rand01();}

      //Function to Generate an integer Number uniformly distributed
witihin a range
      int randint(int irmin, int irmax)
            {return int(1.0*irmin+((irmax-irmin)+1.0-1e-8)*rand01());}

      //Function to generate a boolean random result, give probability
of returning true
      bool randbool(double dptrue)
            {if (rand01() <= dptrue) return true; else return false;}

      //Function to generate a poisson distributed random value
      int randpoisson(double daverage)
      {
            if (daverage<1.0e-7) daverage=1.0e-7;
            double drnd01=rand01();

            int icurchoice=0;
            double dcurfact=1.0;
            double dcurp=exp(-daverage)*pow(daverage, 0.0)/dcurfact;

            while (dcurp<drnd01) {

                  icurchoice++;
                  dcurfact = dcurfact * icurchoice;
                  dcurp+=exp(-daverage)*pow(daverage,
double(icurchoice))/dcurfact;
            }

            return icurchoice;
      }

      //Function to generate a normally distributed random number
      double randn(double daverage, double dsd)
      {
            double dpi=4.0*atan(1.0);

            double drnd1=rand01();
            double drnd2=rand01();
            double dy=sqrt(-2.0*log(drnd1))*cos(2.0*dpi*drnd2);

            return daverage + dsd*dy;
      }
};
```

```
#endif      //H_CPP_RandGenRand
```

**File:		rfuzzy.h**

```
//Includes
#ifndef H_CPP_RFuzzy
#define H_CPP_RFuzzy

#include "RandGen.h"




//Abstract class for Fuzzy membership function
class RFZMemberShipFnAbs
{
protected:

	/*Each derived class must define its own data storage, 
allocation/deallocatin
	and tuning, but all derived classes will eventually use a "number" 
of levels*/

	//Number of set levels
	int inlevels;

public:

	//Constructor
	RFZMemberShipFnAbs()
	{
		inlevels=0; //Is a flag for non-initilized state
	}
	//Destructor
	~RFZMemberShipFnAbs() {inlevels=0;}

	//Function to check if class is initialized
	bool isinitialized()
	{
		if (inlevels<=0) return false;
		else return true;
	}

	//Function to return the number of set levels
	int getnlevels() const {return inlevels;}
```

```
      //Virtual Function, for intializing memory allocation, given
number of set levels,
      //    also allows passing generic initialization data via a void
pointer
      virtual void init(int icnlevels, const void *pvinitdata=NULL) {}

      //Virtual Function for tuning the membership functions
      virtual void tune(const void *pvtunepar) {}
      //Virtual Function for tuning the membership functions
      virtual void tune(const double *pdtunepar) {}

      //Virtual equality operator to make a duplicate copy of another
      virtual void operator=(const RFZMemberShipFnAbs &other) {}

      //Virtual Function for writing into a file stream
      virtual void writeinf(ofstream *pfout) const {}
      //Virtual Function for reading contents out of a file stream
      virtual void readfromf(ifstream *pfin) {}


      //Virtual Function to check the membership value of an input in
one of the sets
      virtual double getmembershipval(double dinvalue, int isetlevelid)
const {return 0.0;}

      //Virtual Function to plot the contents in a tab-delimited text
file
      virtual void plotmemfn(const char *pcfname, int insteps=1001)
const {}
};



//Implementation class for Fuzzy Traingular membership function
class RFZMemberShipFnTrig : public RFZMemberShipFnAbs
{
protected:

      //Storage of set level centers
      double *pdlevelcenters;

public:

      //Constructor
      RFZMemberShipFnTrig()
      {
            RFZMemberShipFnAbs::RFZMemberShipFnAbs();
            pdlevelcenters=NULL;
      }
      //Destructor
```

```
	~RFZMemberShipFnTrig()
	{
		if (inlevels>0)
		{
			delete [inlevels] pdlevelcenters;
			pdlevelcenters=NULL;
		}
		RFZMemberShipFnAbs::~RFZMemberShipFnAbs();
	}


	//Initialization function
	void init(int icnlevels, const void *pvinitdata=NULL);

	//Function for tuning the membership functions
	void tune(const void *pvtunepar)
	{
		const double *pdtune=(const double *)(pvtunepar);
		tune(pdtune);
	}
	//Function for tuning the membership functions
	void tune(const double *pdtunepar);

	//equality operator to make a duplicate copy of another
	void operator=(const RFZMemberShipFnAbs &other);

	//Function for writing into a file stream
	void writeinf(ofstream *pfout) const;
	//Function for reading contents out of a file stream
	void readfromf(ifstream *pfin);


	//Function to check the membership value of an input in one of the
sets
	double getmembershipval(double dinvalue, int isetlevelid) const
	{
		if (isetlevelid<0) return 0.0;
		if (isetlevelid>=inlevels) return 0.0;

		if (isetlevelid==0)
		{
			if (dinvalue<=pdlevelcenters[0]) return 1.0;
			else if (dinvalue>=pdlevelcenters[1]) return 0.0;
			else return 1.0 - (dinvalue-
pdlevelcenters[0])/(pdlevelcenters[1]-pdlevelcenters[0]);
		}
		else if (isetlevelid==inlevels-1)
		{
			if (dinvalue>=pdlevelcenters[inlevels-1]) return 1.0;
```

```
                  else if (dinvalue<=pdlevelcenters[inlevels-2]) return
0.0;
                  else return (dinvalue-pdlevelcenters[inlevels-
2])/(pdlevelcenters[inlevels-1]-pdlevelcenters[inlevels-2]);
            }
            else
            {
                  if (dinvalue<=pdlevelcenters[isetlevelid-1]) return
0.0;
                  else if (dinvalue>=pdlevelcenters[isetlevelid+1])
return 0.0;
                  else if (dinvalue<=pdlevelcenters[isetlevelid]) return
(dinvalue-pdlevelcenters[isetlevelid-1])/(pdlevelcenters[isetlevelid]-
pdlevelcenters[isetlevelid-1]);
                  else return 1.0-(dinvalue-
pdlevelcenters[isetlevelid])/(pdlevelcenters[isetlevelid+1]-
pdlevelcenters[isetlevelid]);
            }
      }

      //Function to plot the contents in a tab-delimited text file
      void plotmemfn(const char *pcfname, int insteps=1001) const;
};



//Implementation class for Fuzzy Sigmoid membership function
class RFZMemberShipFnSigmoid : public RFZMemberShipFnAbs
{
protected:

      //Storage of level centers
      double *pdlevelcenters;

      //Storage of a-parameters
      double *pdapars;
      //Storage of b-parameters
      double *pdbpars;
      //Storage of c-parameters
      double *pdcpars;

      //a-parameter scaling value
      double daparscaling;


      //Utility function for sigmoid calculation
      double utlcalcsigmoid(double dx, double da, double dc) const
      {
            double dy=da*(dx-dc);
            if (dy>0.0) return 1.0/(1.0+exp(-dy));
```

```
			else return exp(dy)/(1.0+exp(dy));
		}

public:

		//Constructor
		RFZMemberShipFnSigmoid()
		{
			RFZMemberShipFnAbs::RFZMemberShipFnAbs();
			daparscaling=2.0;
		}
		//Destructor
		~RFZMemberShipFnSigmoid()
		{
			if (inlevels>0)
			{
				delete [inlevels] pdlevelcenters;
				delete [inlevels-1] pdapars;
				delete [inlevels] pdbpars;
				delete [inlevels-1] pdcpars;

				pdlevelcenters=NULL;
				pdapars=NULL;
				pdbpars=NULL;
				pdcpars=NULL;
			}

			RFZMemberShipFnAbs::~RFZMemberShipFnAbs();
		}


		//Initialization function
		void init(int icnlevels, const void *pvinitdata=NULL);

		//Function for tuning the membership functions
		void tune(const void *pvtunepar)
		{
			const double *pdtune=(const double *)(pvtunepar);
			tune(pdtune);
		}
		//Function for tuning the membership functions
		void tune(const double *pdtunepar);

		//equality operator to make a duplicate copy of another
		void operator=(const RFZMemberShipFnAbs &other);

		//Function for writing into a file stream
		void writeinf(ofstream *pfout) const;
		//Function for reading contents out of a file stream
		void readfromf(ifstream *pfin);
```

```
      //Function to check the membership value of an input in one of the
sets
      double getmembershipval(double dinvalue, int isetlevelid) const
      {
            if (isetlevelid<0) return 0.0;
            if (isetlevelid>=inlevels) return 0.0;

            if (isetlevelid==0)
            {
                  return pdbpars[0]*(1.0-utlcalcsigmoid(dinvalue,
pdapars[0], pdcpars[0]));
            }
            else if (isetlevelid==inlevels-1)
            {
                  return pdbpars[inlevels-1]*utlcalcsigmoid(dinvalue,
pdapars[inlevels-2], pdcpars[inlevels-2]);
            }
            else
            {
                  return pdbpars[isetlevelid]*utlcalcsigmoid(dinvalue,
pdapars[isetlevelid-1], pdcpars[isetlevelid-1])
                        *(1.0-utlcalcsigmoid(dinvalue,
pdapars[isetlevelid], pdcpars[isetlevelid]));
            }
      }

      //Function to plot the contents in a tab-delimited text file
      void plotmemfn(const char *pcfname, int insteps=1001) const;
};



//Class for Inputs fuzzifier (container of fuzzy membership function for
each input)
class RFZInputFuzzyfier
{
protected:

      //Number of input variables - also serves as memory allocation
flag
      int ininputs;

      //Storage of Defaulted membership functions - sigmoidal type
      RFZMemberShipFnSigmoid *psigmdftstore;
      //Storage of Pointers to membership functions
      RFZMemberShipFnAbs **ppmembershipfns;
      //Temporary storage of a current input vector
      double *pdtmpcurinput;

      //Function to free allocated memory
      void freemem();

public:

      //Constructor
      RFZInputFuzzyfier();
      //Destructor
      ~RFZInputFuzzyfier() {freemem();}


      //Initialization function - defaults to sigmoid memberships
      void init(int icninputs);
      //Function for initialization of memory allocation from file
stream
      void initf(ifstream *pfin);

