MULTI-STATION INJECTION MOLDING --
DESIGN AND ANALYSIS
USING EVOLUTIONARY ALGORITHMS

by

Anthony Wolf

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science,
Graduate Department of Mechanical and Industrial Engineering,
University of Toronto

© Copyright by Anthony Wolf 2001
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-63127-3
MULTI-STATION INJECTION MOLDING –
DESIGN AND ANALYSIS
USING EVOLUTIONARY ALGORITHMS

Master of Applied Science
2001
Anthony Russell Wolf

Department of Mechanical and Industrial Engineering
University of Toronto

A tool to facilitate the feasibility study of a newly proposed multi-station injection molding system is developed. The conceptual design and proposed embodiment of the new system are geared toward the development of a system flexible enough to handle multiple part types and production volumes. A comprehensive design model is used to structure the problem. Two customized Evolutionary Programming algorithms are used to optimize the new and traditional system configurations. The systems are compared in their ability to satisfy a range of production requirements captured by the 54 trials of a full factorial design of experiments. Results indicate that the multi-station system outperformed the traditional system in at least 57% of the trials confirming the viability of the new system. Furthermore, an analysis of variance indicates that the factors part thickness, projected area, production volume and part variety significantly affect the cost differential between the two systems.
**Acknowledgements**

I would like to thank Prof. Li Shu and Prof. Ron Venter for their help and guidance during this thesis. Prof. Shu, thanks for helping me stay on track throughout the two years and for the healthy doses of patience when needed.

There are a number of people to whom I owe a debt of gratitude. To Daniella for kindly listening to my every thought, and for your boundless encouragement through every twist and turn. To my parents and sister for their ongoing support, which started when I was born and shows no end in sight. Lastly, to Ms. Margie Wolfson, Mrs. Judy Joffe, Dr. Eli Honig, Prof. Eric Pickett and all those whose tutelage has influenced me, your insightful and encouraging words have always lifted me higher and guided me forward.

As with any laborious pursuit, looking back one wonders where the effort and time spent has gone. This document helps capture a small piece of the dividends. The rest, the knowledge and skills, I will carry with me wherever I go.

---

**Firefly**

It lifts, and glows, and streaks the night. Turn to look, it's out of sight.

- Shilo
Table of Contents

Abstract ......................................................................................................................... ii
Acknowledgements ........................................................................................................ iii
Table of Contents ......................................................................................................... iv
List of Figures ................................................................................................................ xi
List of Tables .................................................................................................................. xiii
List of Equations .......................................................................................................... xiv
List of Acronyms .......................................................................................................... xvi

1. Introduction .............................................................................................................. 1
   1.1. Problem Statement and Motivation .................................................................. 1

1.2. Related Research ................................................................................................. 2
   1.2.1. Optimal Setting of Process Parameters...................................................... 2
   1.2.2. Machine Modification................................................................................... 4

1.3. Background: Lean Production ............................................................................. 7
   1.3.1. Takt Time..................................................................................................... 7

   1.3.2. Level Production........................................................................................ 9
   1.3.3. Single-Piece Flow ................................................................................... 10
   1.3.4. Right Sized Machines ........................................................................... 12

1.4. Research Objectives and Thesis Structure......................................................... 12

2. Design of a Lean IM System ..................................................................................... 14
   2.1. Scope of Design .............................................................................................. 14

      2.1.1. Traditional-System Design .................................................................... 14

      2.1.2. New-System Design ............................................................................. 16
7. Results .................................................................................................................. 71

7.1. Introduction ........................................................................................................... 71

7.2. Design of Experiments ......................................................................................... 71

7.2.1. Scenario 1 - Traditional vs. New system (best-case) ....................................... 72

7.2.2. Scenario 2 - Traditional vs. New system (worst-case) .................................... 73

7.2.3. Scenario 3 - Traditional vs. New system (nominal case) ............................... 73

7.2.4. Selection of Factors and Levels ....................................................................... 74

7.2.4.1. Factor A - Wall Thickness ........................................................................... 75

7.2.4.2. Factor B - Projected Area .......................................................................... 75

7.2.4.3. Factor C - Production Volume .................................................................... 76

7.2.4.4. Factor D - Part Variety ................................................................................ 76

7.2.5. Part Shape ........................................................................................................ 78

7.2.6. Data Generation ............................................................................................... 78

7.2.6.1. Generating the data ..................................................................................... 79

7.2.6.2. Creating the minimum data sets ................................................................. 79

7.2.6.3. Calculating the cost differentials ................................................................. 80

7.2.6.4. A note on single-replicate-data sets ............................................................ 80

7.3. Data ...................................................................................................................... 81

7.4. Analysis & Results ............................................................................................... 83

7.4.1. General Observations ....................................................................................... 83

7.4.1.1. Category 1 observations - Traditional system outperformed the new system ....................................................................................................................... 84
### 7.4.1.2. Category 2 observations - No feasible configuration of the traditional system

- 84

### 7.4.1.3. Category 3 observations - New system outperformed the traditional system

- 85

### 7.4.1.4. Effect of different levels on the data

- 85

### 7.4.2. Analysis of Variance

- 87

#### 7.4.2.1. Introduction

- 87

#### 7.4.2.2. ANOVA table

- 89

#### 7.4.2.3. Single Replicate Experiments

- 91

#### 7.4.2.4. Validity Checks

- 92

### 7.5. Discussion

- 94

#### 7.5.1. Significant Treatments

- 95

#### 7.5.2. Limitations of the ANOVA Results

- 96

### 8. Summary and Conclusions

- 98

#### 8.1. Introduction

- 98

#### 8.2. Research Review

- 98

#### 8.3. Future Work

- 100

##### 8.3.1. Improving Cost Estimates

- 100

##### 8.3.2. Expanding the Use of the Design Models

- 100

##### 8.3.3. Mapping the Solution Space

- 101

##### 8.3.4. Development of a Physical Prototype

- 101

#### 8.4. Additional Considerations

- 102

#### 8.5. Final Conclusions

- 102
Appendices

Appendix I – Design Parameters

Appendix II - Parameter Range

Appendix III - Raw Data – (minimum value set)

Appendix IV – ANOVA Tables

Appendix V – Traditional-System Evolutionary Programming Algorithm

Appendix VI – New-System Evolutionary Programming Algorithm

References
List of Figures

Figure 1 - Build-to-Schedule calculation example .......................................................... 8
Figure 2 – (A) Batch Production versus (B) Level Production ........................................ 9
Figure 3 – (A) Batch Production vs. (B) Single-Piece Flow through multiple operations 10
Figure 4 – (A) Traditional injection molding using serial production equated to (B) Batch Production through multiple operations .......................................................... 11
Figure 5 – Functional Decomposition of the injection-molding process ......................... 14
Figure 6 – Design of a traditional injection-molding system .............................................. 15
Figure 7 – Design of the newly proposed injection-molding system ............................... 17
Figure 8 – Proposed embodiment of the new injection-molding system ......................... 20
Figure 9 – Schematic of the Design Model ..................................................................... 21
Figure 10 – Design model flow chart for the multi-station and traditional injection- molding machines ........................................................................................................... 23
Figure 11 – Optimization Formulation ............................................................................. 31
Figure 12 – Efficiency of various type of search and optimization approaches with respect to problem types (adapted from Goldberg, 1989) .............................................................. 33
Figure 13 – Key steps in an Evolutionary Algorithm ......................................................... 34
Figure 14 – National average injection-molding machine rates ....................................... 42
Figure 15 – 3D array allele genome ................................................................................. 47
Figure 16 – 3D array - Transformation 1 ......................................................................... 48
Figure 17 – 3D array – Transformation 2 ......................................................................... 49
Figure 18 – 3D array – Transformation 3 ......................................................................... 51
Figure 19 – Cavity-run combinations for each allele set value ........................................ 53
Figure 20 – First part of the composite genome ................................................. 60
Figure 21 – Second part of the composite genome ........................................... 61
Figure 22 – Complete composite genome ....................................................... 61
Figure 23 – Plot of estimated machine cost vs. tonnage .................................. 65
Figure 24 – Split Operator .............................................................................. 69
Figure 25 – Join Operator .............................................................................. 69
Figure 26 – Increase Operator ....................................................................... 70
Figure 27 – Decrease Operator ....................................................................... 70
Figure 28 - Parts shape .................................................................................. 78
Figure 29 - Data generation ........................................................................... 79
Figure 30 – Categorization of scenario data with regard to system performance .... 83
Figure 31 – Number of category 2 trials for each factor-level setting ............... 86
Figure 32 – Normal probability plot of studentized residuals for scenarios 1,2,3...... 93
Figure 33 – Studentized Residuals versus predicted values for scenario 1 cost data. ..... 94
List of Tables

Table 1 – An illustration of SMED principles applied to the fastening of a mold plate .... 5
Table 2 – Example set of elements in the design parameter vector ........................................ 25
Table 3 – Traditional system design variable vector ............................................................... 26
Table 4 - New system design variable vector ......................................................................... 26
Table 5 - Traditional system variable vector range ............................................................... 28
Table 6 - New system variable vector range .......................................................................... 28
Table 7 – Number of parts per cycle based on frequency and customer demand ........... 30
Table 8 – Cavity-run combinations that can be used to build exactly 100 parts ........... 50
Table 9 – Station and buffer identification letters ................................................................. 60
Table 10 – Estimated machine cost vs. tonnage .................................................................. 64
Table 11 – Cost settings for Scenario 1 .............................................................................. 72
Table 12 – Cost settings for Scenario 2 .............................................................................. 73
Table 13 – Cost settings for Scenario 2 .............................................................................. 74
Table 14 - Four factors and their respective levels ............................................................... 74
Table 15 – Table showing the factor-level settings of the 54 trials ....................................... 76
Table 16 – Cost differentials for the three scenarios............................................................ 81
Table 17 – Sample ANOVA table ....................................................................................... 90
## List of Equations

1. **Equation 1** - Build-to-Schedule definition
2. **Equation 2** - Achieving 100% BTS with varying frequencies
3. **Equation 3** - Multi-cavity mold cost calculation
4. **Equation 4** - Cycle time based processing cost calculation
5. **Equation 5** - Processing cost calculation for the traditional and new systems
6. **Equation 6** - Capital recovery factor
7. **Equation 7** - Mapping of allele set values into cavity -run combinations
8. **Equation 8** - Traditional system - total mold cost
9. **Equation 9** - Traditional Machine - total annual processing cost
10. **Equation 10** - Traditional system - Reject condition 1
11. **Equation 11** - Traditional system - Reject condition 2
12. **Equation 12** - Traditional system - Reject condition 3
13. **Equation 13** - Cost calculation for duplicate molds
14. **Equation 14** - Overall mold cost calculation
15. **Equation 15** - Station/Buffer cost calculation
16. **Equation 16** - New system - Reject condition 1
17. **Equation 17** - New system - Reject condition 2
18. **Equation 18** - New system - Reject condition 3
19. **Equation 19** - Single and separate mean models of the data
20. **Equation 20** - Definition of the treatment mean
21. **Equation 21** - Four factor response model
### List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANOVA</td>
<td>analysis of variance</td>
</tr>
<tr>
<td>BTS</td>
<td>build-to-schedule</td>
</tr>
<tr>
<td>DOE</td>
<td>design of experiments</td>
</tr>
<tr>
<td>EA</td>
<td>evolutionary algorithm</td>
</tr>
<tr>
<td>EP</td>
<td>evolutionary programming</td>
</tr>
<tr>
<td>FMS</td>
<td>flexible manufacturing system</td>
</tr>
<tr>
<td>GA</td>
<td>genetic algorithm</td>
</tr>
<tr>
<td>MCO</td>
<td>Moldability and Cost Optimization</td>
</tr>
<tr>
<td>OEM</td>
<td>original equipment manufacturer</td>
</tr>
<tr>
<td>SMED</td>
<td>single minute exchange die</td>
</tr>
</tbody>
</table>
1. Introduction

1.1. Problem Statement and Motivation

The manufacturing sector has always been under pressure to produce parts quicker and at a lower cost in order to remain financially competitive. Led by automotive Original Equipment Manufacturers (OEM), these demands have been extended in recent years to include the ability to handle increased product variety and smaller batch sizes. Subsequently, injection molders are required to handle frequent product changes without incurring large additional costs.

To address these new demands, Just-In-Time and Flexible production systems have been cited as methods by which custom injection molders can remain competitive (Lankton, 1985, Packman, I.D., 1986). More recently, industry reports indicate that the production demands are still beyond the capabilities of many molders. Here, the research to help alleviate the problem has focused on the production facility (Grande, 1995, Offergeld, 1998).

The emphasis on reducing production time and increasing flexibility has placed increasing demands on the system. This has led to the need for faster, larger, and more expensive machines and tight tolerances for the operating parameters, which are difficult to control. The goal, therefore, is to return to the conceptual design stage of the injection-molding machine, to identify system requirements based on current manufacturing demands and to develop a new system to better address those requirements.

This research describes the application of 'Lean Production' principles to the injection-molding process at the machine level. A new multi-station system is proposed. This system is then compared with the traditional system using newly developed evaluation
tools. The comparison assesses the ability of each system in handling multiple parts and smaller batch sizes. To begin, however, a broad spectrum of related research is reviewed.

1.2. Related Research

A number of modifications to the injection-molding process have been suggested to improve the performance and flexibility of current systems. These modifications can be divided into two main groups. The first deals with the optimal setting of the process variables of the machine and focuses on improving quality and throughput. The second deals with machine modifications and focuses on improving flexibility.

1.2.1. Optimal Setting of Process Parameters

A number of research efforts based on various approaches have been performed in the domain of variable setting for injection molding. Bernhardt and Kassa (1995) describe the use of an expert system to benchmark an injection-molding process. The system compares the molding process to a best-in-class process, which has been optimized to produce quality parts at the maximum rate. This expert system, called Moldability and Cost Optimization (MCO), determines the standard cycle for a job using data from experiments that have been conducted around the world over a 30-year period. The system calculates the shortest possible mean-machine process based on best-in-class process settings for the particular mold/machine system.

Other expert systems have been developed to recommend qualitative changes of molding parameters in response to mold defects. Various systems exist in the arena of defect correction employing different techniques. Jan et al. (1992, 1993) makes use of decision indices that show the likelihood that a particular variable is responsible for defects. Other
expert systems rely on prepositional and fuzzy logic. These systems can recommend quantitative changes to molding parameters (Tan K. H. & Yuen, M. M. F., 1995). Expert systems are suited to injection molding due to the large amount of on-the-job experience and knowledge associated with the process. Despite this, however, the difficulty of acquiring and translating the injection molders' experiential knowledge into a usable format for computerized systems still impedes expert system usage.