      //Function to set pointer to a membership function class
      void setmembershipfn(int iinputid, RFZMemberShipFnAbs
*pmembershipfn);

      //Function to re-initialize a membership function
      void reinitmembershipfn(int iinputid, int inlevels, const void
*pvinitdata=NULL);

      //Function to tune a membership function
      void tunemembershipfn(int iinputid, const void *pvtunepar);
      //Function to tune a membership function
      void tunemembershipfn(int iinputid, const double *pdtunepar);

      //Function to re-initialize and tune (all) current membership
functions from file stream
      void tunemembershipfnsf(ifstream *pfin);


      //Function to check if class is ready to perform calculations
      bool isreadytocompute() const;


      //Function to write class contents to file stream
      void writef(ofstream *pfout) const;


      //Function accept an input vector and internally calculate its
membership values
      void setcurinput(const double *pdcurinput) const
      {
```

```
		if (!isreadytocompute()) return;

		const double *pdctmp=pdtmpcurinput;
		double *pdtmp=(double *)(pdctmp);

		for (int icount=0; icount<ininputs; icount++)
		{
			pdtmp[icount]=pdcurinput[icount];
		}
	}

	//Function to return the current input's membership value (for
inputID & level)
	double getcurmembershipvalue(int iinputid, int isetlevelid) const
	{
		if (!isreadytocompute()) return 0.0;

		if (iinputid<0) return 0.0;
		if (iinputid>=ininputs) return 0.0;

		return ppmembershipfns[iinputid]-
>getmembershipval(pdtmpcurinput[iinputid], isetlevelid);
	}

	//Function to return the number of inputs
	int getninputs() const {return ininputs;}

	//Function to return a constant pointer to the membership function
requested
	const RFZMemberShipFnAbs *getcptrmembershipfn(int iinputid) const
	{
		if (iinputid<0) return NULL;
		if (iinputid>=ininputs) return NULL;
		return ppmembershipfns[iinputid];
	}
};




//Class for Fuzzy elementary "boolean" expression
class RFZElemBooleanExpr
{
protected:

	//Input variable ID
	int iinputid;

	//Level ID to check
	int itargetlevel;
```

```
public:

      //Constructor
      RFZElemBooleanExpr()
      {
            iinputid=0;
            itargetlevel=0;
      }
      //Destructor
      ~RFZElemBooleanExpr() {}

      //Initialization function
      void init(int icinputid, int ictargetlevel)
      {
            iinputid=icinputid;
            itargetlevel=ictargetlevel;
      }
      //Function to write contents to file stream
      void writef(ofstream *pfout) const
      {
            *pfout << iinputid << char(9) << itargetlevel << endl;
      }
      //Function to read contents from file stream
      void readf(ifstream *pfin)
      {
            *pfin >> iinputid;
            *pfin >> itargetlevel;
      }


      //Function to check current expression value with regards to a set
of inputs' state
      double getexprvalue(const RFZInputFuzzyfier *pfuzzyinputstate)
const
      {
            return pfuzzyinputstate->getcurmembershipvalue(iinputid,
itargetlevel);
      }

      //Function to return ID of input variable
      int getinputid() const {return iinputid;}

      //Function to return target level for input variable
      int gettargetlevel() const {return itargetlevel;}
};



//Abstract class for fuzzy rule
```

```
class RFZFuzzyRuleAbs
{
protected:

      //Output variable ID
      int ioutputid;

      //Base value for output
      double doutbaseval;

public:

      //Constructor
      RFZFuzzyRuleAbs()
      {
            ioutputid=0;
            doutbaseval=0.0;
      }
      //Destructor
      ~RFZFuzzyRuleAbs() {}

      //Function to return Output ID
      int getoutputid() const {return ioutputid;}


      //Function to set output values
      void setoutputvalues(int icoutpitid, double dcoutbaseval)
      {
            ioutputid=icoutpitid;
            doutbaseval=dcoutbaseval;
      }


      //Virtual Function to compute output value
      virtual double getoutputval(const RFZInputFuzzyfier
*pfuzzyinputstate) const {return doutbaseval;}

      //Virtual Function to adjust the rule
      virtual void setexprvalues(const int *pipar, const double
*pdpar=NULL, const void *pvpar=NULL) {}

      //Virtual Function to write contents to a file stream
      virtual void writef(ofstream *pfout) const
      {
            *pfout << ioutputid << char(9) << doutbaseval << endl;
      }

      //Virtual Funciton to read contents from a file stream
      virtual void readf(ifstream *pfin)
      {
            *pfin >> ioutputid;
            *pfin >> doutbaseval;
      }


      //Virtual Function to return the highest ID of input variable in
expression
      virtual int getmaxinputid() const {return 0;}

      //Virtual Function to write rule in a nicely formatted form to a
file stream
      virtual void formatrule(ofstream *pfout) const {}

      //Virtual assigment operator to copy from another
      virtual void operator=(const RFZFuzzyRuleAbs &other) {}
};



//Implementation class for fuzzy rule, as a simple string of and-
operators as min vals
class RFZFuzzyRuleSStrAndasMin : public RFZFuzzyRuleAbs
{
protected:

      //Number of elementary expressions
      int inexprs;

      //Storage of elementary expressions
      RFZElemBooleanExpr *pexprs;


public:

      //Constructor
      RFZFuzzyRuleSStrAndasMin()
      {
            RFZFuzzyRuleAbs::RFZFuzzyRuleAbs();

            inexprs=1;
            pexprs=new RFZElemBooleanExpr[inexprs];
      }
      //Destructor
      ~RFZFuzzyRuleSStrAndasMin()
      {
            delete [inexprs] pexprs;

            RFZFuzzyRuleAbs::~RFZFuzzyRuleAbs();
      }
```

```
	//Function to compute output value
	double getoutputval(const RFZInputFuzzyfier *pfuzzyinputstate)
const
	{
		double dcurfact=1.0;
		double dcurval;

		for (int icount=0; icount<inexprs; icount++)
		{
			dcurval=pexprs[icount].getexprvalue(pfuzzyinputstate);
			if (dcurfact>dcurval) dcurfact=dcurval;
		}

		return dcurfact*doutbaseval;
	}

	//Function to adjust the rule
	void setexprvalues(const int *pipar, const double *pdpar=NULL,
const void *pvpar=NULL);

	//Function to write contents to a file stream
	void writef(ofstream *pfout) const;

	//Funciton to read contents from a file stream
	void readf(ifstream *pfin);

	//Function to return the highest ID of input variable in
expression
	int getmaxinputid() const
	{
		int imaxinid=0;
		for (int icount=0; icount<inexprs; icount++)
		{
			if (imaxinid<pexprs[icount].getinputid())
			{
				imaxinid=pexprs[icount].getinputid();
			}
		}
		return imaxinid;
	}

	//Function to write rule in a nicely formatted form to a file
stream
	void formatrule(ofstream *pfout) const
	{
		*pfout << "IF ";
		*pfout << "(InputID_" << pexprs[0].getinputid() << "
IsAtLevel "
			<< pexprs[0].gettargetlevel() << ")";
```

```
            for (int icount=1; icount<inexprs; icount++)
            {
                  *pfout << " AND (InputID_" << 
pexprs[icount].getinputid() << " IsAtLevel "
                        << pexprs[icount].gettargetlevel() << ")";
            }
            *pfout << " THEN OutputID_" << ioutputid << " IsAdujustedBy 
" << doutbaseval << endl;
      }

      //Assignment operator to copy from another
      void operator=(const RFZFuzzyRuleAbs &other);
};




//Class for Fuzzy Rule Base (collection of rules)
class RFZFuzzyRuleBase
{
protected:

      //Number of rules
      int inrules;

      //Storage of Defaulted rules - simple and-strings
      RFZFuzzyRuleSStrAndasMin *pandstrs;

      //Storage of Pointers to rules
      RFZFuzzyRuleAbs **pprules;


      //Function to free allocated memory
      void freemem();

public:

      //Constructor
      RFZFuzzyRuleBase();
      //Destructor
      ~RFZFuzzyRuleBase() {freemem();}


      //Function to reset all contents, and/or pre-allocate memory for a 
number of rules
      void reset(int icnrules=0);
      //Function to perform the reset from file stream
      void resetf(ifstream *pfin);
      //Function to set a rule (by linking pointers to appropriate rule
class)
      void setrule(int iruleid, RFZFuzzyRuleAbs *pruleobj);

      //Function to adjust exisiting rule's output properties
      void setruleoutputvalues(int iruleid, int icoutputid, double
dcoutbaseval);

      //Function to adjust existing rule's logical reasoning
      void setruleexprvalues(int iruleid, const int *pipar, const double
*pdpar=NULL,
            const void *pvpar=NULL);

      //Function to read the rules' data from file stream
      void setrulesf(ifstream *pfin);


      //Function to write class contents to file stream
      void writef(ofstream *pfout) const;

      //Function to provide a nicely formatted listing of Fuzzy rules to
a file
      void formatfuzzyrules(const char *pcfname) const;

      //Function to return the number of rules
      int getnrules() const {return inrules;}

      //Function to return a constant pointer to rule object
      const RFZFuzzyRuleAbs *getcptrrule(int iruleid) const {return
pprules[iruleid];}

      //Function to check the maximum input ID
      int getmaxinputid() const
      {
            int imaxinid=0;
            for (int icount=0; icount<inrules; icount++)
            {
                  if (imaxinid<pprules[icount]->getmaxinputid())
                  {
                        imaxinid=pprules[icount]->getmaxinputid();
                  }
            }
            return imaxinid;
      }
      //Function to check the maximum output ID
      int getmaxoutputid() const
      {
            int imaxoutid=0;
            for (int icount=0; icount<inrules; icount++)
            {
                  if (imaxoutid<pprules[icount]->getoutputid())
                  {
                        imaxoutid=pprules[icount]->getoutputid();
                  }
            }
            return imaxoutid;
      }
};




//Class for an output stochasitc variable activated by voting in rule
base
class RFZRandOutputVarViaRBVoting
{
protected:

      //Current Value
      double dcurval;

      //Minimum value for SD
      double dminsigma;

      //Current variable adjustments setting
      double dcurav;
      //Current variable adjustments absolute sum
      double dadjstabssum;

      //Current SD
      double dcursigma;

public:

      //Constructor
      RFZRandOutputVarViaRBVoting();
      //Destructor
      ~RFZRandOutputVarViaRBVoting() {}

      //Assignment operator to copy from another
      void operator=(const RFZRandOutputVarViaRBVoting &other);


      //Function to setup "large value", also resets voting
      void setminsigma(double dcminsigma) {dminsigma=dcminsigma;}
      //Function to set current value, also resets voting
      void setcurvalue(double dccurval);


      //Function to place a vote
```

```
	void votetochangeby(double dvotedchange);


	//Function to write the tunable (large value) to file stream
	void writef(ofstream *pfout) const
	{
		*pfout << dminsigma << endl;
	}
	//Function to read the tunable (min. SD) from file stream
	void readf(ifstream *pfin)
	{
		double dtmpval;
		*pfin >> dtmpval;
		setminsigma(dtmpval);
	}

	//Function to write current distribution values to file stream
	void writedistribtof(ofstream *pfout) const
	{
		*pfout << dcurval << char(9) << dcurav << char(9) <<
dcursigma << endl;
	}

	//Function to return current value
	double getcurval() const {return dcurval;}

	//Function to return current average value of votes
	double getcurav() const {return dcurav;}
	//Function to return current SD
	double getcursd() const {return dcursigma;}


	//Function to perform a Monte-Carlo instant of the variable
accoring to current distribution
	double getrandval(RandGen *prand) const
	{
		if (dcursigma<=0.0) return dcurav;
		return prand->randn(dcurav, dcursigma);
	}
	//Function to perform a Monte-Carlo instant of the variable
accoring to current distribution
	double getrandval(RandGen *prand, double drmin, double drmax)
const
	{
		double dmcval=getrandval(prand);
		if (dmcval<drmin) dmcval=drmin;
		if (dmcval>drmax) dmcval=drmax;
		return dmcval;
	}
```

```
      //Function to perform a Monte-Carlo instant of the variable
accoring to current distribution
      double getrandval(RandGen *prand, int inchoices, const double
*pdchoices) const
      {
            double dcval=getrandval(prand);

            int ichoiceid=0;
            double dbestdist=fabs(dcval-pdchoices[0]);
            double dcurdist;

            for (int icount=1; icount<inchoices; icount++)
            {
                  dcurdist=fabs(dcval-pdchoices[icount]);
                  if (dbestdist>dcurdist)
                  {
                        dbestdist=dcurdist;
                        ichoiceid=icount;
                  }
            }
            return pdchoices[ichoiceid];
      }
};



//Class for a set of output stochasitc variables activated by voting
from a rule base
class RFZRandOutputVarViaRBVotingSet
{
protected:

      //Number of output variables
      int inoutputs;

      //Storage of output variable objects
      RFZRandOutputVarViaRBVoting *poutvars;

public:

      //Constructor
      RFZRandOutputVarViaRBVotingSet()
      {
            inoutputs=0;
            poutvars=NULL;
      }

      //Destructor
      ~RFZRandOutputVarViaRBVotingSet()
      {
```

```
		if (inoutputs>0) delete [inoutputs] poutvars;
	}

	//Initialization function
	void init(int icnoutputs, const double *pdminsds=NULL);

	//Function to write output variables' "large value" tunable to a
file stream
	void writef(ofstream *pfout) const;

	//Function to read output variables' "large value" tunable from a
file stream
	void readf(ifstream *pfin);

	//Function to write current distribution values to file
	void writedistribtof(const char *pcfname) const
	{
		ofstream fout(pcfname);
		for (int icount=0; icount<inoutputs; icount++)
		{
			poutvars[icount].writedistribtof(&fout);
		}
		fout.close();
	}

	//Function to set output variables states to cope with input
states and rule base
	void settostate(const RFZInputFuzzyfier *pinputstate,
		const RFZFuzzyRuleBase *prulebase, const double
*pdcurvarvals)
	{
		if (inoutputs<=0) return;

		int icount;
		for (icount=0; icount<inoutputs; icount++)
		{
			poutvars[icount].setcurvalue(pdcurvarvals[icount]);
		}

		int inrules=prulebase->getnrules();
		const RFZFuzzyRuleAbs *pcurrule;

		for (icount=0; icount<inrules; icount++)
		{
			pcurrule=prulebase->getcptrrule(icount);
			poutvars[pcurrule-
>getoutputid()].votetochangeby(pcurrule->getoutputval(pinputstate));
		}
	}
```

```
      //Function to return the number of output variables
      int getnoutvars() const {return inoutputs;}
      //Function to return a constant pointer to output variables
      const RFZRandOutputVarViaRBVoting *getoutvars() const {return
poutvars;}
};


#endif      //H_CPP_RFuzzy
```

**File:	AbsCrashMode.h**

```
//Include section
#ifndef H_CPP_AbsCrashMode
#define H_CPP_AbsCrashMode

#include <fstream>
#include <cmath>
using namespace std;



//Pre-set constants

const double D_CrashModeBase_TimeInstantTol = 1.0e-7;
const double D_CrashModeHardLim_DeformVal = 1.0e-6;



//Base Class (abstracted) for definition of crash mode
class CrashModeBase
{
protected:

      //An identifier for Crash Mode Object "Type"
      int iobjtype;
      //Initialization flag
      int iisinitialized;
      //Solution availability flag
      int iissolavailable;

      //Number of zones
      int inzones;
      //Maximum number of deformation types per zone
      int inmaxdeformtypes;
      //Number of deformation tyoes in each zone
      int *pindeformtypes;
      //Number of "recordable time instances" for time
      int intimeinstt;
      //Number of "recordable time instances" for the deformation
      int intimeinstdef;
      //Storage of time instances
      double *pdtimeinst;
      //Storage of deformation instances
      double *pddefinst;


      //Utility function to allocate memory for CM data storage
      void utl_malloc(int icnzones, int icnmaxdeformtypes,
            int icntimeinstt, int icntimeinstdef)
      {
            utl_freemem();

            inzones=icnzones;
            inmaxdeformtypes=icnmaxdeformtypes;
            intimeinstt=icntimeinstt;
            intimeinstdef=icntimeinstdef;

            pindeformtypes=new int[inzones*inmaxdeformtypes];
            pdtimeinst=new double[intimeinstt];
            pddefinst=new
double[inzones*inmaxdeformtypes*intimeinstdef];
      }
      //Utility function to freem memory of CM data storage
      void utl_freemem()
      {
            if (inzones*inmaxdeformtypes>0) delete
[inzones*inmaxdeformtypes] pindeformtypes;
            if (intimeinstt>0) delete [intimeinstt] pdtimeinst;
            if (inzones*inmaxdeformtypes*intimeinstdef>0)
                  delete [inzones*inmaxdeformtypes*intimeinstdef]
pddefinst;

            inzones=0;
            inmaxdeformtypes=0;
            intimeinstt=0;
            intimeinstdef=0;

            pindeformtypes=NULL;
            pdtimeinst=NULL;
            pddefinst=NULL;

            iisinitialized=0;
            iissolavailable=0;
      }

      //Utility function to place a value in its correct ZDT position in
matrix
```

```
      void utl_placedfrmvalue(double ddfrmvalue, int izoneid, int
ideftypeid, int itimeinstid)
      {
            pddefinst[inzones*inmaxdeformtypes*itimeinstid +
izoneid*inmaxdeformtypes + ideftypeid]
                  = ddfrmvalue;
      }

public:

      //Constructor
      CrashModeBase()
      {
            iobjtype=0;                   //Object Type is set to "Base
Class"
            iisinitialized=0;       //Initialization is set to "OFF"

            inzones=0;
            inmaxdeformtypes=0;
            intimeinstt=0;
            intimeinstdef=0;

            pindeformtypes=NULL;
            pdtimeinst=NULL;
            pddefinst=NULL;
      }
      //Destructor
      ~CrashModeBase() {utl_freemem();}


      //Function to return identifier for object type
      int get_objtype() const {return iobjtype;}

      //Function to return the number of zones
      int get_nzones() const {return inzones;}
      //Function to return the maximum number of deformation types
      int get_nmaxdeformtypes() const {return inmaxdeformtypes;}
      //Function to return the number of deformation types in all zones
      const int *get_ndeformtypes() const {return pindeformtypes;}

      //Function to return the number of stored time instants
      int get_nstoretimeinstants() const {return intimeinstt;}
      //Function to return the number of stored deformation instants
      int get_nstoredefrominstants() const {return intimeinstdef;}

      //Function to return the stored time instant values
      const double *get_storetimeinstants() const {return pdtimeinst;}

      //Function to return object initialization state (returns true if
isinitialized==1)
```

```
      bool check_initialization() const {if (iisinitialized==1) return
true; else return false;}
      //Function to return solution availability state (returns true if
iissolavailable==1)
      bool check_solavailability() const {if (iissolavailable==1) return
true; else return false;}


                  /*Virtual Functions that need to be defined in derived
classes*/

      //Crash mode model initialization from data object
      virtual void init(const void *pvdata) {}
      //Crash mode model initialization from file
      virtual void initf(const char *pcmmodelfname) {}
      //Saving model to file
      virtual void savemodel(const char *pcmmodelfname) const {}

      //Extracting CM values from a solution storage object (can also be
a file)
      virtual void extractCM(const void *pvsolobj) {}

      //Writing CM values to file
      virtual void saveCMvals(const char *pcfname) const {}
      //Reading CM values from file
      virtual void readCMvals(const char *pcfname) {}