Other approaches aimed at finding the optimal process variable settings for injection molding include mathematical modeling, use of process windows, and Design of Experiments (DOE). Due to the inherent complexity of the injection-molding process, the development of mathematical models often rely heavily on approximations and, therefore, lead to a narrow scope of application, e.g., filling a cylindrical tube shape.

In contrast to mathematical modeling, which seeks to understand the mechanics of the process in order to determine the optimal process settings, process windows and Design of Experiments use experimental or simulation-based data. Because of the large amount of data to be collected, both approaches are time consuming and costly and require an intricate knowledge of the injection-molding systems. Still other approaches, including Case-Based Reasoning and the use of Genetic Algorithms (Mok et al., 1999), have been proposed.

The numerous approaches discussed above are needed because typically over a dozen interdependent process variables are involved, which makes for a highly complex system. Traditionally variables are set by trial-and-error, which is not conducive to a process requiring a high degree of flexibility.
1.2.2. Machine Modification

Concurrent with research focused on process-parameter optimization is the largely industry-based effort to improve system flexibility through machine modifications. Changes to the injection-molding apparatus typically include the addition of a quick-mold-change system. These systems help to reduce the changeover time and thereby improve flexibility (Rozema and Travaglini, 1995). The earliest form of quick-mold-change system, the Single-Minute Exchange Die (SMED) system developed by Toyota Motor Corporation, involved a reduction in setup times through an improved changeover process (Szatkowski and Reasor, 1991).

The SMED system focuses on removing or modifying the manual die-changing process. This is achieved by breaking down the changeover process into a series of small operations. Each of these operations is then classified as either an internal or an external operation. External operations are those that can be performed off-line prior to the machine shutdown for die changeover. Internal operations can only be performed during the changeover process. In addition to the fact that many of the operations were external, a large portion of the remaining internal factors could be converted to external operations. For example, Table 1 shows the series of operations typically involved in fastening a mold plate to the machine.

The two cases illustrate the use of SMED principles. Case 1 shows the initial series of operations, all of which are internal. In Case 2 the 'add nut' operation has been converted into an external operation by making use of a slot in the plate instead of a hole.
More recently, a number of other approaches have been developed ranging from fully automated quick-mold-change technology to relatively simple setups utilizing hydraulic mold clamps and top entry by crane. Newer still, are the magnetic-mold mount systems and floor-based systems that use an air table and robot to assist mold change (Wilder, 1990). The initial implementation of quick-mold-change equipment has occurred in automotive injection-molding plants intent on eliminating unnecessary inventory caused by large fluctuations in demand across a wide product range.

While some use these quick-mold-change systems to augment a traditional injection-molding machine, others suggest using this technology as the basis for a Flexible Manufacturing System (FMS) or cell (Henze, 1988). FMS uses multiple mold changers and
industrial robots to achieve quick changeover times and allow for smaller production runs (Von Eysmondt, 1989). Using a combination of manual and automated quick-mold-change technology, one molder has achieved a level of flexibility that allows production run sizes as small as six pieces (Schut, 1999).

In addition to quick-mold-change systems, other manufacturers have developed a carousel, or horizontal mold rotation, injection-molding system. While these systems were originally developed for thick-part injection molding, the benefits of separate injection, cooling and ejection stages have become apparent with regard to flexibility. The use of a multi-stage rotary system required lower clamping forces and injection pressures, reducing operating costs and tooling requirements. Low pressure molding allows the tooling to be made of more easily machined and lower cost steel (Neilley, 1997). Due to the multi-station nature of these systems, they are ideal for co-injection molding where the core can be replaced during processing (Jaroschek and Steger, 1998). Co-injection molding allows parts to be manufactured with multiple polymer types in one process. Despite the increased flexibility, the number of molds that can move through the system is limited. Furthermore, the movement of the molds is constrained by the rotation of the mold carousel, i.e., the molds are physically coupled. Lack of independent movement of the molds eliminates the potential of a mold exchange station where molds can be exchanged to accommodate mold maintenance and/or changes to the production requirements.

While considerable research has been conducted regarding setting process variables for injection molding, a more limited effort seems to have been applied to the redesign of the injection-molding system itself. The focus continues to be solely on increasing output,
instead of streamlining the injection-molding process by eliminating waste and improving the handling of multiple part types.

1.3. **Background: Lean Production**

The design and evaluation of a new injection-molding system required an established set of design objectives. These objectives were used to guide the conceptual development of the design, and later, acted as measures by which the system’s performance could be assessed. To establish these objectives, elements of ‘Lean Production’ were used. Lean Production can be defined as a manufacturing and managerial philosophy that focuses on reducing all forms of waste in a system to allow a company to handle increased levels of product mix and smaller, more frequent production runs. Of importance to the design model are concepts such as Takt Time, Level Production, Single-Piece Flow, and Right-Size Machines (Cochran, 1999).

1.3.1. **Takt Time**

Lean manufacturing stresses the importance of building no more and no less than that demanded by the customer in order to avoid waste. To achieve this goal, the production rate must match exactly the Takt Time or customer demand rate. Building too quickly creates finished goods inventory, while building too slowly generates the need for overtime, excess transportation, etc. The Takt Time becomes the driver of the production rates and, in turn, the material flow rates throughout the system. To measure a system’s ability to produce parts at the Takt Time, a measure called Build-To-Schedule (BTS) is used. BTS is defined in Equation 1, and illustrated in Figure 1 using an example calculation.
\[ \text{Given:} \]
\[
x_i = \# \text{ part type } i \text{ produced} \\
y_i = \# \text{ part type } i \text{ demanded} \\
B.T.S. = \begin{cases} \\
\frac{x_i}{y_i} & \text{if } x_i < y_i \\
\frac{2y_i - x_i}{y_i} & \text{if } x_i > y_i \\
\end{cases} \\
\text{(Eqn. 1)}
\]

\[ B.T.S. \text{ system} = \prod_{i=1}^{p} B.T.S. \\
where \ p = \# \text{ part types} \]

**Equation 1 - Build-to-Schedule definition**

**Figure 1 - Build-to-Schedule calculation example**

BTS is a metric used to determine how closely the production system matches the
demanded rate. Since the period over which the data can be recorded is unspecified, the
measure becomes less precise for longer data collection periods due to an averaging of the
data. To ensure a useful measure is attained, the BTS should be calculated over a maximum
period of a day.
1.3.2. Level Production

Once the Takt Time is established, the system must be designed to meet these production requirements. This is achieved by setting the constraint operation of the system to meet the Takt Time. When a number of parts are being built, the constraint operation must be capable of running at the shortest required Takt Time.

Producing multiple types of parts introduces an additional consideration in terms of the production run or batch size. Typically, parts are produced in large batches to avoid machine changeover times. However, this practice creates swollen inventories, quality control problems and delayed lead times, and runs counter to Lean Production principles. This situation provides the impetus for a system redesign focused on achieving Level Production. Level Production is achieved by a system that meets the customers demands in terms of both batch size and timing for a variety of parts. The system redesign invariably addresses the problem of reducing changeover times. Current injection-molding systems rely on large production runs due to long changeover times associated with mold or material changes. Figure 2 shows three parts being produced using Batch Production (2A) versus Level Production (2B).

![Figure 2 - (A) Batch Production versus (B) Level Production](image-url)
1.3.3. **Single-Piece Flow**

In conjunction with batch-size reduction associated with Level Production, Single-Piece Flow is introduced to eliminate waste during the movement of parts through multiple operations. Batches that run through multiple operations experience lot delay as each part must wait until the batch is finished before moving to the next operation. Single-Piece Flow, on the other hand, moves individual pieces\(^1\) through each station separately, allowing separate operations to run in parallel. Figure 3 shows Batch Production (3A) compared to Single-Piece Flow (3B) through multiple operations. The total production time is reduced due to the elimination of lot delay.

![Diagram showing Batch Production vs. Single-Piece Flow through multiple operations](image)

**Figure 3** – (A) Batch Production vs. (B) Single-Piece Flow through multiple operations

Current injection-molding machines perform the injection, cooling, and ejection operations serially, as shown in Figure 4A. This causes the system to experience lot delay

\(^1\) The ultimate goal is the movement of individual pieces. Typically, small batch sizes that match the customer order sizes are used.
equivalent to running the entire batch serially through each operation. In Figure 4, the time taken for the parts to run through all operations is the same for both serial (4A) and batch production (4B), which is longer than the case with Single-Piece Flow (4C).

Figures 4 – (A) Traditional injection molding using serial production equated to (B) Batch Production through multiple operations.

With both Level Production and Single-Piece Flow in place, products are pulled through the system with each downstream operation pulling parts from the upstream operation. In this way, all processes will produce at the rate of the constraint operation, which is set to match the Takt Time.
1.3.4. **Right Sized Machines**

Lean Production emphasizes the use of numerous small machines in place of large, multipurpose machines to improve the flexibility and robustness of the overall system. Injection-molding machines are typically rated by the clamping force and/or injection pressure attainable. For facilities running high volumes of multiple parts, the injection-molding machines are fitted with high-cavity molds. The accommodation of high-cavity molds coupled with the continued need for reduced injection times leads to the use of large machines with increased clamping forces. High injection pressures and clamping forces increase the operating cost of the system and lower the system-level robustness. The implementation of Right-Size machines leads to system redundancy using small, uncoupled machines. (Quick, 1989)

1.4. **Research Objectives and Thesis Structure**

The approach introduced in this thesis focuses on the injection-molding apparatus as opposed to the production facility. The project scope encompasses the redesign and optimization of a newly proposed injection-molding system as an alternative to traditional machines. An examination of the traditional injection-molding process identifies the obstacles encountered when introducing a variety of part types, smaller and more frequent production runs, and minimized costs. The conceptual development of the new system, including a description of the proposed embodiment of the design, is presented. To allow for a comparison of the two systems, evaluative tools are developed, facilitating a cost-based feasibility comparison of the traditional and proposed injection-molding machines.

The development of the assessment tool begins with the design model, which provides a framework within which the system inputs and outputs can be related and
measured. An Evolutionary Algorithm (EA) optimization is used as the design function. This provides the relationship between the system-variable inputs and the resulting system outputs. The algorithm uses a number of components customized to suit the design requirements of the proposed system. The combinatorial nature of the problem and the heterogeneous nature of the input-value set led to the development of novel evolutionary entities. First, a genome was developed to codify the system variables. Second, evolutionary operators were developed to mutate the variable values stored in the genome each generation without creating infeasible variable sets.

The following sections outline the steps taken in the development of an evaluative tool for the multi-station, injection-molding system. The system description, complete design model formulation, as well as the details of the Evolutionary Algorithm, are described below.
Chapter 2: Design of a Lean IM System

2. Design of a Lean IM System

2.1. Scope of Design

The implementation of Lean Manufacturing principles in various industries has proved effective in making companies more competitive by eliminating waste from the system while meeting customer demand. Principles of Lean Manufacturing are applied to the injection-molding industry in an effort to improve part quality, productivity, and system flexibility. Of key concern is the system’s ability to produce parts to meet the customer demand rate and, when dealing with multiple part types, the desired proportions.

2.1.1. Traditional-System Design

The design of an injection-molding system that could provide significantly improved flexibility began with a high level functional decomposition of the process. Five key functional requirements were identified as shown in Figure 5.

![Figure 5 - Functional Decomposition of the injection-molding process](image)

The physical entities found on a traditional injection-molding machine that are used to satisfy these requirements are shown in the rightmost vector in Figure 6. The relationship between the functional requirements and the physical design features of the system is captured by the design matrix. In the matrix, a strong relationship is indicated by the symbol ‘x’, while a weak, or non-existent relationship is indicated by the symbol ‘o’. For example,
the 'x' in the upper left corner of the design matrix indicates that the sprue and nozzle affects the receiving of the melt.

![Design of a traditional injection-molding system](image)

**Figure 6 – Design of a traditional injection-molding system**

The design matrix reveals four problematic relationships that create a coupled design, which often implies inflexibility and difficulty in controlling machine operation. These relationships are highlighted and numbered in Figure 6. The first relationship involves the interaction of the cooling channels with the distribution of the melt. A cool mold can affect the distribution of the melt through the runners and gates and into the mold itself. This problem can create short shots due to melt freezing at the gates or thin-walled sections. Freezing is often addressed by increasing the injection pressure, which reduces its occurrence and reduces the processing time. The processing costs, however, are increased and the high pressure causes accelerated mold wear.

While the first relationship exhibited a direct coupling of the system, the second, third, and fourth relationships occur because of the sequential execution of process steps in a traditional injection-molding machine. These three relationships only create a coupled system when the functional requirements have to be satisfied in parallel. Since traditional injection-molding systems do not operate these steps in parallel, this is not typically considered a problem. However, guided by the principles of Lean Production, this reliance on sequential or serial operations (temporal coupling) is itself a problem due to the creation
of lot delay. These three relationships highlight couplings in the system between the design features of the mold and the injection-molding machine. If eliminated, some of the functional requirements could be run in parallel, eliminating lot delay. The relationships are examined in more detail below.

The second relationship identified in Figure 6 describes the interaction between the cooling of the melt and the core and cavity. Here the core and cavity, which are designed for part formation, can affect the cooling of the melt. The problem arises because additional parts cannot be formed while the melt is cooling. The operations are coupled due to the temporal reliance of two functional requirements on a single design feature.

The third and fourth relationships also involve a temporal coupling of components. In this case, parts can only be ejected from a mold once the part has been formed and the melt has cooled. The problem arises because the core and cavity, cooling channels, and ejector pins are all physically coupled. Apart from the temporal coupling exhibited by the fourth relationship, the core and cavity affect part ejection due to part shrinkage onto the core. This problem, however, can be eliminated through proper mold design by incorporating adequate drafts and the proper treatment of undercuts.

The four coupled relationships identified provided the impetus and direction for the redesign of the injection-molding system. The new-system design is described in the following section.

2.1.2. New-System Design

Figure 7 shows the six functional requirements of the new system along with a new set of design features. The corresponding design matrix is included to indicate the relationships that now exist.
Each of the abovementioned problems was eliminated using the new set of design features. The first problem, relating to short shots due to the solidification of melt during its distribution, was overcome by introducing a new design feature to heat the mold prior to injection. A heated mold eliminates the risk of frozen gates, and allows thin walled parts (<1mm) to be manufactured without the need for special polymers and high injection pressures. By grouping the second, third and fourth functional requirements together to form a single unit, the final design can be broken into four uncoupled units, a heating unit, an injection unit, a cooling unit, and an ejector unit. The components within the injection unit still exhibit a temporal coupling, but because these functional requirements are to be satisfied serially by one physical unit, this relationship does not pose a problem. The second, third and fourth problems were solved by using multiple molds and stations, i.e., cooling and ejection units, to allow the respective functional requirements to be temporally uncoupled.