                  /*Virtual Functions that "may" need to be re-defined
in derived classes*/

      //Function to return "equivalent deformation value", given zone,
deformation type & time
      virtual double getcmvalue_zdt(int izoneid, int ideftypeid, int
itimeinstid) const
      {
            if (iisinitialized!=1) return -1.0;
            if (iissolavailable!=1) return -2.0;
            if ((izoneid<0)||(izoneid>=inzones)) return -3.0;
            if ((ideftypeid<0)||(ideftypeid>=pindeformtypes[izoneid]))
return -4.0;
            if ((itimeinstid<0)||(itimeinstid>=intimeinstdef)) return -
5.0;

            return pddefinst[inzones*inmaxdeformtypes*itimeinstid +
                                    izoneid*inmaxdeformtypes +
ideftypeid];
      }
      //Function to return "equivalent deformation value", given zone,
deformation type & time
```

```
      virtual double getcmvalue_zdt(int izoneid, int ideftypeid, double
dtimeinst) const
      {
            if (iisinitialized!=1) return -1.0;
            if (iissolavailable!=1) return -2.0;
            if ((izoneid<0)||(izoneid>=inzones)) return -3.0;
            if ((ideftypeid<0)||(ideftypeid>=pindeformtypes[izoneid]))
return -4.0;
            if (intimeinstt!=intimeinstdef) return -5.0;
            if (intimeinstt<2) return -6.0;

            int itid1=0;
            int itid2=1;
            for (int icount=0; icount<intimeinstt-1; icount++)
            {
                  if (pdtimeinst[itid2]>=dtimeinst) break;
                  itid1=itid2;
                  itid2++;
            }

            double dval1 = pddefinst[inzones*inmaxdeformtypes*itid1 +
izoneid*inmaxdeformtypes + ideftypeid];
            double dval2 = pddefinst[inzones*inmaxdeformtypes*itid2 +
izoneid*inmaxdeformtypes + ideftypeid];
            double drrel = (dtimeinst-
pdtimeinst[itid1])/(pdtimeinst[itid2]-pdtimeinst[itid1]);
            return (1.0-drrel)*dval1 + drrel*dval2;
      }
      //Function to return "equivalent deformation value", given zone &
deformation type (final time values)
      virtual double getcmvalue_zd(int izoneid, int ideftypeid) const
      {
            if (iisinitialized!=1) return -1.0;
            if (iissolavailable!=1) return -2.0;

            int inlaststeady=intimeinstdef/10;
            if (inlaststeady<1) inlaststeady=1;

            double dsumsteady=0.0;
            for (int icount=0; icount<inlaststeady; icount++)
            {
                  dsumsteady+=getcmvalue_zdt(izoneid, ideftypeid,
intimeinstdef-1-icount);
            }

            return dsumsteady/double(inlaststeady);
      }
      //Function to return "equivalent deformation value", given zone &
time (summed over deformation types)
      virtual double getcmvalue_zt(int izoneid, int itimeinstid) const
      {
            if (iisinitialized!=1) return -1.0;
            if (iissolavailable!=1) return -2.0;

            double dsumdef=0.0;
            int indeftypes=pindeformtypes[izoneid];
            for (int icount=0; icount<indeftypes; icount++)
            {
                  dsumdef+=getcmvalue_zdt(izoneid, icount, itimeinstid);
            }
            return dsumdef;
      }
      //Function to return "equivalent deformation value", given zone &
time (summed over deformation types)
      virtual double getcmvalue_zt(int izoneid, double dtimeinst) const
      {
            if (iisinitialized!=1) return -1.0;
            if (iissolavailable!=1) return -2.0;

            double dsumdef=0.0;
            int indeftypes=pindeformtypes[izoneid];
            for (int icount=0; icount<indeftypes; icount++)
            {
                  dsumdef+=getcmvalue_zdt(izoneid, icount, dtimeinst);
            }
            return dsumdef;
      }
      //Function to return "equivalent deformation value", given zone
(final time values - summed over deformation types)
      virtual double getcmvalue_z(int izoneid) const
      {
            if (iisinitialized!=1) return -1.0;
            if (iissolavailable!=1) return -2.0;

            double dsumdef=0.0;
            int indeftypes=pindeformtypes[izoneid];
            for (int icount=0; icount<indeftypes; icount++)
            {
                  dsumdef+=getcmvalue_zd(izoneid, icount);
            }
            return dsumdef;
      }
};



//Sub-category of CrashModeBase: Detailed deformation history crash mode
curves
class CrashModeDetTHist : public CrashModeBase
{
```

```
public:

      //Constructor
      CrashModeDetTHist()
      {
            CrashModeBase::CrashModeBase();

            iobjtype=1;                   //Object Type is set to
"Detailed Time History"
            iisinitialized=1;       //Initialization is set to "ON"
      }
      //Destructor
      ~CrashModeDetTHist() {CrashModeBase::~CrashModeBase();}

      //Writing CM values to file
      void saveCMvals(const char *pcfname) const;
      //Reading CM values from file
      void readCMvals(const char *pcfname);
};



//Sub-category of CrashModeBase: Deformation history approximated via
hard-limit basis
class CrashModeHardLim : public CrashModeBase
{
protected:

      //Utility function for extraction and storage of CM values
      void extractCMpcmp(const double *pdmags, const double *pdtinsts,
            int icnzones, double dclastime);
public:

      //Constructor
      CrashModeHardLim()
      {
            CrashModeBase::CrashModeBase();

            iobjtype=2;                   //Object Type is set to "Hard-
limit approximation"
            iisinitialized=1;       //Initialization is set to "ON"
      }
      //Destructor
      ~CrashModeHardLim() {CrashModeBase::~CrashModeBase();}


      //Writing CM values to file
      void saveCMvals(const char *pcfname) const;
      //Reading CM values from file
      void readCMvals(const char *pcfname);
      //Function to convert from a detailed CM history
      void convertfromdethist(const CrashModeDetTHist *pdetcmhistobj);


      //Function to return "equivalent deformation value", given zone,
deformation type & time Instant ID
      double getcmvalue_zdt(int izoneid, int ideftypeid, int
itimeinstid) const
      {
            if (iisinitialized!=1) return -1.0;
            if (iissolavailable!=1) return -2.0;
            if ((izoneid<0)||(izoneid>=inzones)) return -3.0;
            if ((ideftypeid<0)||(ideftypeid>=pindeformtypes[izoneid]))
return -4.0;

            if (itimeinstid==0)
            {
                  return 0.0;
            }
            else
            {
                  return pddefinst[izoneid*inmaxdeformtypes +
ideftypeid];
            }
      }
      //Function to return "equivalent deformation value", given zone,
deformation type & time
      double getcmvalue_zdt(int izoneid, int ideftypeid, double
dtimeinst) const
      {
            if (iisinitialized!=1) return -1.0;
            if (iissolavailable!=1) return -2.0;
            if ((izoneid<0)||(izoneid>=inzones)) return -3.0;
            if ((ideftypeid<0)||(ideftypeid>=pindeformtypes[izoneid]))
return -4.0;

            double djumpinstant=pddefinst[inzones*inmaxdeformtypes +
izoneid*inmaxdeformtypes + ideftypeid];

            if (dtimeinst<djumpinstant-D_CrashModeBase_TimeInstantTol)
            {
                  return 0.0;
            }
            else if
(dtimeinst>djumpinstant+D_CrashModeBase_TimeInstantTol)
            {
                  return pddefinst[izoneid*inmaxdeformtypes +
ideftypeid];
```

```
            }
            else
            {
                  double drelval=0.5*(dtimeinst-
djumpinstant+D_CrashModeBase_TimeInstantTol)/D_CrashModeBase_TimeInstant
Tol;
                  return drelval*pddefinst[izoneid*inmaxdeformtypes +
ideftypeid];
            }
      }
      //Function to return "equivalent deformation value", given zone &
deformation type (final time values)
      double getcmvalue_zd(int izoneid, int ideftypeid) const
      {
            if (iisinitialized!=1) return -1.0;
            if (iissolavailable!=1) return -2.0;
            return getcmvalue_zdt(izoneid, ideftypeid, 1);
      }
      //Function to return "equivalent deformation value", given zone &
time (summed over deformation types)
      double getcmvalue_zt(int izoneid, int itimeinstid) const
      {
            if (iisinitialized!=1) return -1.0;
            if (iissolavailable!=1) return -2.0;

            if (itimeinstid<=0) itimeinstid=0;
            else itimeinstid=1;

            double dsumdef=0.0;
            int indeftypes=pindeformtypes[izoneid];
            for (int icount=0; icount<indeftypes; icount++)
            {
                  dsumdef+=getcmvalue_zdt(izoneid, icount, itimeinstid);
            }
            return dsumdef;
      }
      //Function to return "equivalent deformation value", given zone &
time (summed over deformation types)
      double getcmvalue_zt(int izoneid, double dtimeinst) const
      {
            if (iisinitialized!=1) return -1.0;
            if (iissolavailable!=1) return -2.0;

            double dsumdef=0.0;
            int indeftypes=pindeformtypes[izoneid];
            for (int icount=0; icount<indeftypes; icount++)
            {
                  dsumdef+=getcmvalue_zdt(izoneid, icount, dtimeinst);
            }
            return dsumdef;
```

```
	}
	//Function to return "equivalent deformation value", given zone
(final time values - summed over deformation types)
	double getcmvalue_z(int izoneid) const
	{
		if (iisinitialized!=1) return -1.0;
		if (iissolavailable!=1) return -2.0;

		double dsumdef=0.0;
		int indeftypes=pindeformtypes[izoneid];
		for (int icount=0; icount<indeftypes; icount++)
		{
			dsumdef+=getcmvalue_zd(izoneid, icount);
		}
		return dsumdef;
	}

};


#endif		//H_CPP_AbsCrashMode
```

File:		**autocmmatcher.h**

```
//Include section
#ifndef H_CPP_AutoCMMatcher
#define H_CPP_AutoCMMatcher

#include "rfuzzy.h"
#include "AbsCrashMode.h"


//Class for Fuzzy Sampler
class ACMMFuzzySampler
{
protected:

	//Membership functions for Input Fuzzification
	RFZInputFuzzyfier inputfuzzifier;

	//Fuzzy Rule Base
	RFZFuzzyRuleBase fuzzyrules;