With multiple molds moving through a series of functionally and physically uncoupled operations, the system is able to perform operations in parallel. Furthermore, additional stations can be added to help balance the operating times to minimize idle times. Using multiple molds means that the number of cavities of each mold can be smaller. This leads to lower clamping forces and injection pressures, a design consistent with the principle of Right Sized Machines.
Chapter 2: Design of a Lean IM System

The new multi-station injection-molding system consists of four uncoupled stations: heating, injection, cooling, and ejection - a design consistent with Axiomatic Design (Suh, 1990). The following section describes the proposed embodiment of the new system.

2.2. Proposed Embodiment

The proposed embodiment of this system includes separate stations for each of the four units, connected by a mold conveyor system. The stations are arranged in a manner that allows a mold to sequentially visit each of the stations in the system until their combined series of processes form a completed part. The system allows for the movement of multiple molds through the system producing single- or multiple-part types. A mold will only move to the next station if an opening arises, i.e., the molds are pulled through the system. The number of molds per part and the number of cavities for each of those molds can vary. Additionally, the capacity of each of the stations or the preceding buffers can vary. Because the stations operate in parallel, the additional heating station could be added to the system with little additional processing time incurred.

The addition of a heating station and the ability to handle multiple molds remove the temporal coupling in the system, which leads to a number of benefits in terms of flexibility, throughput and quality. First, the heated molds allow thin-walled parts to be produced, and require smaller clamping forces and injection pressures. This leads to reduced mold wear and less sensitivity to residual stresses caused by thin-thick wall transitions.

Second, because the cooling of the melt is performed in parallel with the other operations, the need for complex cooling systems is reduced. With a cycle time less dependent on rapid cooling rates, the operating conditions are brought into a more easily
controlled region where residual stresses and anisotropy, due to rapid cooling, can be more easily controlled.

Third, by having molds for each part type move through the system together, the system can handle multiple part types. This allows different versions of a part or part families to be produced at the same time. By varying the number of molds and their respective cavities, different numbers of units of each part type can be produced (during one cycle of molds through the system). The system can therefore meet the demands for the multiple part types.

Fourth, because multiple molds are used to produce a single part type, molds can be added, removed or exchanged to adjust to small fluctuations in demand. In addition, mold maintenance can be performed on individual molds without the need to shut down the machine. To accommodate more molds, the stations and buffer capacities could be increased. By allowing the station capacities to change, the operation time of the constraint operation can effectively be changed so that the Takt Time can be met and the operation times can be balanced. This improved processing ability comes at a cost, however. Additional molds, cavities, and station and buffer capacities mean higher mold and machine costs.

Figure 8 shows the proposed embodiment of the design. Two mold types are shown, reflecting the system's ability to concurrently handle the varying part types. The number of molds per part and the number of cavities for each of those molds can vary, as can the capacity of each of the stations or their associated buffers.
Figure 8 – Proposed embodiment of the new injection-molding system

The following chapter describes the development of the design model used to optimize the new proposed system configuration and allow for a fair assessment of the cost/benefit tradeoffs of the new system with respect to the traditional injection-molding system.
3. Design Model Definition

3.1. Introduction

Design modeling provides a formalized structure to the solution of the design problem at hand. This problem can be described as follows.

Given the production volume and part types for a set of parts to be manufactured, find the best system configuration for both the new and traditional systems that will satisfy the production requirements and minimize costs.

Once determined, these system configurations can be used in various cost analyses of the new machine versus the traditional equipment based on different production volumes, part thickness, and number of part types. The design model guides the development of an evaluative tool to determine the optimal system configuration and evaluate the design.

The design modeling process consists of five elements: design outputs, design objectives, design inputs, design range and the design function. Identifying these elements helps to organize the information required for problem formulation and forms the basis for problem description using an optimization formalism. It should be noted that this form of design modeling is typically used during the detailed design stage of a new system. Since the focus of this thesis is to establish a framework within which the benefits of a new system can be assessed, a formal design modeling process is used during the conceptual stage of development. Figure 9 shows the relationship between the design modeling elements.

![Figure 9 - Schematic of the Design Model](image)
The design model is a mapping of a particular configuration or state of the system, expressed as design inputs, into the set of design requirements to be satisfied. The mapping is achieved using the design function, which can be analytical or simulation based. The following sections explain the elements of the design model in detail. Furthermore, these elements are described as they apply to the design models for the newly proposed system and the traditional system. Figure 10 shows the elements of the design models for the multi-station and traditional injection-molding machines. Elements that are alike in both models are depicted using the same shading while elements that differ are shown using different shading. The sections/chapters in which each of the elements is discussed are shown to the left of each element.

3.2. Design Outputs

Design outputs are the main features of the design that must be monitored or evaluated to ensure the design fulfils its requirements. The two criteria of the objective function were selected to directly address the needs of industry, i.e., helping companies to become more profitable and better serve the needs of their consumers. Both are in line with Lean Production principles stressing the system’s ability to match production with the demand rate. The design outputs are:

1. Build to Schedule (BTS) – a measure of the ability of the system to build the correct type, quantity, and mix of parts.

2. Total Cost – a measure of the system level cost.

The first criterion, BTS, is aimed at achieving Level Production and Single-Piece Flow. The second criterion, Total Cost, helps to lift profitability by ensuring that the aforementioned flexibility is attained at the lowest possible cost.
3.3. **Design Objective**

The design objectives describe the designer’s intent regarding the constraints or desired values of each design output. Design objectives can be grouped into one of two categories, *wish* objectives and *must* objectives. *Wish* objectives describe those outputs where minimized, maximized, or targeted values are desired (called smaller-the-better,
larger-the-better, or on-target-the-better, respectively). *Must* objectives describe outputs that have a defined range of permissible values that ensure the design requirements are met. The design objectives for this model are:

*Wish* Objectives:

a. Cost - smaller-the better

*Must* Objectives:

b. Build-to-schedule must equal 100% (B.T.S. = 100%)

### 3.4. Design Inputs

The design inputs are divided into two types: design parameters and design variables. Parameters are fixed during the solution of a particular design case. However, they can change from one design case to another, specifically when a range of problems is examined. The design variables, however, are changed during the design process in order to achieve the desired output. Design variables and parameters are represented using the following vector notation:

Design Variable Vector \( \mathbf{X} = \{ X_m \}_{m \in I, [1, \# \text{ variables}]} \)

Design Parameter Vector \( \mathbf{P} = \{ P_n \}_{n \in I, [1, \# \text{ parameters}]} \)

Design Input Vector \( \mathbf{Y} = \{ \mathbf{X}, \mathbf{P} \} \)

The design variable vector represents one of the many system configurations that may be input into the design model, and along with the parameter vector represents the total input set for the design model.
3.4.1. Design Parameters

The design parameter vector is the same for both the traditional and new design models. The parameters, \( P_1 \) to \( P_{34} \) are grouped into five categories: part, material, machine, processing, and optimization. Examples of the parameters from each category are listed in Table 2. For a complete list, see Appendix I.

<table>
<thead>
<tr>
<th>Term used in the design model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_3 ) L-thickness</td>
<td>Max Wall Thickness</td>
</tr>
<tr>
<td>( P_5 ) L-width</td>
<td>Width</td>
</tr>
<tr>
<td>( P_6 ) L-height</td>
<td>Height</td>
</tr>
<tr>
<td>( P_{14} ) Mat_i</td>
<td>Material Type i</td>
</tr>
<tr>
<td>( P_{20} ) F_clamp</td>
<td>Clamping Force</td>
</tr>
<tr>
<td>( P_{23} ) T_available</td>
<td>Available time per day</td>
</tr>
<tr>
<td>( P_{24} ) days</td>
<td>Production days per year</td>
</tr>
<tr>
<td>( P_{26} ) MoldPeriod</td>
<td>Mold Amortization Period</td>
</tr>
<tr>
<td>( P_{27} ) MachinePeriod</td>
<td>Machine Amortization Period</td>
</tr>
<tr>
<td>( P_{29} ) population_size</td>
<td>Population size</td>
</tr>
<tr>
<td>( P_{32} ) number_of_generations</td>
<td>Number of Generations</td>
</tr>
</tbody>
</table>

The values for each of these parameters are held constant during the execution of the design model, i.e., the mapping of the design inputs into design outputs using the design function.

3.4.2. Design Variables

The design variable vector differs for the two design models. The design variable vectors for both models are listed in Tables 3 and 4 below.
### 3.4.2.1. Traditional System Design Variable Vector

**Table 3 - Traditional system design variable vector**

<table>
<thead>
<tr>
<th>Term used in the design model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_{1T} ) - Molds</td>
<td>PercentPart array ( % ) parts handled by machine ( j )</td>
</tr>
<tr>
<td>( X_{2T} ) - Molds</td>
<td>NoCav array # cavities per mold ( ij )</td>
</tr>
</tbody>
</table>

In the traditional system design model, it is assumed that a maximum of one mold for each part type exists for each machine that produces the part. The percent of units of part \( i \) handled by machine \( j \) can vary, as can the number of cavities per mold. Chapter 5 provides a complete description of the traditional machine design function describing the mapping of these variables and parameters into a corresponding set of design outputs.

### 3.4.2.2. New System Design Variable Vector

**Table 4 - New system design variable vector**

<table>
<thead>
<tr>
<th>Term used in the design model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_1 ) - Station Capacity</td>
<td>SCap\text{heat}</td>
</tr>
<tr>
<td>( X_2 ) - Station Capacity</td>
<td>SCap\text{inject}</td>
</tr>
<tr>
<td>( X_3 ) - Station Capacity</td>
<td>SCap\text{cool}</td>
</tr>
<tr>
<td>( X_4 ) - Station Capacity</td>
<td>SCap\text{eject}</td>
</tr>
<tr>
<td>( X_5 ) - Buffer Capacity</td>
<td>BCap\text{heat}</td>
</tr>
</tbody>
</table>

Heating Station Capacity  
Injection Station Capacity  
Cooling Station Capacity  
Ejection Station Capacity  
Heating Station Buffer Capacity
Chapter 3: Design Model Definition

| \(X_6\) | BCap\text{\textsubscript{project}} | Injection Station Buffer Capacity |
| \(X_7\) | BCap\text{\textsubscript{cool}} | Cooling Station Buffer Capacity |
| \(X_8\) | BCap\text{\textsubscript{eject}} | Ejection Station Buffer Capacity |

Molds

| \(X_9\) | | No. molds for each part |
| \(X_{10}\) | | No. cavities per mold |

The new system design model incorporates a number of additional variables to capture the new features of the proposed embodiment. These variables include the capacities of the four processing stations and their respective buffers. In addition, the number of molds for each part type can vary along with the number of cavities for each of those molds. The mapping of these design variables and parameters into design outputs is discussed further in Chapter 6.

3.5. Design Range

The design range defines the extent of possible values of each of the variables or parameters. With regard to parameters, the range typically refers to design alternatives, with the allowable values being based on existing materials and standard part sizes, etc. The variable range involves realistic constraints of the design variables that limit the design variable space.

3.5.1. Parameter Range

See Appendix II for a complete list of the parameter ranges used in the new- and traditional-system design models.
3.5.2. Variable Range

Tables 5 and 6 show the traditional and new system variable vector ranges.

**Table 5 - Traditional system variable vector range**

<table>
<thead>
<tr>
<th>Term used in the design model</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Molds</td>
<td></td>
</tr>
<tr>
<td>( X_{1T} )</td>
<td>PercentPart array</td>
</tr>
<tr>
<td>( X_{2T} )</td>
<td>NoCav array</td>
</tr>
</tbody>
</table>

**Table 6 - New system variable vector range**

<table>
<thead>
<tr>
<th>Term used in the design model</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Station Capacity</td>
<td></td>
</tr>
<tr>
<td>( X_1 )</td>
<td>Heat</td>
</tr>
<tr>
<td>( X_2 )</td>
<td>Inject</td>
</tr>
<tr>
<td>( X_3 )</td>
<td>Cool</td>
</tr>
<tr>
<td>( X_4 )</td>
<td>Eject</td>
</tr>
<tr>
<td>Buffer Capacity</td>
<td></td>
</tr>
<tr>
<td>( X_5 )</td>
<td>Heat</td>
</tr>
<tr>
<td>( X_6 )</td>
<td>Inject</td>
</tr>
<tr>
<td>( X_7 )</td>
<td>Cool</td>
</tr>
<tr>
<td>( X_8 )</td>
<td>Eject</td>
</tr>
<tr>
<td>Molds</td>
<td></td>
</tr>
<tr>
<td>( X_9 )</td>
<td>No. molds for each part</td>
</tr>
<tr>
<td>( X_{10} )</td>
<td>No. cavities per mold</td>
</tr>
</tbody>
</table>
3.6. Design Function

The design function is a mapping between the design inputs and the design outputs, incorporating the designer's intent captured by the design objectives. In this case, two design objectives are to be satisfied, the must objective of 100% Build-to-schedule, and the wish objective to minimize the total cost. To achieve the first objective, the design function had to ensure that the number of parts built during a given time period matched the customer demand rates for the various part types.

Satisfying the design objectives affects the design functions of the new and traditional systems in different ways. In the case of the traditional system, the application of the optimization is straightforward given specific ranges for the each of the design variables. On the other hand, for the new system, because the number of cycles of molds through the system could vary, the design function only requires that the ratio of parts built during each cycle matched the ratio of customer demand rates for all part types.

To meet the design objectives, the frequency, or number of cycles through the system multiplied by the number of units of each part produced per cycle must equal the quantity demanded. Equation 2 illustrates this relationship.

\[
\text{Frequency or number of times the molds circulate through the system} \times \text{number of units of each part produced per cycle} = \text{Customer demand for each part type} \quad (\text{Eqn. 2})
\]

Equation 2 – Achieving 100% BTS with varying frequencies

Given the customer demand rates for all part types and the possible set of frequencies, the number of units to be produced per cycle can be calculated. For example, Table 7 illustrates the various acceptable cycle quantities with the respective frequency that yields the demanded part quantities (given the customer demand for each part of 30 units of part A, 18 units of part B, and 6 units of part C).
Furthermore, within each of the frequency-ratio couplets, there exist a number of possible mold and cavity combinations to achieve the cycle quantity. For example, (using the values from Table 7) with a frequency of 6 (fourth column), the system must produce 3 parts (third row) of Type B each cycle. This can be accomplished using any of the following mold combinations,

- 1 mold – 3 cavities
- 2 molds – 1 cavity, 2 cavities
- 3 molds – 1 cavity, 1 cavity, 1 cavity

N.B. The sum of cavities across all molds equals three.

The design function tests all possible frequencies as determined by the largest common denominator across the customers demand rates for the various part types. The number of molds and cavities whose values are captured by the design variable vector are then optimized during the execution of the design function. Satisfying the must design objective of 100% Build-to-Schedule translates to a constraint on the design where the possible mold cavity and frequency combinations are limited. The wish design objective, minimizing total cost, forms the objective function of the system optimization.
formulation of the optimization problem using all the elements of the design model is shown in Figure 11 below.

![Figure 11 - Optimization Formulation](image)

Due to the large number of design inputs and complex nature of the variable and parameter ranges for both the new and traditional systems, i.e., the fact that many can take on integer or fixed values, the optimization becomes difficult to solve using a closed-form solution. The combinatorial nature of the problem introduced by the need for the production rate to meet the customer demand rate prompted the use of an Evolutionary Algorithm (EA). This type of algorithm provides the ability to search through a complex design space quickly and efficiently. Although the global optimal solution is not guaranteed, it was felt that the algorithm would be suitable for the feasibility analysis.

The following chapters provide a complete description of EAs and their use in both the new and traditional systems' design functions. Chapter 4 describes the structure, types, and applicability of EAs. Chapters 5 and 6 cover the traditional- and new-system design functions including details of the objective function, and the development of new EA components (genome and operators) designed to capture and explore the acceptable variable values subject to design ranges and optimization constraints.
4. Solution Approach – Evolutionary Algorithms

4.1. Introduction

An Evolutionary Algorithm (EA) is an umbrella term used to describe computer-based, problem-solving systems that use computational models based on evolutionary processes as key elements in their design and implementation. Typically, four problem-solving strategies are included.

- Genetic Algorithm (GA) - (Holland, 1975) and (Goldberg, 1989)
- Evolutionary Programming (EP) - (Fogul et al., 66)
- Evolutionary Strategies - (Rechenberg, 1973)
- Genetic Programming - (Koza, 1992)

While all share a common conceptual base of simulating the evolution of individual structures through a process of selection, mutation and reproduction, there are differences in the approach and therefore, applicability, of each strategy to various problems. For the new and traditional-system design functions, Evolutionary Programming algorithms were used. To better appreciate the suitability of EPs to the problem at hand, a more thorough examination of the evolutionary process is necessary. First, however, the applicability of evolutionary algorithms as a whole is discussed.

Evolutionary Algorithms are optimization algorithms effective for searching through a multi-dimensional solution space. In particular, EAs are useful when traditional search methods, which often require smooth, continuous variable spaces, cannot find solutions in a reasonable period of time due to the size or complex nature of the search space. Figure 12 illustrates the relative efficiency of traditional search schemes compared with the more robust evolutionary approaches. In Figure 12, an illustrative, albeit fictitious, efficiency index is
plotted across a problem continuum for a specialized search approach, enumerative approach, and an evolutionary approach. The specialized approach, including such calculus-based search methods as the gradient approach, performs well in its narrow problem class, but becomes highly inefficient elsewhere. In contrast, the enumerative search approach works equally inefficiently across a broad spectrum as shown by the lower performance curve. The evolutionary approaches, however, are more robust and work well across a broad spectrum of problems including combinatorial and multi-modal problems. These best describe the type of problem in the new and traditional-system design functions. It should be noted that the evolutionary approach is non-exhaustive and does not guarantee the optimal solution. In some cases involving unimodal problems, the evolutionary approach may be outperformed by a specialized approach. However, the overall robustness of the approach, i.e., the ability to handle a broader spectrum of problem types, is often more desirable, as was the case in both design models.

![Figure 12](#)

**Figure 12** – Efficiency of various type of search and optimization approaches with respect to problem types (adapted from Goldberg, 1989)
The following section outlines the general steps performed by evolutionary algorithms. This background provides a basis for further discussion concerning the choice of using an Evolutionary-Programming-based Evolutionary Algorithm.

4.2. *Evolutionary Process*

All Evolutionary Algorithms follow the key steps shown in Figure 13.

![Figure 13 - Key steps in an Evolutionary Algorithm](image)

4.2.1. **Create Genome Population**

The algorithm begins by randomly generating a population of feasible genomes. Each genome is a codified set of elements that captures the information of a particular variable set and represents one possible solution. The structure of the genome is an important factor in the classification of the various types of EAs. The different types and requirements for representation are described further in section 4.3. The use of a population
of genomes is an important feature of EAs. Unlike most other optimization techniques that track the movement of a single point through the search space, EAs allow the evolution of multiple points through the search space.

4.2.2. Test for Fitness

The genomes are then tested for fitness using the cost objective function. The objective function calculates the total cost of the system given the design parameters of the system and the variable set specified by the genome. The specifics of the objective function are explained in Section 4.5.

4.2.3. Select Fittest Genomes

The fittest genomes are selected for use in the population of the following generation using a specified selection scheme. These genomes form the basis for a new population. A number of selection schemes are used in common practice. Section 4.6 describes the various alternatives in more detail.

4.2.4. Apply Evolutionary Operators and Generate New Population

Evolutionary operators are then used to modify the above genomes selected for the following generation, thereby creating new genomes to complete the population. This is a fundamental step in the optimization process. A variety of evolutionary operators has been proposed in the literature and their use depends heavily on the solution representation in the genome. Section 4.7 describes the types and usages of the operators in more detail.

Next, the genomes in the new population are tested and the process repeats for a specified number of generations. The process continually generates new variable sets giving the highest chance of survival to the fittest genomes. For each generation, the maximum,
minimum, and mean scores for the population are tracked, providing a method by which the convergence of the algorithm can be monitored.

The algorithms used for the two design models were developed in the C++ programming language using the Genetic Algorithm Library 2.4.5 (Wall, 1996).

4.3. Solution Representation - Genome

The selection of the type of genome to be used greatly affects the fitness test and the evolutionary operators which are the steps that are concerned with the interplay between the input space with the solution space. Typically, with Genetic Algorithms (GA), the genome consists of a binary string used to encode the design variable vector. The fitness function and operators evaluate and manipulate this abstract encoding during the operation of the algorithm. Evolutionary Programming (EP), on the other hand, maintains an underlying assumption that the solution space can be characterized directly in terms of the system variables as opposed to a binary or other representation of those variables. In this case, the operators directly manipulate variable values. The direct representation allows complex design variables consisting of elements with combinatorial, continuous, and stepwise ranges to be more easily captured, but increases the complexity of the operators that can be used. The reason for this increase in complexity lies in the fact that certain properties of the variable set must be maintained during the manipulation of the operator to avoid an illegal or infeasible solution. An illegal solution arises when an evolutionary operator creates a genome that does not represent a valid solution. Infeasibility describes a solution that lies outside the feasible region of the solution space, and often occurs as a result of a highly constrained optimization problem. Infeasible solutions increase the complexity of the fitness
function, which must now be able to identify infeasible genomes and assign penalties to lower their fitness and effectively eliminate them from the following population.

Although the distinction between the type of genome used in GAs and EPs is one of the main differences between the two approaches, many of the required properties of solution encoding that have been identified primarily for GAs can be applied to EPs. The relevant properties are discussed briefly below (Gen, M. and Cgeng, R., 2000):

Non-redundancy – The mapping from variable space to the solution space should be 1-to-1. Mappings from the genome to the solution fall in to one of three categories. A 1-to-1 mapping is the most desirable with each genome representing only one distinct solution. Less desirable, but still acceptable, is the n-to-1 mapping where n genomes correspond to the same solution. This situation leads to wasted time during the optimization as redundant genomes are investigated. The last case, a 1-to-n mapping is unacceptable since a genome can result in multiple solutions. Given a solution in this case, the corresponding variable set cannot be determined, making the algorithm ineffectual.

Completeness – Any solution has a corresponding encoding. This property guarantees that any point in the solution space is accessible during an evolutionary search.

Lamarkian – The meaning of the variable values is not context-dependent. If a genome exhibits the Lamarkian property, then the variables in each genome can be interpreted independently and are not dependent on one another. This property is important to ensure that genomes can pass its merits to future populations through the evolutionary operators. In the non-Lamarkian case, the portions of the design variable vector that are passed from genome to genome through an operator may contain a different meaning based on the context, and therefore, do not allow the child-genome to inherit any ‘goodness’ from
the parent. Some degree of the Lamarkian property is always sought when selecting the genome structure. For example, consider a genome that codifies the shortest path that can be followed to visit a series of randomly positioned points. If four points are to be visited, the genome might appear as a string, listing the order of points to visit, e.g., [2,4,1,3]. Because this genome is order dependent, when selecting an operator for this genome, care should be taken to ensure that a portion of the ordering of this genome is transmitted to subsequent genomes during reproduction. A popular operator used in this case would involve dividing the genome at a randomly selected point, e.g., [2,4,1|3], and then swapping the two sections to create a new genome, e.g., [3,2,4,1]. In this way, the some of the merits of the parent genome can be passed to the children of each generation.

4.4. Population Generation

After the type of genome to be used has been determined, the initial population of genomes is created. The population is generated using an initialization function, which creates a specified number of genomes. Typically, the genomes are randomly generated covering the complete design space; however, some initialization functions exploit prior knowledge of the system and create genomes in a particular region of the variable space to reduce computation times.

4.5. Fitness Evaluation

The objective function provides an evaluative measure of the fitness of a genome. The function provides an output score for a set of variables and parameters, which is used to measure the genome's ability to fulfil the two design objectives. The genome and the evolutionary operators were designed to satisfy the must design objective of 100% Build-to-
Schedule. The objective function helps to achieve the *wish* design objective of minimizing the total cost of the system by providing an evaluation of the cost for a given genome. That is, the objective function will provide the total cost of the system based on a given codified set of variables. The EA uses this fitness measure to ensure that the lower cost options are given a better chance of survival in each generation.

The total cost of the system consists of mold costs, processing costs, station costs, and reject costs. These costs are described further in Sections 4.5.1-4.5.4. These sections introduce the key elements of the costs and include details of those elements of the cost evaluation that are common across the two design models. Chapters 5 and 6 build upon this base by adding model-specific details to each of the evolutionary algorithm steps.

Since the processing cost is calculated on an annual basis while equipment related costs are fixed, a common metric was needed. All costs have been converted to an annual cost by amortizing the cost of the equipment over its expected life. The details of the amortization are described in Section 4.5.5.

### 4.5.1. Mold Cost

The mold cost includes the cost of all mold components including the creation of the core and cavities and the base in which the core and cavities are mounted. The cost calculation begins with the cost analysis of a single cavity mold. This cost includes the mold-base cost and the core and cavity processing costs, which are based on part size and complexity, and material type. The cost of a multi-cavity mold is then calculated using Equation 3. This formula is used instead of a pure summation of the cavity costs to account for the design and CNC programming cost savings made when machining duplicate cavities (Boothroyd, et al., 1994).
\[ C_{\text{multicavity mold}} = (C_{\text{single cavity mold}})^{n^m} \]  
(Eqn. 3)

where \( m \) is the multi-cavity mold index = 0.7
\( n \) is the number of cavities

Equation 3 – Multi-cavity mold cost calculation

This multi-cavity mold cost is used as the base for the total mold costs in the traditional and new systems. The cost of the molds in the traditional system were summed (section 5.4), while additional calculations were needed in the new-system design model to account for the multiple molds that circulate through the system (section 6.4).

4.5.2. Processing Cost

Processing costs include the hourly cost of running the injection-molding machine. This cost is highly dependent on the clamping force required, which itself is dependent on the size, complexity and number of cavities in each mold. When multiple molds are considered, each machine or station must be capable of handling the largest and most complicated mold.

The processing cost calculation is based on the calculation used by Boothroyd, Dewhurst and Knight (Boothroyd, et al., 1994). Equation 4 finds the total cost by multiplying the machine hourly rate by the total time, which consists of the cycle time multiplied by the number of cycles.

\[ C_{\text{processing}} = (K_1 + K_2F)(N_i / n)t \]  
(Eqn. 4)

where:
\[ K_1, K_2 = \text{machine rate coefficients} \]
\[ F = \text{clamp force (kN)} \]
\[ N_i = \# \text{molded parts required} \]
\[ n = \text{number of cavities in the mold} \]
\[ t = \text{total cycle time} \]

Equation 4 – Cycle time based processing cost calculation
Equation 4 is modified when used to evaluate the traditional and new systems. In these cases, the machine hourly rate is multiplied by the total available time for production. Because the two systems are being evaluated with respect to Lean Production principles, the time used for production is the total available time. Here, the rate of production is altered to ensure that the number of parts produced matches the demand rate. The modified formula, Equation 5, is shown below.

\[ C_{\text{daily processing cost}} = (K_1 + K_2 F)(\text{available time per day}) \]

(Eqn. 5)

where:
- \( K_1, K_2 \) = machine rate coefficients
- \( F \) = clamp force (kN)

Equation 5 – Processing cost calculation for the traditional and new systems

The clamp force is found by first calculating the maximum separating force generated during the injection of the molten plastic. This force is found by multiplying the total projected shot area of the part (a function of part geometry) by the maximum cavity pressure (50% of the recommended injection pressure for the type of polymer used).

The machine rate coefficients \( (K_1, K_2) \) are derived from the US national average injection-molding machine rates. They represent the coefficients of the linear relationship between the hourly injection-molding machine rate and the machine clamp force. For the two design models, the coefficients were recalculated using current data from year 2000 converted to Canadian dollars (CAD) (Plastics Technology Magazine, 2000) as shown in Figure 14. Based on Figure 14, the values used for \( K_1 \) and \( K_2 \) were 25.013 $/hr and 0.0075 $/hr/kN, respectively.
4.5.3. Machine or Station Cost

The machine or station costs include the initial investment in the fixed hardware of the injection-molding system. The specific costs used in the traditional and new design models are explained in Sections 5.4.3 and 6.4.3 respectively.

4.5.4. Reject Costs

Reject costs are costs introduced artificially into the total cost equation to ensure that particular genomes are given a disadvantage during the propagation of genomes to create the genome population for the next generation. The details of the reject conditions used in the two design functions are specified in Sections 5.4.4 and 6.4.4.
4.5.5. Amortization

To convert all costs to an annual cost, the equipment costs, including the mold costs and the machine or station costs, were amortized over the valuable life of the unit. For molds, an amortization period of three years was used, while five years was used for machine and station costs. These values were estimates gathered from interviews with a local injection-molding facility (personal communication, April 2001). This model assumes a 10% annually compounded interest rate and no resale value at the end of the period. These assumptions err on the side of caution and, therefore, provide conservative estimates. The calculation of the yearly payments, given the cost of the unit, is often referred to as the capital recovery factor. See Equation 6.

\[
A = P \frac{i(1+i)^n}{(1+i)^n-1}
\]

(Eqn. 6)

where:

- \(A\) - annual cost
- \(P\) - cost of the unit
- \(i\) - interest rate
- \(n\) - number of periods

Equation 6 – Capital recovery factor

4.6. Selection Process

The selection process directs the genetic search toward promising regions of the search space. Genomes with more desirable traits are selected for reproduction based on the type of selection scheme chosen. A number of selection schemes have been proposed with the Tournament and Roulette-Wheel selection schemes dominating in popularity. The Tournament selection scheme mimics the type of mating structure often found in herds of animals where stags compete for the privilege of mating with a herd of hinds. Under this scheme, a set of genomes, called the tournament size, is selected from the population.
The best single genome per tournament is then selected for reproduction. This selection scheme exhibits both stochastic and deterministic features simultaneously.