	//Rand variales for sampling
	RFZRandOutputVarViaRBVotingSet randvarsset;


public:
	//Constructor
	ACMMFuzzySampler() {}
	//Destructor
	~ACMMFuzzySampler() {}


	//Function to return a pointer to input fuzzifier object
	RFZInputFuzzyfier *getptr_inputfuzzifier() {return
&inputfuzzifier;}
	//Function to return a pointer to fuzzy rule base
	RFZFuzzyRuleBase *getptr_fuzzyrules() {return &fuzzyrules;}
	//Function to return a pointer to the random variales for sampling
	RFZRandOutputVarViaRBVotingSet *getptr_randvarssetr() {return
&randvarsset;}

	//Function to return a constant pointer to input fuzzifier object
	const RFZInputFuzzyfier *getcptr_inputfuzzifier() const {return
&inputfuzzifier;}
	//Function to return a constant pointer to fuzzy rule base
	const RFZFuzzyRuleBase *getcptr_fuzzyrules() const {return
&fuzzyrules;}
	//Function to return a constant pointer to the random variales for
sampling
	const RFZRandOutputVarViaRBVotingSet *getcptr_randvarssetr() const
{return &randvarsset;}


	//Function to write contents to a file
	void writef(const char *pcfname) const;
	//Function to read contents from a file
	void readf(const char *pcfname);

	//Function to write current distribution of the Gauusian variables
to file
	void writedistribtof(const char *pcfname) const
	{
		randvarsset.writedistribtof(pcfname);
	}

	//Function to return the number of inputs
	int getninputerrs() const {return inputfuzzifier.getninputs();}
	//Function to return the number of rules
	int getnrules() const {return fuzzyrules.getnrules();}
	//Function to return the number of controlled/sampled/output
variables
	int getnoutputcrtl() const {return randvarsset.getnoutvars();}


	//Function to check for I/O consistency
	bool isconsistent() const
	{
		if (!inputfuzzifier.isreadytocompute()) return false;

		int ininputerrs=inputfuzzifier.getninputs();
		int inoutputcrtl=randvarsset.getnoutvars();

		if (fuzzyrules.getmaxinputid()>=ininputerrs) return false;
		if (fuzzyrules.getmaxoutputid()>=inoutputcrtl) return false;

		return true;
	}

	//Function to plot the membership functions of an input to a file
	void plotinputmemberships(const char *pcfname, int iinputid, int 
inplotpoints=1001) const
	{
		if (iinputid<0) return;
		if (iinputid>=inputfuzzifier.getninputs()) return;

		inputfuzzifier.getcptrmembershipfn(iinputid)-
>plotmemfn(pcfname, inplotpoints);
	}


	//Function to put a nicely formatted listing of Fuzzy rules into a 
file
	void formatfuzzyrules(const char *pcfname) const
	{
		fuzzyrules.formatfuzzyrules(pcfname);
	}


	//Function to ready class for Monte-Carlo sampling
	void readysampling(const double *pdcurinputerrs, const double 
*pdcuroutdesvar)
	{
		inputfuzzifier.setcurinput(pdcurinputerrs);
		randvarsset.settostate(&inputfuzzifier, &fuzzyrules, 
pdcuroutdesvar);
	}
	//Function to produce a Monte-Carlo sample of a design variable 
(output)
	double randsamplevar(int ioutvarid, RandGen *prand)
	{
		int inoutvar=randvarsset.getnoutvars();
		if (ioutvarid<0) return 0.0;
		if (ioutvarid>=inoutvar) return 0.0;
		return 
randvarsset.getoutvars()[ioutvarid].getrandval(prand);
	}
```

```
      //Function to produce a Monte-Carlo sample of a design variable
(output)
      double randsamplevar(int ioutvarid, RandGen *prand, double drmin,
double drmax)
      {
            int inoutvar=randvarsset.getnoutvars();
            if (ioutvarid<0) return 0.0;
            if (ioutvarid>=inoutvar) return 0.0;
            return randvarsset.getoutvars()[ioutvarid].getrandval(prand,
drmin, drmax);
      }
      //Function to produce a Monte-Carlo sample of a design variable
(output)
      double randsamplevar(int ioutvarid, RandGen *prand, int inchoices,
const double *pdchoices)
      {
            int inoutvar=randvarsset.getnoutvars();
            if (ioutvarid<0) return 0.0;
            if (ioutvarid>=inoutvar) return 0.0;
            return randvarsset.getoutvars()[ioutvarid].getrandval(prand,
inchoices, pdchoices);
      }
};



//Class for ACMM design variable definition
class ACMMDesignVarDef
{
protected:

      //Allocated memory size
      int inalloc;
      //Number of "choices", =0 for continuous variables
      int inchoices;

      //Storage of choices, 1st 2 slots used for min/max in case of
continuous variable
      double *pdvalstore;

      //Current value
      double dcurval;

public:

      //Constructor
      ACMMDesignVarDef()
      {
            inalloc=10;
            pdvalstore=new double[inalloc];
            inchoices=0;
            pdvalstore[0]=0.0;
            pdvalstore[1]=1.0;

            dcurval=0.0;
      }
      //Destructor
      ~ACMMDesignVarDef()
      {
            delete [inalloc] pdvalstore;
      }


      //Function to setup the variable as continuous, also resets value
to minimum range
      void setcontinuous(double drmin, double drmax)
      {
            inchoices=0;
            pdvalstore[0]=drmin;
            pdvalstore[1]=drmax;
            dcurval=drmin;
      }

      //Function to setup the variable as discrete, also resets value to
1st option
      void setdiscrete(int icnchoices, const double *pdchoices)
      {
            if (icnchoices<1) return;
            if (icnchoices>inalloc)
            {
                  delete [inalloc] pdvalstore;
                  inalloc=icnchoices;
                  pdvalstore=new double[inalloc];
            }
            inchoices=icnchoices;
            for (int icount=0; icount<inchoices; icount++)
            {
                  pdvalstore[icount]=pdchoices[icount];
            }
            dcurval=pdvalstore[0];
      }

      //Function to read model contents from file stream
      void readf(ifstream *pfin)
      {
            *pfin >> inchoices;
            if (inchoices<0) inchoices=0;
            if (inchoices==0)
            {
```

```
            *pfin >> pdvalstore[0];
            *pfin >> pdvalstore[1];
      }
      else
      {
            if (inchoices>inalloc)
            {
                  delete [inalloc] pdvalstore;
                  inalloc=inchoices;
                  pdvalstore=new double[inalloc];
            }
            for (int icount=0; icount<inchoices; icount++)
            {
                  *pfin >> pdvalstore[icount];
            }
      }
      dcurval=pdvalstore[0];
}

//Function to write model contents to file stream
void writef(ofstream *pfout) const
{
      *pfout << inchoices << char(9);
      if (inchoices==0)
      {
            *pfout << pdvalstore[0] << char(9);
            *pfout << pdvalstore[1] << char(9);
      }
      else
      {
            for (int icount=0; icount<inchoices; icount++)
            {
                  *pfout << pdvalstore[icount] << char(9);
            }
      }
      *pfout << endl;
}

//Function to read value from file stream
void vreadf(ifstream *pfin)
{
      double dtmp;
      *pfin >> dtmp;
      setvalue(dtmp);
}
//Function to write value to file stream
void vwritef(ofstream *pfout) const {*pfout << dcurval <<endl;}


//Function to check if variable is discrete
```

```
bool isdiscrete() const
{
      if (inchoices==0) return false;
      else return true;
}
//Function to return current value of variable
double getcurval() const {return dcurval;}

//Function to return minimum range (if continuous)
double getrmin() const {return pdvalstore[0];}

//Function to return maximum range (if continuous)
double getrmax() const {return pdvalstore[1];}

//Function to return number of choices (if discrete)
int getnchoices() const {return inchoices;}

//Function to return discrete choices
const double *getchoices() const {return pdvalstore;}


//Function to set current value (or set to nearest allowed value)
void setvalue(double dnewval)
{
      dcurval=dnewval;
      if (inchoices==0)
      {
            if (dcurval<pdvalstore[0]) dcurval=pdvalstore[0];
            if (dcurval>pdvalstore[1]) dcurval=pdvalstore[1];
      }
      else
      {
            double dcurerr;
            int ibestchoice=0;
            double dminerr=fabs(dnewval-pdvalstore[0]);
            for (int icount=1; icount<inchoices; icount++)
            {
                  dcurerr=fabs(dnewval-pdvalstore[icount]);
                  if (dminerr>dcurerr)
                  {
                        dminerr=dcurerr;
                        ibestchoice=icount;
                  }
            }
            dcurval=pdvalstore[ibestchoice];
      }
}

//Assignment operator, copies contents of another
void operator=(const ACMMDesignVarDef &other)
```

```
      {
            delete [inalloc] pdvalstore;
            inalloc=other.inalloc;
            pdvalstore=new double[inalloc];
            inchoices=other.inchoices;
            pdvalstore[0]=other.pdvalstore[0];
            pdvalstore[1]=other.pdvalstore[1];
            for (int icount=0; icount<inchoices; icount++)
            {
                  pdvalstore[icount]=other.pdvalstore[icount];
            }
            dcurval=other.dcurval;
      }
};


//Class for Objective an Function and/or Constraint scaling & storage
class ACMMOFConScStr
{
protected:

      //Scaling weight
      double dweight;

      //Storage of current value
      double dcurval;

public:

      //Constructor
      ACMMOFConScStr() {reset();}
      //Destructor
      ~ACMMOFConScStr() {}

      //Function to reset values
      void reset()
      {
            dweight=1.0;
            dcurval=0.0;
      }

      //Function to return weight
      double getweight() const {return dweight;}
      //Function to return current stored value
      double getcurval() const {return dcurval;}

      //Function to set weight
      void setweight(double dcweight) {dweight=dcweight;}
      //Function to set current value
      void setcurval(double dcval) {dcurval=dcval;}
      //Function to read weight from file stream
      void readf(ifstream *pfin) {*pfin >> dweight;}
      //Function to write weight to file stream
      void writef(ofstream *pfout) const {*pfout << dweight << endl;}