In contrast, the Roulette-Wheel selection scheme assigns to each genome a survival probability proportional to the fitness value as compared to the rest of the population. These probabilities are used to create a model roulette wheel that is spun a number of times equal to the population size. The resulting genome from each spin is selected to survive and reproduce. Any individual has a probability $p$ of being chosen, where $p$ is equal to the fitness of the individual divided by the sum of the fitnesses of each individual in the population. By using a spinning wheel with relative percentages, the selection process becomes a stochastic sampling procedure. While the choice of selection scheme is largely arbitrary, Roulette-Wheel selection was chosen for the two design models due to its simplicity and compatibility with asexual reproduction characteristic of evolutionary programming algorithms. The various types of reproduction, and the mechanisms involved, are described in the following section.

4.7. Evolutionary Operators

The evolutionary operators are functions that modify the genomes in a population to create candidates for the next generation. Classic GAs make use of two types of operators: crossover and mutation. The crossover operator recombines two parent genomes to produce two offspring. This type of operator mimics sexual reproduction where genetic information from the two parents is combined randomly to generate the genetic information of the child. The mutation operator randomly changes elements of the genome in the child. These changes are often geared toward exploration of the search space, while the crossover operator is geared toward a movement toward a local optimum.
Because of the existence of complex constraints inherent in combinatorial optimization problems, the GA's use of a simple abstract encoding, typically binary, with crossover and mutation operators often yields many infeasible and illegal solutions. To address this problem, Evolutionary-Programming (EP) was used. As noted in Section 4.3, this type of EA uses a more direct encoding of the design variables in the genomes. In addition to the genome structure, EP differs from the GA in the type of evolutionary operators used. With no constraint on the genomic representation required by the combinatorial nature of the problem, EP uses mutation as the only operator. In effect, EP generates a new population of genomes by using asexual reproduction. Because of the customized genomes used in each of the design functions, new and novel mutation operators were developed. The operation of these operators on the genomes of each design function is explained in detail in Sections 5.6 and 6.6.
5. **Traditional-System Design Function**

5.1. **Introduction**

Chapter 4 described in detail the benefits of using evolutionary algorithms and the operation and key components of the evolutionary process. With this background in place, attention is now turned toward the specific evolutionary algorithm used as the design function of the traditional system design model.

The traditional-system design function maps the design variables and parameters into a near-optimal system cost using an Evolutionary-Programming algorithm. While this algorithm’s objective function addresses the design objective of minimizing cost, the other objective, 100% BTS, is addressed by the genome and operators used in the algorithm. During the operation of the EP algorithm, the maximum, minimum, and mean scores for the population were tracked during each generation, providing a method by which the success of the evolution could be monitored. The following sections outline the key elements of the EP algorithm used.

5.2. **The Genome**

The genome structure used was based on a 3D array where each element in the array could take on one of a specified set of values called the allele set. The allele set used consisted of integer values ranging from 0 to 5. The number of values in the allele set is referred to as the *granularity*, which corresponds to the level of detail that can be accommodated by the model. A low *granularity* leads to a coarser model but that which is less computationally demanding, while a high *granularity* model includes more detail but requires more computing resources. A granularity of six was used for this model since
higher values increased the computation time without providing a corresponding increase in optimization accuracy.

5.2.1. Genome Structure

The 3D-array-allele structure was used because it allows the codification of design variables along each of the three axes of the genome. The height of the genome represents the number of part types, while the width represents the maximum number of machines that may be used by the system. This value was chosen to be equal to the number of part types to ensure that each part type could be built on its own machine, if necessary. The depth of the genome is set to 2 levels, where each level represents different design variables. The first level stores values that are a codified representation of the number of units of a part type (A, B, etc.), that machine (1,2, etc.) will build. The second level stores codified information regarding the number of cavities in the mold used to build the number of units specified in the first level. These values are called the cavity-run index. Figure 15 shows a 2x2x2 array randomly filled with values from the allele set (0,5).

![Diagram](image)

**Figure 15 - 3D array allele genome**
Figure 15 depicts the two levels of depth in the 3D array using two shades. It is important to note that the allele-set values that fill the array are codified representations of the actual values. To find the direct representation of the variables, i.e., the actual number of units of each part type to be produced by each machine, and the number of cavities for each mold, the array undergoes a number of transformations.

5.2.2. Transformation 1

The first transformation calculates the proportion of units of each part type handled by each machine. The units of a part type (A, B, etc.) are first added across the machines (1, 2, etc.). Each element in the row is then divided by this sum value giving the portion of each part type handled by each machine. Figure 16 shows the calculation of the fraction of parts of type B to be handled by machine 2. Here, the sum of allele-set values representing part B across machines 1 & 2 equals 4, and the fraction of part type B built by machine 2 equals $\frac{1}{4}$.

**Figure 16 – 3D array - Transformation 1**

\[
\text{fraction of parts to be produced} = \frac{\text{allele set value}}{\sum \text{allele set values}} = \frac{1}{3+4} = \frac{1}{4}
\]
5.2.3. Transformation 2

The second transformation incorporates the given production volumes. The production volumes for a particular part type are multiplied by each of the proportional values related to that part type, obtained in transformation 1. This transformation converts the genomic material into the actual number of units of each part type each machine will have to produce. Figure 17 shows the second transformation applied to each element in the array. The mechanics of transformation 2 are shown for the fraction of part type B run on machine 1 (3/4). This fraction was then multiplied by the given production volumes for part type B (200 units) to give a value of 150.

5.2.4. Transformation 3

The third transformation makes use of the values stored in the second level of depth in the array. These values are used to determine which of the various cavity-run combinations is used to produce the number of units specified by the genome (first level).
Chapter 5: Traditional-System Design Function

To begin, all the cavity-run combinations are calculated for a given array element. For example, 100 units of part type A are built on machine 2. Nine possible cavity-run combinations can be used to produce exactly 100 parts, as shown in Table 8. The cavity-run combinations represent the pairs of integer values that when multiplied equal the production volume in question.

Table 8 – Cavity-run combinations that can be used to build exactly 100 parts

<table>
<thead>
<tr>
<th>Combination #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cavities</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>10</td>
<td>20</td>
<td>25</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>Runs</td>
<td>100</td>
<td>50</td>
<td>25</td>
<td>20</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Once the cavity-run combinations have been established, the allele-set values stored in the second level of depth in the array are mapped into respective combinations using Equation 7.

\[
\text{round} \left[ \left( \frac{\text{allele\_set\ value}}{\text{max. allele\_set\ value}} \right) \times (\text{# combinations} - 1) \right] = \text{specific combo}
\]

For example (Part A, machine 2, level 2):

\[
\text{round} \left[ \left( \frac{2}{5} \right) (9 - 1) \right] = 3 \quad \Rightarrow \text{combination 3 (5 cavities, 20 runs)}
\]

Equation 7 – Mapping of allele set values into cavity-run combinations

Equation 7 specifies which combination is used. In the example above, the third combination is used, corresponding to 5 cavities and 20 runs. Figure 18 shows the third transformation being applied to part type A run on machine 2.
As defined, the set of possible genomes is highly dependent on the maximum allele-set value, which is dependent on the level of *granularity* of the allele set. The effects of *granularity* on the genome were mentioned previously with regard to the level of detail of the design model, and the computational resources required. These effects are twofold and warrant further discussion.

5.2.5. **Effects of Granularity**

First, the level of *granularity* of the allele set restricts the possible distributions of units across machines. The *granularity* of the allele set dictates the number of allele-set values that can be used, which in turn directly affects the set of possible fractions that can result from the first transformation. Because the set of possible distributions of units across all machines is limited, so too is the design space. The use of a limited design space is undesirable but the effects are offset by the reduced computation time for the algorithm.

The second issue concerning the choice of *granularity* level for the model arises from the fact that the allele-set values are mapped into cavity-run combinations in the third
transformation. The set of possible values in the second level of depth in the genome is dictated by the granularity level. Depending on the number of units to be produced, derived from the second transformation, there exists a corresponding set of cavity-run combinations. The number of combinations can be larger than, smaller than, or equal to the number of allele-set values. When the number of combinations does not equal the number of allele-set values, there is no longer a 1-to-1 mapping between the genome and the solution space. When the number of cavity-run combination is less than the number of allele-set values, some of the representations become redundant, i.e., more than one genome will represent the same solution. This redundancy leads to some wasted computational resources, but does not affect the completeness of the solution space. In contrast, when the number of cavity-run combinations is larger than the number of allele set values, the completeness of the genomic representation is compromised. For example, Figure 19 shows the cavity-run combinations that result using each of the allele-set values from 0 to 5, as determined by the level of granularity. The example assumes that 100 units are to be produced and therefore 9 combinations are possible (combination numbers 0-8). The number of combinations is larger then the number of allele-set values and therefore, some of the combinations cannot be represented by the genome.

In Figure 19, combination numbers 1, 4, and 7 cannot be reached using any of the allele-set values. Since these combinations comprise a region of the solution space, the genome does not satisfy the completeness property. This problem can be addressed by using a granularity level with a value that is equal to the maximum number of cavity-run combinations of any element in the genome. Increasing the size of the allele set, however, leads to an increase in complexity of the problem and ultimately leads to prohibitive run
times for the algorithm. Preliminary runs of the algorithms used larger allele sets with granularities up to 11, i.e. values from 0 to 10. The larger allele sets had a marginal increase in accuracy and led to excessive run times when 50+ trials where run in the design of experiments (Chapter 7). For this reason, a granularity value of 6 was used. The understanding is that, although incomplete, the accessible regions of the solution space were scattered evenly across the complete solution space.

<table>
<thead>
<tr>
<th>Allele-set value</th>
<th>Transformation 3 calculation</th>
<th>Combination number</th>
<th>Cavities Runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>round([{9/5}^*{9-1}])</td>
<td>0</td>
<td>1 100</td>
</tr>
<tr>
<td>1</td>
<td>round([{1/5}^*{9-1}])</td>
<td>2</td>
<td>4 25</td>
</tr>
<tr>
<td>2</td>
<td>round([{2/5}^*{9-1}])</td>
<td>3</td>
<td>5 20</td>
</tr>
<tr>
<td>3</td>
<td>round([{3/5}^*{9-1}])</td>
<td>5</td>
<td>20 5</td>
</tr>
<tr>
<td>4</td>
<td>round([{4/5}^*{9-1}])</td>
<td>6</td>
<td>25 4</td>
</tr>
<tr>
<td>5</td>
<td>round([{5/5}^*{9-1}])</td>
<td>8</td>
<td>100 1</td>
</tr>
</tbody>
</table>

Figure 19 – Cavity-run combinations for each allele set value

5.3. Population Generation

The initialization function used for the EP algorithm generates 30 3D-array-allele genomes, as dictated by the ‘population size’ parameter in the design parameter vector. Each element in the array is filled with a value selected randomly from the allele set.
5.4. Fitness Evaluation

The fitness function used evaluates the cost of the system including the mold, processing, machine, and rejects costs. The scope of these costs was introduced in Section 4.5. The following sections describe the model-specific details of each of the cost components. Depending on the array-element values of the genome, the traditional system could consist of 1 to n machines (where n = the number of part types). Because the cost calculation for each machine is the same, the fitness evaluation function was designed to calculate the total cost of a single machine. This value was then multiplied by the number of machines specified by the genome.

5.4.1. Mold Cost

To calculate the mold cost of a particular machine, the cost of the molds used for each part type was summed. In Section 4.5.1, Equation 3 was used to calculate the multi-cavity mold cost for each part type built by the machine in question. Equation 8 now sums the multi-cavity mold costs across all part types and machines to generate the total mold cost for the traditional system.

\[ C_{\text{total mold}} = \sum_{i=1}^{\# \text{machines}} \sum_{j=1}^{\# \text{parts produced}} (C_{\text{multi-cavity mold}})_j \]  

(Eqn. 8)

Equation 8 – Traditional system - total mold cost

5.4.2. Processing Cost

The daily processing cost calculation used Equation 5. The formula is shown again below for clarity.
Chapter 5: Traditional-System Design Function

\[ C_{\text{daily processing cost}} = (K_1 + K_2F)(\text{available time per day}) \]

(Eqn. 5)

\[ K_1, K_2 = \text{machine rate coefficients} \]

\[ F = \text{clamp force (kN)} \]

The components of this equation were determined as follows. The clamp force value was calculated based on the mold requiring the most pressure. The machine rate coefficients were based on the current machine rate data.

The available time per day was determined by assuming a 3 shift operation, totaling 24 hours per day, minus the mold changeover times of 1.5 hours each. These values were used as a fair representation of the operating and changeover times used in industry (Toronto Plastics Inc., personal communication, April 2001).

The required number of changeover times equaled one less than the number of part types handled by the machine, since a changeover was required between the productions of the different part types. The total-annual-system-processing cost involved summing the daily processing costs of the machines used and then multiplying this value by the assumed 250 days of production per year, as shown in Equation 9.

\[ C_{\text{total annual processing cost}} = \sum_{i=1}^{\# \text{machines used}} (C_{\text{daily processing cost}} * 250) \]

(Eqn. 9)

Equation 9 – Traditional Machine – total annual processing cost

5.4.3. Machine Cost

In the traditional-system design function, the total machine cost is a summation of machine costs of all the machines used. The value of $260000 \(^2\) was used as the cost of each

\(^2\) All dollar values are in Canadian funds
5.4.4. Reject Costs

The reject costs are the artificial costs assigned to infeasible genomes to ensure that they have a lower chance of being selected for propagation. Three conditions exist that, if not met, result in a large reject cost ($1e15) being assigned to the genome.

5.4.4.1. Reject Condition 1

The first condition ensures that the clamping force required by the system, based on the size and complexity of the molds, does not exceed the upper limit of 8500 kN. This limit was based on the data (maximum clamp stroke, driving power, etc) for injection-molding machines available in Boothroyd, et al., 1994.

\[(\text{Maximum Clamp Force}) > 8500 \text{ kN} \quad \text{(Eqn. 10)}\]

Equation 10 – Traditional system - Reject condition 1

5.4.4.2. Reject Condition 2

The second condition arises as a result of the machine being unable to meet the customer demand rate due to the production-volume requirements and/or processing time constraint caused by part geometry. The total processing time of all the units produced by a machine are calculated by multiplying the cycle time of each part type by the number of runs required to produce the number of units specified by the genome. These times are then summed across all part types to find the total processing time for all units on the machine. This time is compared with the available time for production, which is based on the number of parts run on the machine and the respective number of changeover times. The available
time per day is equal to 24 hours less 1.5 hours for each mold changeover. This condition makes certain that the total processing time of all part types produced by a machine can be built in the total available time.

\[
\sum_{i=1}^{p} (\text{cycle time}) (\# \text{ runs required}) > \text{Total Available Time}
\]

where:

\[p = \# \text{ part types produced on a machine}\]

\[\text{Total Available Time} = 24 - (\# \text{ changeover times})(1.5) \text{ hr}\]

Equation 11 – Traditional system – Reject condition 2

5.4.4.3. Reject Condition 3

The third reject condition is a check of the genome’s ability to meet the customer requirements. The potential problem lies in the fact that the random initialization of the genome and the random nature of the evolutionary operator can generate a genome that will not produce any units of one or more part types. The situation arises when the first level of depth in the 3D array contains a value of 0 for all elements in a row. This implies that for a specific part type, none of the units will be built by any of the machines. To avoid this infeasible genome, a check is performed on all genomes where the values in the first level of depth in the array are summed for each part type (summed across each row). The reject cost is assigned if any of these summations result in a value of zero.

\[
\text{For all part types } \sum_{j=1}^{n} (3D \text{ array genome element})_{i,j,o} = 0
\]

where: \(n = \# \text{ machines}\)

Equation 12 – Traditional system – Reject condition 3
5.5. Selection Process

A Roulette-Wheel selection scheme was used to help ensure that the genome with the highest fitness level, i.e. lowest cost, had the greatest chance of survival. This type of selection scheme was described in Section 4.6.

5.6. Evolutionary Operator

5.6.1. Flip mutator

The evolutionary operator used to generate a new population of 3D-array-allele genomes is called the 'flip' mutator. This mutator randomly picks elements in the 3D array and sets the element in the array to any of the values specified by the allele set. The number of elements visited and flipped is based on the specified mutation probability.

5.7. Algorithm Verification

To ensure that the algorithm operated as intended, the results were verified through manual calculations. The verification ensured that the final cost satisfied all the optimization constraints and that the cost was derived from the design variables and parameters specified. In addition, the output data from numerous runs were reviewed to ensure that the costs converged over time.
6. New-System Design Function

6.1. Introduction

The new-system design function uses a customized Evolutionary Programming algorithm that begins with the production volumes for the various part types. Before the EP algorithm is built and run, all possible frequencies that the system can run are calculated based on the production volumes. Each frequency reflects a feasible number of cycles of molds through the system. For example, a single cycle of molds through the stations would have a frequency of 1. The greatest common divisor is found for the production values and is used to generate the list of possible frequencies. The EP algorithm runs a separate time for each of the frequencies, finding the respective close-to-optimal solution. The resulting solutions for the different frequencies are then compared and the solution with the lowest cost is selected as the final solution. During each generation, the maximum, minimum, and mean scores for the population are tracked, providing a method by which the success of the evolution can be monitored.

6.2. The Genome

The genome created must capture all the variable information of the new system, i.e., $X = \{X_m\}^T \{m | m \in I, [1,\# \text{ variables}]\}$. Because information regarding both the mold and station/buffer information must be captured, a new customized composite genome was developed. The genome consists of two parts. The first part is dynamic in size and covers mold information, while the second part has a fixed size and covers the station/buffer information.
Chapter 6: New-System Design Function

The first part of the genome captures information regarding the number of molds of each of the part types and the cavities per mold. This is achieved by storing the information for a mold in one genome element. Within each element, the part type is stored as a letter and the number of cavities for the mold is stored as a number. The number of elements reflects the total number of molds. Figure 20 shows an example of the first part of the genome, describing 6 molds for 3 part types (A, B, C) with the respective number of cavities per mold shown in each element.

![Figure 20 - First part of the composite genome](image)

The second part of the genome captures the station capacities of each of the eight stations. Each station is represented by an element containing a letter to represent the station and a number to represent the station capacity. The stations and their respective letters are shown in Table 9 below. An example of the second part of the genome is shown in Figure 21.

<table>
<thead>
<tr>
<th>Heating Station Buffer</th>
<th>Heating Station</th>
<th>Injection Station Buffer</th>
<th>Injection Station</th>
<th>Cooling Station Buffer</th>
<th>Cooling Station</th>
<th>Ejection Station Buffer</th>
<th>Ejection Station</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

Table 9 - Station and buffer identification letters
Combined, the two components form the new composite genome, shown in Figure 22.

The complete composite genome exhibits many of the desired genomic properties described in Section 4.3, such as Non-redundancy, Completeness, and Lamarckian. Since every solution can be represented by the genome, the representation fulfills the completeness criterion. Next, because the order of the elements in the first part of the genome is of no relevance, this part of the genome does have some redundancy, i.e., there is an n-to-1 mapping of representations to the solution space. This arises due to fact that two sets of identical elements with different orderings are two representations of the same solution. While being order independent increases the search load, there are benefits regarding the Lamarckian property. Remember, a genome exhibits the Lamarckian property when the variables in each genome can be interpreted independently and are not contextually dependent. Both parts of the genome are able to satisfy this criterion with the appropriate selection of evolutionary operators (see Section 6.6). The operators are designed to exploit the specific characteristics of the two genome parts. The order-independent quality of the first part and the contextual-independent quality of the second part are used to ensure that valuable genomic information is passed to future generations.
6.3. Population Generation

The initialization function used to generate the population of 30 genomes treated the two sections of the composite genome differently. For the first part, the initialization function determines the number of units of each part type to be produced, based on the frequency. The function then successively creates a node for the first part type with the number of cavities randomly selected from 1 to the production requirements for the cycle. As each node is created, the cycle production requirements are reduced to the remaining requirements. This process is continued for the same part until the production requirements are satisfied by the mold set. The same process follows for the other part types. Once the first part of the genome has been set, the order of the nodes is randomized.

For the second part of the genome, the number of nodes is set to represent each of the four stations and four buffers. Therefore, the function randomly allots a station-buffer capacity to each node from a predetermined range of 1 to 10. The selection of this range was rather arbitrary, however, the range starts the genome in a more feasible region of the search space.

6.4. Fitness Evaluation

6.4.1. Mold Cost

The cost of the multi-cavity molds are calculated using Equation 3 as described in section 4.5.1. Once each mold cost is determined, the algorithm checks for duplicate molds in terms of part type and cavity numbers. Because these molds are duplicates, there are savings regarding duplicated machining and design that may be captured across the set of molds. The cost of each set of duplicate molds, if present, is calculated using Equation 13.
Chapter 6: New-System Design Function

\[
C_{\text{duplicate mold set}} = \sum_{i} \left( C_{\text{multi-cavity mold}} \right)_{i} n_{i}^{0.7}
\]  
(Eqn. 13)

were: \( n_{i} = \) number of duplicate molds of type \( i \)

**Equation 13 – Cost calculation for duplicate molds**

Equation 13 uses 0.7 as a multi-mold index similar to the multi-cavity mold index used to account for savings in the machining of duplicate cavities (Equation 3). The mold costs for individual molds or duplicate-mold sets are then added to generate the overall mold costs for the mold-cavity combination specified by the genome. The total cost calculation is shown in Equation 14.

\[
C_{\text{overall mold cost}} = \sum_{i} (C_{\text{multi-cavity mold}})_{i} \text{ or } (C_{\text{duplicate mold set}})_{i}
\]  
(Eqn. 14)

**Equation 14 – Overall mold cost calculation**

6.4.2. Processing Cost

As with the traditional system, the daily-processing-cost calculation for the new-system design function uses Equation 5. Recall:

\[
C_{\text{daily processing cost}} = (K_{1} + K_{2}F)(\text{available time per day})
\]  
(Eqn. 5)

where:

- \( K_{1}, K_{2} = \) machine rate coefficients
- \( F = \) clamp force (kN)

The components of the formula were determined as follows. The clamp force was calculated based on the mold requiring the largest clamping pressure based on the part geometry. The machine rate coefficients \( K_{1} \) and \( K_{2} \) were, 25.013$/hr and 0.0075$/hr/kN respectively, as determined in section 4.5.2. The available time per day was set to 24 hrs to account for three 8 hr production shifts. Note that in the new-system design function, no
time is allotted to account for mold changeovers. The total annual system processing cost is
the product of the daily cost and the assumed 250 days of production per year.

6.4.3. Station Cost

The station/buffer cost is the summation of the cost of each station multiplied by the
respective station/buffer capacity. The calculation is shown in Equation 15.

\[ C_{\text{Station/Buffer}} = \left( \sum \text{Buffer capacities} \right) (\text{Cost of buffer}) + \sum \left( \text{Station capacity} \right) (\text{Cost of station}) \]  \hspace{1cm} (Eqn. 15)

Equation 15 – Station/Buffer cost calculation

The values used for the station and buffer costs were chosen based on discussions
with injection-molding facilities. Given the estimated cost vs. tonnage (t) data for a range of
injection-molding machine shown in Table 10 (Toronto Plastics Inc., personal
communication, April 2001), a linear relationship was found and shown in Figure 23. The
relationship indicated a y-intercept value of approximately $70000, which would correspond
to a 0t machine or the basic mechanical infrastructure, controls, etc. of a machine. This
$70000 value was used as a conservative estimate of the buffer station cost.

Table 10 – Estimated machine cost vs. tonnage

<table>
<thead>
<tr>
<th>Tonnage</th>
<th>Estimated Machine Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>$95000</td>
</tr>
<tr>
<td>250</td>
<td>$150000</td>
</tr>
<tr>
<td>500</td>
<td>$260000</td>
</tr>
</tbody>
</table>
Figure 23 – Plot of estimated machine cost vs. tonnage

The station costs used were determined based on the $70000 base cost plus an additional $60000 for station-specific equipment. Using these values, the total cost of the machine stations would equal $320000 or the cost of two traditional machines. It should be noted that since these values were based on estimates, best- and worst-case estimates were also tested. The best-case estimate assumes that the total cost of the four stations equals that of the traditional machine, i.e. $260000. The worst-case estimate, on the other hand, uses a value of $260000 dollars for each of the four processing stations. The details of the range in cost estimates are described in detail in Sections 7.2.1-7.2.2.

6.4.4. Reject Costs

Three conditions exist that would cause a particular mold-cavity combination specified by a genome to be infeasible. The objective function checks for these three conditions. If any are found, the genome is assigned a large reject cost equal to $1e15 which causes the genome to be rejected from the next generation.
6.4.4.1. Reject Condition 1

The first condition ensures that the clamping force required by the system, based on the size and complexity of the molds, does not exceed the upper limit of 85000 kN. This limit was based on the available data (maximum clamp stroke, driving power, etc) for injection-molding machines (Boothroyd, et al., 1994), and is expressed in Equation 16.

\[(\text{Maximum Clamp Force}) > 8500 \text{ kN}\]  \hspace{1cm} (Eqn. 16)

Equation 16 – New system - Reject condition 1

6.4.4.2. Reject condition 2

The second condition arises as a result of the machine being unable to meet the customer demand rate due to the production volume requirements and/or processing time constraint caused by part geometry. The total processing times for each of the molds are calculated. The longest resulting time forms the upper limit for the cycle time that can be achieved by the machine. The cycle-Takt Time is then calculated by dividing the total available time by the frequency of the run. If the longest processing cycle time is greater than the cycle-Takt Time required, then the machine will be unable to produce the required number of parts in the available time. In this case, a large reject cost is assigned to the genome to ensure that it is rejected during the selection step of the EP algorithm.

\[(\text{Longest Cycle Time of any mold in the genome}) > \text{Cycle Takt Time}\]  \hspace{1cm} (Eqn. 17)

Equation 17 – New system - Reject condition 2

6.4.4.3. Reject Condition 3

The third condition to be checked involves the flow of the molds through the system. The genome specifies the total number of positions available within the system (a summation
of the buffer and station capacities). If the total number of molds moving through the system is equal to or greater than the positions available, a deadlock situation occurs (Onvural, 1990). If this condition arises, then a reject cost is assigned to the genome to ensure it is rejected from the population during selection.

\[
\text{Number of molds} \geq \left( \frac{\text{Number of positions available}}{\text{in the system}} \right) \quad \text{(Eqn. 18)}
\]

**Equation 18 – New system – Reject condition 3**

### 6.5. Selection Process

A Roulette-Wheel selection scheme is used to help ensure that the genomes with the highest fitness value, i.e. lowest cost, have the greatest chance of survival.

### 6.6. Evolutionary Operators

Two operators were used to mutate the composite genomes to generate new genomes: ‘split/join’ mutator, and ‘increase/decrease’ mutator. These mutators were designed specifically for this design model and differ drastically from mutation operators traditionally used in EAs. The ‘split/join’ mutator applies to the first part of the genome, modifying mold information, while the ‘increase/decrease’ mutator applies to the last eight elements that deal with station capacities. Since both operators require only one genome to generate a new member of the population, the reproduction is considered asexual.

Both operators are applied to each genome that remains in the population after the selection process. Each element in the genome is visited but only undergoes mutation by the relevant mutator according to the mutation probability parameter.
6.6.1. Split/join Mutator

The split/join mutator is designed to change the mold information of a genome while still maintaining 100% BTS as required by the second design objective. To achieve this, the mutator modifies the number of molds and the number of cavities per mold, but ensures that the total number of cavities dedicated to a specific part type remains constant. This in turn ensures that the number of parts of each type, produced every cycle, remains constant.

The mutator affects each element of a genome with the mutation probability, and then uses a 50% ‘coin-toss’ to decide whether to try split the element into two, or join the element to the next element.

The split operation checks that the cavity number for the element is greater than one, before the element is split into two elements. The mutator randomly chooses a number between 1 and (# cavities – 1) and uses that value for the number of cavities in the first element, keeping the same part type. The second element uses the difference between the original value and the first element. The second element is simply inserted into the genome after the first. Figure 24 shows the split operation applied to the third elements of the first part of the genome.

When the join operation is applied to an element, the join operator checks to see whether the next element has the same part-type letter. If not, the operation fails and moves on to the next element. If, however, the part types match, the operator combines the two elements into one element with the same part-type letter and a cavity number equal to the sum of the cavities in the two initial elements.
Figure 25 illustrates the join operation applied to the second and third elements of the first part of the genome.