      //Function to read value from file stream
      void vreadf(ifstream *pfin) {*pfin >> dcurval;}
      //Function to write value to file stream
      void vwritef(ofstream *pfout) const {*pfout << dcurval << endl;}


      //Assignment Operator to copy from another
      void operator=(const ACMMOFConScStr &other)
      {
            dweight=other.dweight;
            dcurval=other.dcurval;
      }
};



//Class for Design variales, Objectives & Constriants model storage &
instantitiation
class ACMMVOCModelInst
{
protected:

      //Number of design variables
      int invar;
      //Number of objectives
      int inobj;
      //Number of constraints
      int incon;

      //Storage of Design Variables
      ACMMDesignVarDef *pvars;
      //Storage of Objectives
      ACMMOFConScStr *pobjs;
      //Storae of Constraints
      ACMMOFConScStr *pcons;


      //Function to free allocated memory
      void freemem();

public:

      //Constructor
      ACMMVOCModelInst();
```

```
//Destructor
~ACMMVOCModelInst() {freemem();}


//Function to initialize allocations for variables
void initvars(int icnvars);
//Function to initialize allocations for objectives
void initobjs(int icnobjs);
//Function to initialize allocations for constraints
void initcons(int icncons);


//Function to write contents (model) to file
void writef(const char *pcfname) const;
//Function to read contents (model) from file
void readf(const char *pcfname);

//Function to write current values to file
void vwritef(const char *pcfname) const;
//Function to read current values from file
void vreadf(const char *pcfname);


//Assignment operator, copies contents of another
void operator=(const ACMMVOCModelInst &other);

//Function to return the number of variables
int getnvar() const {return invar;}
//Function to return the number of objectives
int getnobj() const {return inobj;}
//Function to return the number of constraints
int getncon() const {return incon;}


//Function to return a constant pointer to variables
const ACMMDesignVarDef *getcptrvars() const {return pvars;}
//Function to return a constant pointer to objectives
const ACMMOFConScStr *getcptrobjs() const {return pobjs;}
//Function to return a constant pointer to constraints
const ACMMOFConScStr *getcptrcons() const {return pcons;}


//Function to return a pointer to variables
ACMMDesignVarDef *getptrvars() {return pvars;}
//Function to return a pointer to objectives
ACMMOFConScStr *getptrobjs() {return pobjs;}
//Function to return a pointer to constraints
ACMMOFConScStr *getptrcons() {return pcons;}
```

```
//Utility function to grab variable values
void getvarvals(double *pdvars) const
{
      for (int icount=0; icount<invar; icount++)
      {
            pdvars[icount]=pvars[icount].getcurval();
      }
}
//Utility function to grab objective values
void getobjvals(double *pdobjs) const
{
      for (int icount=0; icount<inobj; icount++)
      {
            pdobjs[icount]=pobjs[icount].getcurval();
      }
}
//Utility function to grab constraint values
void getconvals(double *pdcons) const
{
      for (int icount=0; icount<incon; icount++)
      {
            pdcons[icount]=pcons[icount].getcurval();
      }
}


//Utility function to set variable values
void setvarvals(const double *pdvars)
{
      for (int icount=0; icount<invar; icount++)
      {
            pvars[icount].setvalue(pdvars[icount]);
      }
}
//Utility function to set objective values
void setobjvals(const double *pdobjs)
{
      for (int icount=0; icount<inobj; icount++)
      {
            pobjs[icount].setcurval(pdobjs[icount]);
      }
}
//Utility function to set constraint values
void setconvals(const double *pdcons)
{
      for (int icount=0; icount<incon; icount++)
      {
            pcons[icount].setcurval(pdcons[icount]);
      }
}
```

```
      //Utility function to compare with another in terms of design
objectives
      // returns -1 if worse than other, 1 if better, 0 if same as
      int IsBetterThan(const ACMMVOCModelInst &other) const;
};



//Class for CM error comparison elementary term
class ACMMErrCompElemTerm
{
protected:

      //Zone ID
      int izoneid;

      //Deformation Type ID
      int ideformtypeid;

      //Scaling constant
      double dscwt;

public:

      //Constructor
      ACMMErrCompElemTerm()
      {
            izoneid=0;
            ideformtypeid=0;
            dscwt=1.0;
      }
      //Destructor
      ~ACMMErrCompElemTerm() {}


      //Function to write contents to file stream
      void writef(ofstream *pfout) const
      {
            *pfout << izoneid << char(9) << ideformtypeid << char(9) <<
dscwt << endl;
      }
      //Function to read contents from file stream
      void readf(ifstream *pfin)
      {
            *pfin >> izoneid;
            *pfin >> ideformtypeid;
            *pfin >> dscwt;
      }
```

```
	//Function to set zone ID
	void setzoneid(int iczoneid) {izoneid=iczoneid;}
	//Function to return zone ID
	int getzoneid() const {return izoneid;}

	//Function to set deformation type ID
	void setdeftypeid(int icdeftype) {ideformtypeid=icdeftype;}
	//Function to return deformation type ID
	int getdeftypeid() const {return ideformtypeid;}

	//Function to set scaling weight
	void setsclwtid(double dcwt) {dscwt=dcwt;}
	//Function to return scaling weight
	double getsclwtid() const {return dscwt;}


	//Assignment operator to copy from another
	void operator=(const ACMMErrCompElemTerm &other)
	{
		izoneid=other.izoneid;
		ideformtypeid=other.ideformtypeid;
		dscwt=other.dscwt;
	}


	//Function to calculate equivalent "measure value" out of CM
history
	double calcEqCMValueT(const CrashModeBase *pcmobj, double
dtimeval) const
	{
		return dscwt*pcmobj->getcmvalue_zdt(izoneid, ideformtypeid,
dtimeval);
	}
};



//Class for CM error comparison expression
class ACMMErrCompExpr
{
protected:

	//Number of elementary terms
	int interms;

	//Storage of terms
	ACMMErrCompElemTerm *pterms;
```

```
public:

      //Constructor
      ACMMErrCompExpr()
      {
            interms=0;
            pterms=NULL;
      }
      //Destructor
      ~ACMMErrCompExpr()
      {
            if (interms>0) delete [interms] pterms;
      }

      //Function to initialize/set number of terms
      void init(int icnterms)
      {
            if (interms>0) delete [interms] pterms;
            interms=icnterms;
            if (interms>0) pterms=new ACMMErrCompElemTerm[interms];
      }

      //Function to write contents to file stream
      void writef(ofstream *pfout) const
      {
            *pfout << interms << endl;
            for (int icount=0; icount<interms; icount++)
            {
                  pterms[icount].writef(pfout);
            }
      }
      //Function to read contents from file stream
      void readf(ifstream *pfin)
      {
            int itmp;
            *pfin >> itmp;
            init(itmp);
            for (int icount=0; icount<interms; icount++)
            {
                  pterms[icount].readf(pfin);
            }
      }


      //Function to return the number of terms
      int getnterms() const {return interms;}
      //Function to return a contant pointer to terms
      const ACMMErrCompElemTerm *getcptrterms() const {return pterms;}
      //Function to return a pointer to terms
      ACMMErrCompElemTerm *getptrterms() {return pterms;}
      //Function to calculate equivalent "measure value" out of CM
history
      double calcEqCMValueT(const CrashModeBase *pcmobj, double
dtimeval) const
      {
            double dsumval=0.0;
            for (int icount=0; icount<interms; icount++)
            {
                  dsumval+=pterms[icount].calcEqCMValueT(pcmobj,
dtimeval);
            }
            return dsumval;
      }


      //Assignment operator to copy from another
      void operator=(const ACMMErrCompExpr &other)
      {
            if (interms>0) delete [interms] pterms;
            interms=other.interms;
            if (interms>0) pterms=new ACMMErrCompElemTerm[interms];
            for (int icount=0; icount<interms; icount++)
            {
                  pterms[icount]=other.pterms[icount];
            }
      }
};



//Class for CM error value comparison
class ACMMErrCompValue
{
protected:

      //Hard limit CM values calculation object
      ACMMErrCompExpr hlexpr;

      //Detailed history CM values calculation object
      ACMMErrCompExpr dhexpr;

public:

      //Constructor
      ACMMErrCompValue() {}
      //Destructor
      ~ACMMErrCompValue() {}
      //Function to return a constant pointer to HL CM values
calculation object
      const ACMMErrCompExpr *getcptr_hlobject() const {return &hlexpr;}
      //Function to return a constant pointer to DH CM values
calculation object
      const ACMMErrCompExpr *getcptr_dhobject() const {return &dhexpr;}
      //Function to return a pointer to HL CM values calculation object
      ACMMErrCompExpr *getptr_hlobject() {return &hlexpr;}
      //Function to return a pointer to DH CM values calculation object
      ACMMErrCompExpr *getptr_dhobject() {return &dhexpr;}


      //Function to write contents to file stream
      void writef(ofstream *pfout) const
      {
            hlexpr.writef(pfout);
            dhexpr.writef(pfout);
      }
      //Function to read contents from file stream
      void readf(ifstream *pfin)
      {
            hlexpr.readf(pfin);
            dhexpr.readf(pfin);
      }


      //Function to calculate equivalent error measure
      double calcErrMeasure(const CrashModeDetTHist *pdhcm, const
CrashModeHardLim *phlcm) const
      {
            int inevals=pdhcm->get_nstoretimeinstants();
            const double *pdtimeinsts=pdhcm->get_storetimeinstants();
            double dsumerr=0.0;
            for (int icount=0; icount<inevals; icount++)
            {
                  dsumerr+=dhexpr.calcEqCMValueT(pdhcm,
pdtimeinsts[icount])
                              -hlexpr.calcEqCMValueT(phlcm,
pdtimeinsts[icount]);
            }
            return dsumerr/double(inevals);
      }


      //Assignment operator, to copy from another
      void operator=(const ACMMErrCompValue &other)
      {
            hlexpr=other.hlexpr;
            dhexpr=other.dhexpr;
      }
};



//Class for CM error values comparison
class ACMMErrCompValues
{
protected:

      //Number of error measures
      int inerrs;