6.6.2. Increase/decrease Mutator

The increase/decrease mutator is designed to change the station-capacity information within a genome. The operator successively visits each of the last eight elements in the genome and mutates the element based on the mutation probability parameter. If applied, the mutator either increases or decreases the station capacity based on a 50% 'coin-toss'. When the increase operator is applied, the station/buffer capacity is increased by a random value between 1 and 3. An example of the increase operator being applied to the station/buffer-capacity-related elements of the genome is shown in Figure 26.
The decrease operation works in much the same way as the increase operator except the operator checks to make sure that the resulting capacity does not drop below 1. An example of the decrease operator being applied to the station buffer-capacity-related elements of the genome is shown in Figure 27.

6.7. Algorithm Operation

To ensure that the algorithm operated as intended, the results were verified through manual calculations. The verification ensured that the final cost satisfied all the optimization constraints and that the cost was derived from the design variables and parameters specified. In addition, the output data from numerous runs were reviewed to ensure that the costs converged over time.
7. Results

7.1. Introduction

The previous sections described in detail the design of evaluation tools used to assess the feasibility of the newly proposed injection-molding system as compared to the traditional system. With these design models fully established, attention is turned to the generation and analysis of data and a discussion of the results. The generation of data followed a full factorial experimental design. The design dictated the parameter settings for a series of trial runs across a variety of production scenarios. The analysis focused on highlighting the relative performance of the two systems across the different trials. Therefore, the analysis directly assessed the cost differential between the traditional and new systems. In addition, an analysis of variance was used to highlight significant factors affecting the cost differential.

It should be noted, that these results are limited in their applicability to industry due to the large range in production requirements used for the initial cost comparison between the two systems. A large range was selected to provide a survey of the production space, to be used as a map for further research where specific areas are examined in more detail. The results provide the impetus for more exhaustive system comparisons.

7.2. Design of Experiments

A full factorial design of experiments was used to evaluate the cost of the two systems under different system conditions and to investigate the effect of a number of factors on both systems. Each of these sets of screening experiments was then performed across three scenarios. A comparison of the traditional system was made with each of three versions of the new system – best, worst, and nominal cases. These three versions of the new
system represent a spectrum of parameter settings aimed at alleviating the design model's
dependence on assumptions regarding processing and station costs.

The need to explore this range derives from the fact that this feasibility study is being
conducted during such an early stage of the design process. Because accurate estimates of
station and processing costs are not available during the conceptual stage of design, the use
of scenarios was employed to capture the range in cost estimates based on various
assumptions. The assumptions that define the three versions of the new system and the
respective scenarios, of which they are a part, are described in the following sections.

7.2.1. Scenario 1 - Traditional vs. New system (best-case)

The new-system, best case represents the lower end of the range of possible station
and processing costs. The assumption made here is that the combined station cost of the four
processing stations is equal to the cost of the traditional machine. In addition, it is assumed
that the previous processing cost calculation adequately describes the processing costs of the
four stations. The resulting costs are shown in Table 11.

Table 11 – Cost settings for Scenario 1

<table>
<thead>
<tr>
<th></th>
<th>Station Costs (heating, injection, cooling, ejection)</th>
<th>$65 000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Costs</td>
<td>$35 000</td>
<td></td>
</tr>
<tr>
<td>Processing Costs</td>
<td>$C_{\text{daily processing cost}}</td>
<td></td>
</tr>
</tbody>
</table>

Note: $C_{\text{traditional machine}} = $260 000
      Divide by 4 stations = $65 000 per station

The station costs shown in Table 11 are equal to half the value introduced in Section
6.4.3. To remain consistent, a reduced buffer cost was used. This scenario tests the cost
differential between the traditional system and the lowest cost version of the new system.
7.2.2. Scenario 2 - Traditional vs. New system (worst-case)

While scenario 1 represents the lower end of the spectrum regarding the new system's station and processing costs, scenario 2 represents the upper end of this spectrum. In this case, it is assumed that the equipment and processing costs of each station in the new system equals that of the traditional system. In effect, each processing station is treated as a separate machine. The buffer and processing costs was adjusted to equal four times the value used in scenario 1. The resulting cost settings are shown in Table 12.

<table>
<thead>
<tr>
<th>Table 12 – Cost settings for Scenario 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Station Costs (heating, injection, cooling, ejection)</td>
</tr>
<tr>
<td>Buffer Costs</td>
</tr>
<tr>
<td>Processing Costs</td>
</tr>
</tbody>
</table>

The cost comparison in scenario 2 between the traditional and new-system, worst case is highly conservative. In an attempt to capture values representing a more reasonable and realistic region of the spectrum, the nominal case is included.

7.2.3. Scenario 3 - Traditional vs. New system (nominal case)

The nominal case uses the cost settings introduced in Section 6.4.3. As mentioned previously, the station cost values were estimated using data gathered during discussions with a local injection-molding facility (Toronto Plastics Inc., personal communication, April 2001). These values fall in between the values used in scenarios 1 and 2. Similarly, the processing cost estimate used in this scenario lies between the two extremes, i.e., 1 and 4 times the traditional-system processing cost. These values are more likely to be more...
accurate since they fall between the best- and worst-case cost limits. The cost settings for scenario 3 are shown in Table 13.

### Table 13 – Cost settings for Scenario 2

| Station Costs (heating, injection, cooling, ejection) | $130 000 |
| Buffer Costs                                      | $70 000 |
| Processing Costs                                   | \(2 \times C_{\text{daily processing cost}}\) |

Note:  
- \(C_{\text{traditional machine}} = $260 000\)  
- Divide by 2  
- \(= $130 000\) per station

The following sections describe the key elements of the design of experiments, including the selection of factors and levels, the experimental design and part shape used.

#### 7.2.4. Selection of Factors and Levels

The four factors used in the design of experiments and their respective levels are shown in Table 14.

### Table 14 - Four factors and their respective levels

<table>
<thead>
<tr>
<th>Factors</th>
<th>Part Wall Thickness</th>
<th>Part Projected Area</th>
<th>Production Volumes</th>
<th>Number of Parts*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 mm</td>
<td>4 cm(^2)</td>
<td>1000</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1 mm</td>
<td>4 cm(^2)</td>
<td>1000</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1 mm</td>
<td>100 cm(^2)</td>
<td>10000</td>
<td>8</td>
</tr>
</tbody>
</table>

* The number of parts is a two-level factor.

The factors were chosen to best realize the wide range of production requirements that may be run on each of the machines. Each factor's effect was assessed using the objective function of each design model, which provided some insight into the cost
components most likely affected. Each of the factors and the respective choice of levels are discussed below.

7.2.4.1. **Factor A - Wall Thickness**

A part's wall thickness has a large effect on the cooling time of the part, which depending on the part thickness can make up a large portion of the total processing time. This in turn affects the systems' abilities to meet the required Takt times. Three levels were selected for this factor. The first level describes the case where all part types have a thickness of 1mm. This lower value reflects the lower limit of thickness that can be handled by traditional machines (Dixon, J. R., Poli, C., 1995). Similarly, the third level describes the case where all part types have a thickness of 9mm. Again, this 9mm value forms the upper limit of what is traditionally feasible. The second level describes the case where there is considerable variety in the part thickness across all part types. Here, half the part types have a thickness of 1mm and the other half, 9mm.

7.2.4.2. **Factor B - Projected Area**

The projected area of a part is the area created by projecting the part shape onto the parting plane of the mold. This area affects the size of the mold required and, therefore, the mold cost. In addition, the projected area affects the size of the clamping unit required to accommodate the mold. This, in turn, influences the processing costs. As with the first factor, three levels were selected covering an upper and lower limit and a variety of projected areas across the part types. A lower limit of 4cm² reflects the smallest part area that can accommodate the above range in wall thicknesses. The upper limit of 100cm² is large
enough that a single additional part incurs an additional cost because it cannot be built using the smallest machine size (300kN).

7.2.4.3. **Factor C – Production Volume**

The production volumes of the various part types affect many of the components of the system cost. The production volumes place additional pressure on the systems’ abilities to build parts to meet the demand rates. Three levels were used. The 1000 units per day lower limit and the 10000 units per day upper limit reflect a realistic range of production requirements (Toronto Plastics Inc., personal communication, April 2001). The case of running a variety of high and low volume parts together is covered by the second level (1000/10000 mix).

7.2.4.4. **Factor D – Part Variety**

As with the production volumes, the number of part types required affects the total cost of the system and its ability to meet the production requirements. This factor could take on two levels, 2 or 8 different part types. These values reflect a reasonable test range for the number of part types to be produced at the same time (Toronto Plastics Inc., personal communication, April 2001).

**Table 15 – Table showing the factor-level settings of the 54 trials**

<table>
<thead>
<tr>
<th>Level</th>
<th>Factor C</th>
<th>Factor D</th>
<th>Factor E</th>
<th>Factor F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1000</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1000/10000</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1000/10000</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>10000</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>10000</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4/100</td>
<td>1000</td>
<td>2</td>
</tr>
</tbody>
</table>
In all cases, the levels were chosen to represent a realistic range of part sets manufactured using injection molding. Furthermore, while the number of levels was selected to cover a sufficiently large region of operation, care was taken to minimize the number of

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>4 / 100</th>
<th>1000</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4 / 100</td>
<td>1000 /10000</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4 / 100</td>
<td>1000 /10000</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4 / 100</td>
<td>10000</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4 / 100</td>
<td>10000</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>1000</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>10000</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>10000</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1 / 9</td>
<td>4</td>
<td>1000</td>
<td>2</td>
</tr>
<tr>
<td>1 / 9</td>
<td>4</td>
<td>1000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4</td>
<td>1000 /10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4</td>
<td>1000 /10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4</td>
<td>10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4</td>
<td>10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>1000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>1000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>1000 /10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>1000 /10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>1000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>1000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>1000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>4 / 100</td>
<td>10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>100</td>
<td>1000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>100</td>
<td>1000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>100</td>
<td>1000 /10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>100</td>
<td>1000 /10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>100</td>
<td>10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1 / 9</td>
<td>100</td>
<td>10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>1000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>1000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>1000 /10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>1000 /10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4 / 100</td>
<td>1000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4 / 100</td>
<td>1000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4 / 100</td>
<td>1000 /10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4 / 100</td>
<td>1000 /10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4 / 100</td>
<td>10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4 / 100</td>
<td>10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>1000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>1000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>1000 /10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>1000 /10000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>10000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>10000</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
levels. A larger number of factor levels multiplicatively increases the number of trials to be run for each scenario and in turn incurs added computational time.

7.2.5. Part Shape

The shape of the part used in the each of the trials is shown in Figure 28. The part shape was chosen to represent a low complexity part using the complexity classification system used by Boothroyd et al. (1994). The thickness, depth, and projected area, are all specified by the design parameter vector of the trial in question.

While the thickness and depth of the part are directly specified for the particular trial, the length and width are not. In this case, the projected area is used to generate these part dimensions. For this part, the projected area is simply the length multiplied by the width. Because the projected area is square, the length and width dimensions equal each other. Therefore, these dimensions can be readily determined by taking the square root of the projected-area value.

![Figure 28 - Parts shape](image)

7.2.6. Data Generation

The generation of data for each of the three scenarios consisted of three steps including: generating the data, creating minimum data sets, and calculating the cost differentials. Figure 29 shows the three-step data collection process.
7.2.6.1. Generating the data

The generation of data in the first step involved running the Evolutionary Program for the traditional model and each of the three versions of the new model (best, nominal and worst cases). A full factorial design was chosen to allow the examination of the effects of the four factors, introduced in Section 7.2.2, on the three scenarios, i.e., the comparisons of the traditional system with the three versions of the new system. This design consists of 54 trials (3x3x3x2) representing the complete set of combinations of factor-levels possible.

7.2.6.2. Creating the minimum data sets

During the execution of each design model evaluation, the respective EP was run 3 times with the same parameter vector, generating three sets of output data. A minimum-data
set was then created using the lowest total costs for each of the 54 trials across the three runs. This practice of generating multiple sets of data was used to help offset the non-exhaustive nature of the optimization algorithms. Although the 3 data sets generated were almost identical, there were a number of trials where one of the data sets displayed a sub-optimal solution. The use of the best values from the three sets of data helped to eliminate these sub-optimal solutions from the resulting minimum data set. The creation of the minimum data set forms the second step in data generation.

7.2.6.3. Calculating the cost differentials

The third step, calculation of the cost differentials, consisted of subtracting the new-system model minimum-data-set values for each trial from the respective trial value in the traditional-system model minimum-data set. This is performed for the three versions of the new system models thereby generating the final data sets for each scenario.

7.2.6.4. A note on single-replicate data sets

While three data sets were used to generate the minimum data set for each scenario, they do not reflect multiple replicates of data. Experiments designed to test physical processes often involve the collection of multiple replicates of data in an attempt to capture the random process variability. Because the data in this experiment is generated using optimization algorithms, there is no random variability. The three runs in this experiment were not intended to capture random variation in the system; rather they were used to offset the non-optimal nature of EAs. The minimum data sets for each scenario have only a single cost for each trial, i.e., the minimum data set represents a single replicate. Single-replicate experiments are treated specially during analysis. Section 7.5 covers this topic in detail.
Chapter 7: Results

7.3. Data

The raw data generated during the first and second steps in Section 7.2.6 are shown in Appendix III. Table 16 below, shows the cost differential data for the three scenarios.