      //Storage of error measures calculators
      ACMMErrCompValue *perrcalcstore;

      //Weighting for error measures (required only for comparing
"goodness of CM match")
      double *pderrwts;


      //Function to free allocated memory
      void freemem();

public:


      //Constructor
      ACMMErrCompValues();
      //Destructor
      ~ACMMErrCompValues() {freemem();}


      //Initialization function (sets number of error measures)
      // also initializes weights to unity
      void init(int icnerrs);

      //Function to set weights
      void setweights(const double *pdwtvals);


      //Function to write contents to file
      void writef(const char *pcfname) const;
      //Function to read contents from file
      void readf(const char *pcfname);


      //Function to return the number of error measures
      int getnerrs() const {return inerrs;}
```

```
	//Function to return a constant pointer to error weights
	const double *geterrwtss() const {return pderrwts;}

	//Function to return a constant pointer to error calculation
objects
	const ACMMErrCompValue *getcptrerrobjs() const {return
perrcalcstore;}
	//Function to return a pointer to error calculation objects
	ACMMErrCompValue *getptrerrobjs() {return perrcalcstore;}


	//Function to calculate error measures
	void calcErrs(double *pderrvals, const CrashModeDetTHist *pdhcm,
		const CrashModeHardLim *phlcm) const
	{
		for (int icount=0; icount<inerrs; icount++)
		{

	pderrvals[icount]=perrcalcstore[icount].calcErrMeasure(pdhcm,
phlcm);
		}
	}

	//Function to calculate an "Equivalent" overall error measure
	double calcErr(const CrashModeDetTHist *pdhcm, const
CrashModeHardLim *phlcm) const
	{
		double dsumwt=0.0;
		for (int icount=0; icount<inerrs; icount++)
		{

	dsumwt+=pderrwts[icount]*fabs(perrcalcstore[icount].calcErrMeasure
(pdhcm, phlcm));
		}
		return dsumwt;
	}

	//Function to write plots of CM error differences
	void plotfErrs(const char *pcfname, const CrashModeDetTHist
*pdhcm,
		const CrashModeHardLim *phlcm) const;
};



//Class for proposed CM matching algorith (main core iterator)
class ACMMMainIterator
{
public:
      //Constructor
      ACMMMainIterator() {}
      //Destructor
      ~ACMMMainIterator() {}

      //Function to run a step of the algorithm via file input
      int runsetpf(const char *pcstepmasterfname);
};


#endif      //H_CPP_AutoCMMatcher
```

BIBLIOGRAPHY

BIBLIOGRAPHY

Abramowicz, W., 2003, “Thin-Walled Structures as Impact Energy Absorbers,” Thin Walled Structures, vol. 41, pp. 91-107.

Abramowicz, W., 2004, “An alternative formulation of the FE method for arbitrary discrete/continuous models,” Int. J. of Impact Engineering, vol. 30, pp. 1081-1098.

Andersson, J. and Redhe, M., 2003, “Response Surface Methods for Pareto Optimization in Crashworthiness Design,” Proceedings of the ASME 2003 Design Engineering and Technical Conference, September 2-6, Chicago, IL, DETC 03 / DAC 48752.

Bennett, J. A., Lust, R. V. and Wang, J.T., 1991, “Optimal Design Strategies in Crashworthiness and Occupant Protection,” ASME Winter Annual Meeting, Atlanta, GA, 126, pp. 51-66.

Besset, D., 2001, Object-Oriented Implementation of Numerical Methods, Morgan Kaufmann, San Francisco, CA.

Box, G., Hunter, W. and Hunter, J., 1978, Statistics for Experimenters, John Wiley & Sons, New York.

Chellappa, S. and Diaz, A., 2002, “A Multi-Resolution Reduction Scheme for Structural Design,” Proc. NSF 2002 Conference, January 2002, pp. 98-107.

Chen, S., 2001, “An Approach for Impact Structure Optimization using the Robust Genetic Algorithm,” Finite Elements in Analysis and Design, Vol. 37, pp. 431-446.

Chen, W., Allen, J., Mavris, D. and Mistree, F., 1996, “A Concept Exploration Method for Determining Robust Top-Level Specifications,” Engineering Optimization, Vol. 26, pp. 137-158.

Chen, W., Allen, J., Tsui, K. and Mistree, F., 1996, “A Procedure for Robust Design: Minimizing Variations Caused by Noise Factors and Control Factors,” ASME Journal of Mechanical Design, Vol. 118, pp. 478-485.

Chen, W., Sahai, A., Messac, A., and Sundararaj, G., 2000, “Exploration of the Effectiveness of Physical Programming in Robust Design,” ASME Journal of Mechanical Design, Vol. 122, pp. 155-163.

Chen, W., Wiecek, M., and Zhang, J., 1999, "Quality Utility – A Compromise Programming Approach to Robust Design," ASME Journal of Mechanical Design, Vol. 121, pp. 179-187.

Chen., W., Garimella, R. and Michelena, N., 1999, "Robust Design for Improved Vehicle Handling under a Range of Maneuver Conditions," Proceedings of the ASME 1999 Design Engineering and Technical Conferences, September 12-15, Las Vegas, Nevada, DETC 99 / DAC 8580.

Coello, C., Van Veldhuizen, D. and Lamont, G., 2002, Evolutionary Algorithms for Solving Multi-Objective Problems, Kluwer Academic/Plenum Publishers, New York.

Deb, K., Argawal, S., Pratab, A. and Meyarivan, T., 2000, "A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II," Proceedings of the Parallel Problem Solving from Nature VI Conference, Paris, France, pp. 849-858.

Dyn, N., Levin, D. and Rippa, S., 1986, "Numerical Procedures for Surface Fitting of Scattered Data by Radial Basis Functions," SIAM Journal of Scientific and Statistical Computing, Vol. 7, No. 2, pp. 639-659.

ESI, 2003, PAM-Crash Software Manuals, ESI Group, 6, rue Hamelin, BP 2008-16, 75761 Paris Cedex 16, France.

Gea, H. C. and Luo, J., 2001, "Design for Energy Absorption: A Topology Optimization Approach," Proc. ASME 2001 Design Engineering and Technical Conferences, September 9-12, Pittsburgh, PA, DETC 2001 / DAC 21060.

Goldberg, D., 1989, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Inc., New York.

Gu, L., Tyan, T. and Yang, R. J., 2004, "Vehicle Structure Optimization for Crash Pulse," Proceedings of the ASME 2004 Design Engineering and Technical Conference, September 28 – October 2, Salt Lake City, Utah, DETC 2004-57479.

Guenter, F., Muellerschon, H. and Roux, W., 2004, "Robustness Study of an LS-DYNA Occupant Simulation Model at DaimlerChrysler Commercial Vehicles using LS-OPT," 8th International LS-DYNA users Conference, pp. 1031-1042.

Hajela, P. and Lee, E., 1997, "Topological Optimization of Robocraft Subfloor Structures for Crashworthiness Condsiderations, " Computers and Structures, Vol. Vol. 64, No. 1-4, pp. 65-76.

Hamza, K. and Saitou, K., 2003, “Design Optimization of Vehicle Structures for Crashworthiness using Equivalent Mechanism Approximations,” Proc. ASME 2003 Design Engineering and Technical Conference, September 2-6, Chicago, IL, DETC/DAC 48751.

Hamza, K. and Saitou, K., 2004, “Crash Mode Analysis of Vehicle Structures based on Equivalent Mechanism Approximations,” Proc. 5th International Symposium on Tools and Methods of Competitive Engineering, Lausanne, Switzerland, April 13 - 17, pp. 277-287.

Hamza, K. and Saitou, K., 2004, “Crashworthiness Design Using Meta-Models for Approximating of Box-Section Members,” Proc. 8th Cairo University International Conference on Mechanical Design and Production, Cairo, Egypt, January 4-6, 1, pp. 591-602.

Hamza, K. and Saitou, K., 2004, “Design for Crashworthiness of Vehicle Structures via Equivalent Mechanism Approximations and Crash Mode Matching,” Proc. ASME 2004 Mechanical Engineering Congress, November 13-20, Anaheim, CA, IMECE2004-62226.

Hamza, K. and Saitou, K., 2004, “Design Optimization of Vehicle Structures for Crashworthiness via Equivalent Mechanism Approximations,” Proc. SAE World Congress, Detroit, MI, Paper no. 04B-126.

Hamza, K. and Saitou, K., 2005, “Design for Structural Crashworthiness using Equivalent Mechanism Approximations,” Journal of Mechanical Design, vol. 127, pp. 485-492.

Han, J. and Yamada, K., 2000, “Maximization of the Crushing Energy Absorption of the S-Shaped Thin-Walled Square Tube,” Proc. 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, September 6-8, Long Beach, CA, AIAA Paper Number: AIAA-2000-4750.

Hansen, L. and Salamon, P., 1990, “Neural Network Ensembles,” IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 12, No. 10, pp. 993-1001.

Haykin, S., 1998, Neural Networks: A Comprehensive Foundation, 2nd Edition, Prentice Hall, Englewood Cliffs, New Jersey.

Hopgood, A. A., 2001, Intelligent Systems for Engineers and Scientists, 2nd Edition, CRC Press, New York, USA.

Ignatovich, C. L. and Diaz, A., 2002, "Physical Surrogates in Design Optimization for Enhanced Crashworthiness," Proc. 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, September 4-6, Atlanta, Georgia, AIAA Paper Number: AIAA-2002-5537.

Jin, R., Du, X., and Chen, W., 2001, "The Use of Metamodeling Techniques for Optimization Uder Uncertainty," Proceedings of the ASME 2001 Design Engineering and Technical Conferences, September 9-12, Pittsburgh, PA, DETC 2001 / DAC 21039.

Koanti, R. P. and Caliskan, A. G., 2001, "Stochastic Applications in Crashworthiness," Proc. 2001 ASME International Mechanical Engineering Congress, November 11-16, New York, NY, IMECE 2001/AMD 25433.

Krige, D., 1951, "A Statistical Approach to some Mine Evaluations and Allied Problems at the Witwatersrand," M.Sc. Thesis, University of Witwatersrand, Germany.

Kurtaran, H., Omar, T. and Eskandarian, A., 2001, "Crashworthiness Design Optimization of Energy-Absorbing Rails for the Automotive Industry," Proc. ASME 2001 International Mechanical Engineering Congress and Exposition, November 11-16, New York, NY, IMECE2001-AMD25452.

LSTC, 2001, LS-DYNA Software Manuals, Livermore Software Technology Corporation, Livermore, CA, USA.

Luo, J., Gea, H. C. and Yang, R. J., 2000, "Topology Optimization for Crush Design," Proc. 8th AIAA /USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, September 6-8, Long Beach, CA, AIAA Paper Number: AIAA-2000-4770.

Martin, J., and Simpson, T., 2004, "On the use of Kriging Models to Approximate Deterministic Computer Models," Proceedings of the ASME 2004 Design Engineering and Technical Conference, September 28 – October 2, Salt Lake City, Utah, DETC 2004-57300.

Mase, T., Wang, J. T., Mayer, R., Bonello, K. and Pachon, L., 1999, "A Virtual Bumper Test Laboratory for FMVR 581," Proc. ASME 1999 Design Engineering and Technical Conferences, September 12-15, Las Vegas, Nevada, DETC 99 / DAC 8572.

MathWorks, 2001, MatLab 6 Documentation, MathWorks Inc., Natick, MA, USA.

Mayer, R. R., 2001, "Application of Topological Optimization Techniques to Automotive Structural Design," Proc. ASME 2001 International Mechanical Engineering Congress and Exposition, November 11-16, New York, NY, IMECE 2001 / AMD 25458.

Mayer, R. R., Kikuchi, N. and Scott, R. A., 1996, "Application of Topological Optimization Techniques to Structural Crashworthiness," International Journal for Numerical Methods in Engineering, vol. 39, pp. 1383-1403.

Mayer, R. R., Maurer, D. and Bottcher, C., 2000, "Application of Topological Optimization Program to the Danner Test Simulation," Proc. ASME 2000 Design Engineering and Technical Conferences, September 10-13, Baltimore, Maryland, DETC 2000 / DAC 14292.

McAllister, C. and Simpson, T., 2003, "Multidisciplinary Robust Design Optimization of an Internal Combustion Engine," ASME Journal of Mechanical Design, Vol. 125, pp. 124-130.

Michalewicz, Z., 1996, Genetic Algorithms + Data Structures = Evolution Programs. 3rd edition, Springer-Verlag, Berlin-Heidelberg

Michalewiz, Z. and Fogel, D. B., 2000, How to Solve it: Modern Heuristics, Springer-Verlag Berlin Heidelberg, New York.

Myers, R. and Montgomery, D., 1995, Response Surface Methodology: Process and Product Optimization using Designed Experiments, Wiley & Sons, New York.

NHTSA, National Highway Traffic Safety Administration, http://www.nhtsa.dot.gov/

Owen, A., 1992, "Orthogonal Arrays for Computer Experiments, Integration and Visualization," Statistica Sinica, Vol. 2, pp. 439-452.

Parkinson, A., Sorensen, C. and Pourhassan, N., 1993, "A General Approach for Robust Optimal Design," ASME Journal of Mechanical Design, Vol. 115, pp. 74-80.

Phadke, M., 1993, Quality Engineering using Robust Design, Prentice Hall PTR, Englewood Cliffs, New Jersey.

Redhe, M. and Nilsson, L., 2002, "Using Space Mapping and Surrogate Models to Optimize Vehicle Crashworthiness Design," Proc. 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, September 4-6, Atlanta, Georgia, AIAA Paper Number: AIAA-2002-5536.

Rumelhart, D., Hinton, G. and Williams, R., 1986, "Learning Internal Representations by Error Propagation," Parallel Distributed Processing: Explorations in the Microstructures of Cognition, 1: Foundations, MIT Press, Cambridge, MA, pp. 318-362.

Resnick, S., 1992, Adventures in Stochastic Processes, Birkhaeuser Boston c/o Springer Science, New York, NY.

Sacks, J., Welch, W., Mitchell, T. and Wynn, H., 1989, "Design and Analysis of Computer Experiments," Statistical Science, Vol. 4, No. 4, pp. 409-435.

Saitou, K., Izui, K., Nishiwaki, S. and Papalambros, P., 2005, "A Survey of Structural Optimization in Mechanical Product Development," Journal of Computing and Information Science in Engineering, Vol. 5, pp. 214-226.

Sasena, M., Papalambros, P and Goovaerts, 2002, "Exploration of Metamodeling Sampling Criteria for Constrained Global Optimization," Engineering Optimization, Vol 34, No. 3, pp. 263-278.

Shi, Q. Hagiwara, I. and Takashima, F., 1999, "The Most Probable Optimal Design Method for Global Optimization," Proc. ASME 1999 Design Engineering and Technical Conferences, September 12-15, Las Vegas, Nevada, DETC 99 / DAC 8635.

Siah, S., Sasena, M., Volakis, J., Papalambros, P. and Wise, R., 2004, "Fast Parameter Optimization of Large-Scale Electromagnetic Objects using DIRECT with Kriging Metamodeling," IEEE Transactions on Microwave Theory and Techniques, Vol. 52, No. 1, pp. 276-284.

Simpson, T., Booker, A., Ghosh, D., Giunta, A., Koch, P. and Yang, R., 2004, "Approximation Methods in Multidisciplinary Analysis and Optimization: A Panel Discussion," Structural and Multidisciplinary Optimization, vol. 27, pp. 302-313.

Song, J. O., 1986, "An Optimization Method for Crashworthiness Design," SAE transactions, Paper number: 860804, pp. 39-46.

Soto, C. A. and Diaz, A. R., 1999, "Basic Models for Topology Design Optimization in Crashworthiness Problems," Proc. ASME 1999 Design Engineering and Technical Conferences, September 12-15, Las Vegas, Nevada, DETC 99 / DAC 8591.

Soto, C. A., 2001, "Optimal Structural Topology Design for Energy Absorption: A Heurtistic Approach," Proc. ASME 2001 Design Engineering and Technical Conferences, September 9-12, Pittsburgh, PA, DETC 2001 / DAC 21126.

Soto, C. A., 2001, "Structural Topology for Crashworthiness Design by Matching Plastic Strain and Stress Levels," Proc. ASME 2001 International Mechanical Engineering Congress and Exposition, November 11-16, New York, NY, IMECE 2001 / AMD 25455.

Soto., C., 2003, "Structural Topology Design Optimization for Controlled Crash Behavior," Proc. ASME 2003 Design Engineering and Technical Conference, September 2-6, Chicago, IL, DETC/DAC-48733.

Srivastava, A., Hacker, K., Lewis, K. and Simpson T., 2004, "A Method for Using Legacy Data for Metamodel-Based Design of Large-Scale Systems," Structural and Multidisciplinary Optimization, Vol. 28, pp. 146-155.

Stein, M., 1987, "Large Sample Properties of Simulations using Latin Hypercube Sampling," Technometrics, Vol. 29, pp. 143-151.

Sundaresan, S., Ishii, K. and Houser, D., 1995, "A Robust Optimization Procedure with Variations on Design Variables and Constraints," Engineering Optimization, Vol. 24, pp. 101-117.

Taguchi, G., 1993, Taguchi on Robust Technology Development: Bringing Quality Engineering Upstream, ASME Press, New York.

Takada, K. and Abramowicz, W., 2004, "Fast Crash Analysis of 3D Beam Structures Based on Object Oriented Formulation," Proc. 2004 SAE World Congress, Detroit, Michigan, Paper no. 04B-119.

Tang, X. and Cheng, J., 1997, "Crash/Crush Analysis of Vehicle Structures Utilizing Thin-Walled Beam Elements," Proc. ASME Design Engineering and Technical Conferences, September 14-17, Sacramento, CA, DETC97/CIE-4454.

Wanas, N. and Kamel, M., 2002, "Weighted Combination of Neural Network Ensembles," Proceedings of the IEEE International Joint Conference on Neural Networks, Vol. 2, pp. 1748-1752.

Werfel, J., Mitchell, M. and Crutchfield, J., 2000, "Resource Sharing and Coevolution in Evolving Cellular Automata," IEEE Transactions on Evolutionary Computation, Vol. 4, No. 4, pp. 388-393.

Wu, Y. and Arribas, J., 2003, "Fusing Output Information in Neural Networks: Ensemble Performs Better," Proceeding of the 25th International Conference of the IEEE EMBS, September 17-21, Cancun, Mexico, pp. 2265-2268.

Yang, R. J., Gu, L., Liaw, L., Gearhart, C., Tho, C. H., Liu, X. and Wang, B. P., 2000, "Approximations for Safety Optimization of Large Systems," Proc. ASME 2000 Design Engineering and Technical Conferences, September 10-13, Baltimore, Maryland, DETC 2000 / DAC 14245.

Yang, R. J., Gu, L., Tho, C. H. and Sobieski, J., 2001, "Multidisciplinary Optimization of a Full Vehicle with High Performance Computing," Proceedings of the American Institute of Aeronautics and Astronautics 2001 Conference, pp. 688-698, AIAA Paper Number: AIAA-2001-1273.

Yang, R. J., Tho, C. H., Wu, C. C., Johnson, D. and Cheng, J., 1999, "A Numerical Study of Crash Optimization," Proc. ASME 1999 Design Engineering and Technical Conferences, September 12-15, Las Vegas, Nevada, DETC 99 / DAC 8590.

Yang, R. J., Wang, N., Tho, C. H., Bobineau, J. P. and Wang, B. P., 2001, "Metamodeling Development for Vehicle Frontal Impact Simulation," Proc. ASME 2001 Design Engineering and Technical Conferences, September 9-12, Pittsburgh, PA, DETC 2001 / DAC 21012.

Zhou, Z., Wu, J. and Tang, W., 2002, "Ensembling Neural Networks: Many Could be Better than All," Artificial Intelligence, Vol. 137, pp. 239-263.