<table>
<thead>
<tr>
<th>Trial #</th>
<th>Scenario 1 $C_{Traditional} - C_{Best Case}$</th>
<th>Scenario 2 $C_{Traditional} - C_{Worst Case}$</th>
<th>Scenario 3 $C_{Traditional} - C_{Nominal Case}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-36931.7</td>
<td>-845843</td>
<td>-306569</td>
</tr>
<tr>
<td>2</td>
<td>19774</td>
<td>-808922</td>
<td>-251182</td>
</tr>
<tr>
<td>3</td>
<td>-37585.4</td>
<td>-846909</td>
<td>-307635</td>
</tr>
<tr>
<td>4</td>
<td>171464.3</td>
<td>-684403</td>
<td>-124522</td>
</tr>
<tr>
<td>5</td>
<td>-32111.2</td>
<td>-837693</td>
<td>-298419</td>
</tr>
<tr>
<td>6</td>
<td>1E+15</td>
<td>1E+15</td>
<td>1E+15</td>
</tr>
<tr>
<td>7</td>
<td>-36931.6</td>
<td>-913343</td>
<td>-329069</td>
</tr>
<tr>
<td>8</td>
<td>1E+15</td>
<td>1E+15</td>
<td>1E+15</td>
</tr>
<tr>
<td>9</td>
<td>172743.4</td>
<td>-722335</td>
<td>-129527</td>
</tr>
<tr>
<td>10</td>
<td>1E+15</td>
<td>1E+15</td>
<td>1E+15</td>
</tr>
<tr>
<td>11</td>
<td>324102.7</td>
<td>-552308</td>
<td>31965.66</td>
</tr>
<tr>
<td>12</td>
<td>1E+15</td>
<td>1E+15</td>
<td>1E+15</td>
</tr>
<tr>
<td>13</td>
<td>-36931.7</td>
<td>-913343</td>
<td>-329069</td>
</tr>
<tr>
<td>14</td>
<td>1E+15</td>
<td>1E+15</td>
<td>1E+15</td>
</tr>
<tr>
<td>15</td>
<td>296068.3</td>
<td>-623188</td>
<td>-57520.6</td>
</tr>
<tr>
<td>16</td>
<td>1E+15</td>
<td>1E+15</td>
<td>1E+15</td>
</tr>
<tr>
<td>17</td>
<td>330486.9</td>
<td>-549180</td>
<td>38349.88</td>
</tr>
<tr>
<td>18</td>
<td>1E+15</td>
<td>1E+15</td>
<td>1E+15</td>
</tr>
<tr>
<td>19</td>
<td>-34813.6</td>
<td>-843725</td>
<td>-305719</td>
</tr>
<tr>
<td>20</td>
<td>310695.6</td>
<td>-525914</td>
<td>31825.74</td>
</tr>
<tr>
<td>21</td>
<td>9029.728</td>
<td>-800141</td>
<td>-266089</td>
</tr>
<tr>
<td>22</td>
<td>1E+15</td>
<td>1E+15</td>
<td>1E+15</td>
</tr>
<tr>
<td>23</td>
<td>3656.717</td>
<td>-795313</td>
<td>-277803</td>
</tr>
<tr>
<td>24</td>
<td>1E+15</td>
<td>1E+15</td>
<td>1E+15</td>
</tr>
<tr>
<td>25</td>
<td>157504.6</td>
<td>-721169</td>
<td>-134632</td>
</tr>
</tbody>
</table>
Chapter 7: Results

These data are used as the basis for a discussion of the results and analysis to follow.
7.4. **Analysis & Results**

7.4.1. **General Observations**

Before conducting a formal analysis of the data, some general observations of the data are discussed. The 54 trials for each scenario can be divided into three distinct categories:

- **Category 1.** Trials where the traditional system outperformed the new system
- **Category 2.** Trials where no feasible configuration of the traditional system could be used to satisfy the optimization constraints
- **Category 3.** Trials where a feasible traditional system configuration was found but still cost more than the new system configuration

Since a feasible system configuration was found for all 54 trials using the new system, trials in category 2 can be included with category 3 as cases where the new system outperformed the traditional system. Figure 30, shows the percentage make-up of the three categories of trials for each scenario’s data set.

![Figure 30 – Categorization of scenario data with regard to system performance](image-url)
7.4.1.1. Category 1 observations - Traditional system outperformed the new system

As expected, scenario 2, the comparison of the traditional system to the new-system worst-case version, exhibits the most trials where the traditional system outperforms the new system, i.e., category 1 (43%). Scenario 3 (Trad. system vs. new-system nominal case) follows with 33% and scenario 1 (Trad. system vs. new-system best-case) with 13%.

It is important to note that the seven trials that make up the 13% in scenario 1 occur in all scenarios, i.e., in these seven trials the traditional system always results in a lower cost. An examination of these trials showed that all were cases where 3 of the 4 factors were at their lowest levels. This indicates that the traditional system tends to perform better when the system requirements are at the lower end of the spectrum. Furthermore, all cases where the new system has a lower cost for scenario 1, but higher cost for scenarios 2 and 3, are cases where two of the four factors are at their lowest levels. This observation provides further evidence that the traditional system's strength lies in the trials where the factor levels are low and the production demands on the system are smallest.

7.4.1.2. Category 2 observations - No feasible configuration of the traditional system

In the second category, where the traditional system is incapable of meeting the optimization constraints, the cost differential appears as a value of $1E15 or greater. In these cases, the traditional system cost has a reject cost component, which makes its cost much greater than the new-system costs. As shown in Figure 30, all three scenarios have 31 trials (57%) that fit category 2. Most of these trials can be described as having factor-level settings of 3 medium and 1 low-level, or higher. This segment of trials represents the upper end of the spectrum in terms of production requirements providing important evidence to support the feasibility and utility of the new injection-molding system.
7.4.1.3. **Category 3 observations - New system outperformed the traditional system**

Additional support for the new system is provided by the trials in the third category where the new system has a lower cost than a feasible configuration of the traditional system. The percent of trials in this category differs across the three scenarios, as indicated in Figure 30. Of note is the fact that scenario 2 has no trials in this category. This implies that the new-system worst case only outperforms the traditional system when the traditional system cannot handle the production requirements. Scenarios 1 and 3, respectively exhibit 30% and 9% of category 3 trials, where the new system can satisfy the production requirements at a lower cost than the traditional system.

Overall, the cost differential data for the three scenarios indicate that the new system outperforms the traditional system in most trials, i.e., all category 2 and 3 trials. The data indicates an upper limit of 87% (57% + 30%) for scenario 1, a lower limit of 57% (57% + 0%) for scenario 2, and a midpoint of 66% (57% + 9%) for scenario 3.

7.4.1.4. **Effect of different levels on the data**

Another important observation was made regarding the cost differentials between the different levels of each factor. This effect was measured using a plot of the number of category 2 trials for each factor-level setting as shown in Figure 31 below.
Figure 31 – Number of category 2 trials for each factor-level setting

The data for this plot was computed by selectively sorting the trials into the three levels of the part-thickness, projected-area, and production-volume factors. The number of category 2 trials, i.e., trials with cost differentials equaling $1E15 or higher, were then counted for each level of each factor. As indicated by the Figure 31, the number of category 2 trials increases from the ‘low’ level setting to the ‘mixed low/high’ setting for the three factors tested. Furthermore, the number of category 2 trials for factors, part thickness and projected area, remain the same as the levels are changed from ‘mixed low and high’ to ‘high’. The production-volume factor exhibits a similar trend, however, there is a small increase between the ‘mixed low/high’ and ‘high’ levels. This increase indicates a trial where the change in level of the production-volume factor, from ‘mixed low/high’ to ‘high’, pushes the traditional system beyond its limit of capability.

These observations make sense, since the two factors, part thickness and projected area, affect the mold and equipment costs. These costs cover system components that need to accommodate the upper limit of the factor levels, i.e. maximum part thicknesses and
projected areas. With these factors, the effects of ‘mixed low/high’ and ‘high’ levels on the mold and equipment costs are effectively the same. Production volume, on the other hand, only affects the time required to build the parts. In this case, the ‘mixed low/high’ level requires fewer parts to be built, and therefore, places a lower demand on the system, than does the ‘high’ level.

The following section provides a detailed statistical analysis to determine the effect of each factor and the interactions thereof, on the cost differential of each scenario.

7.4.2. Analysis of Variance

7.4.2.1. Introduction

Having obtained a set of data that describes the differential system response to a group of factors, attention is now turned to determining whether any of the treatments, i.e. factors or interactions thereof, significantly affects the response. An Analysis of Variance (ANOVA) was used to compare two models of fit for the data, the single mean model and the model of separate treatment groups. The first model assumes that all treatments have the same mean \( \mu \), while the second model assumes that each treatment has its own mean response \( \mu_i \). This treatment mean \( \mu_i \) is often expressed as \( \mu_i = \mu^* + \alpha_i \), where \( \mu^* \) is called the ‘overall mean’, and \( \alpha_i \) is called the ‘ith treatment effect’. Using this formulation, the single mean model is a special case where the \( \alpha_i \) values are all equal to zero. The two models are shown in Equation 19 as they relate to the four factors in the design of experiments. Each of the factors A, B, C, and D correspond to the terms \( \alpha, \beta, \gamma, \) and \( \delta \), respectively.
Chapter 7: Results

Equation 19 - Single and separate mean models of the data

Note that the single-mean model is a special case of the separate-means model, where all the treatment effects equal each other and equal zero.

The ANOVA is based on a model of the trial response, which includes the overall mean, treatment effects, and response error. The response model is related to the model of separate treatment means, in that mean model takes the average of responses over $n$ replicates. Equation 20 shows this relationship.

\[
\mu_{ijkl} = \frac{\sum_{m=1}^{a} Y_{ijklm}}{n}
\]  
(Eqn. 20)

Equation 20 - Definition of the treatment mean

The model used for the 4-factor design of experiments is shown in Equation 21.
\[ Y_{ijkl} = \mu + \alpha_i + \beta_j + \gamma_k + \delta_l + \varepsilon_{ijkl} \]

(overall mean and main effects for the four factors, A, B, C, D)

+ \alpha\beta_{ij} + \alpha\gamma_{ik} + \alpha\delta_{il} + \beta\gamma_{jk} + \beta\delta_{jl} + \gamma\delta_{kl} \quad \text{(two-factor interactions)}

+ \alpha\beta_{ijl} + \alpha\beta\delta_{ijkl} + \alpha\gamma\delta_{ijkl} + \beta\gamma\delta_{ijkl} \quad \text{(three-factor interactions)}

+ \alpha\beta\gamma\delta_{ijkl} \quad \text{(four-factor interactions)}

(Eqn. 21)

(error)

where:

\[ i = 1 \text{ to } a, \quad j = 1 \text{ to } b, \quad k = 1 \text{ to } c, \quad l = 1 \text{ to } d, \]

\[ \varepsilon_{ijkl} \] are assumed NID(0, \sigma^2)

**Equation 21 – Four factor response model**

The ANOVA results in the determination of an F-statistic for each treatment, which tests the null hypothesis that the treatment in question has no effect on the response. If rejected, the alternate hypothesis is accepted, and the treatment is considered to have a significant effect. Effectively, the F-statistic is used to test whether a given term is needed in the response model for the data.

The accuracy of the inferences made using ANOVA depends on the validity of the assumptions used. Three important assumptions were made regarding the error in the response model, independence, normal distribution, and constant variance. The validity of these assumptions is discussed in Section 7.4.2.4. First, however, a brief review of the mechanics of the ANOVA process is covered in Section 7.4.2.2, as well as a discussion on the special treatment of single replicate experiments in Section 7.4.2.3.

**7.4.2.2. ANOVA Table**

ANOVA computations are summarized in a table form with columns for source of variation, degrees of freedom, sum of squares, mean squares, and F-statistics. There is a row
in the table for each treatment and error (sometimes called the between- and within-groups variation). A sample is shown below in Table 17.

Table 17 – Sample ANOVA table

<table>
<thead>
<tr>
<th>Source</th>
<th>Degree of Freedom</th>
<th>Sum of Squares</th>
<th>Mean Squares</th>
<th>F-statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Treatments</td>
<td>g-1</td>
<td>SS\text{Trt}</td>
<td>MS\text{Trt}= SS\text{Trt} / (g-1)</td>
<td>MS\text{Trt} / MS\text{E}</td>
</tr>
<tr>
<td>Error</td>
<td>N-g</td>
<td>SS\text{E}</td>
<td>MS\text{E}= SS\text{E} / (N-g)</td>
<td></td>
</tr>
</tbody>
</table>

Each term in the response model is tested using the respective F-statistics, i.e. a ratio of the MS\text{Trt}/MSE. The F-statistic is used to test the null hypothesis that the treatment means are the same (i.e. all the $a_i$ values are zero) versus the alternative that some of the treatment means differ (i.e. some of the $a_i$ values are nonzero). When the null hypothesis is true then the F-statistic follows the F-distribution with g-1 and N-g degrees of freedom. (The two degrees of freedom for the F-distribution come from the numerator and denominator). The computed F-statistic and degree of freedom for a treatment, is then used to find the probability of observing an F-statistic (assuming the null hypothesis), as large or larger that the computed value. This probability is called the 'p-value' or the 'observed significance level' of the test. Small p-values are evidence that the null hypothesis may be incorrect with 0.05 typically used as an indication of significance.

The classification of significance can be misleading in two ways. First, p-values of 0.049 and 0.051 are very similar although traditionally the first would be considered significant and the other not. For this reason the p-values for all treatments are included in the ANOVA tables as opposed to a listing of significant treatments. Second, the significance
indicates that there is a strong chance that the null hypothesis can be rejected, i.e. that a term representing the treatment should appear in the response model. The significance, however, does not indicate the actual effect on the difference in the means caused by the treatment.

The ANOVA tables for the three scenarios are shown in Appendix IV, the results are discussed in Section 7.5.

7.4.2.3. Single Replicate Experiments

As described above, the F-statistic is used to test the null hypothesis for each treatment in the response model. Experiments with a single-replicate response pose a problem with the analysis since there are no degrees of freedom for estimating error, i.e. $N-g = 0$. This would imply that the computation of the F-statistic involves dividing by zero. A number of approaches have been suggested to overcome this problem (Dehlert, G. W., 2000). However, it is important to stress that none provides the estimate of pure experimental error that one would get from multiple replicates. If the error estimate is biased upwards, then the analysis becomes more conservative, i.e. the computed $p$-value is generally larger than the true value. This in turn results in fewer significant terms. If the error value is biased downward, then the result is a liberal analysis with too many significant terms.

The most common method for estimating the error in a single-replicate experiment is to combine one or more higher-order interaction mean squares into an estimate of error. That is, one or more interaction terms are selected and the sums of squares and degrees of freedom are summed to obtain a surrogate-error sum of squares and degree of freedom. If these interactions are null (zero) then the surrogate mean square is an unbiased estimate of error. If, however, any of the interactions chosen is not null then the error mean square tends to be a little larger. This implies that depending on whether the interactions chosen are or are not
present in the pooled error estimate, the analysis will range from valid to conservative, respectively. In each scenario, the four-factor interaction, ABCD, was used as the surrogate error term with 8 degrees of freedom.

7.4.2.4. Validity Checks

The validity of the response model used in previous sections depends on the accuracy of key assumptions regarding the nature of the error term. This section discusses the assumptions that the errors are independent, normally distributed, and have constant variance. Since the treatment errors are not actually observed, the residuals of the response model are used. The residuals are defined as shown in Equation 21, where $y_{ij}$ is the response and $\hat{\mu}_i$ is the estimated mean response.

$$r_{ij} = y_{ij} - \hat{\mu}_i$$

(Eqn. 22) Equation 22 — Definition of residuals

The residual quantities that are observed are used to make decisions concerning how well the errors meet the assumptions.

7.4.2.4.1 Assumption 1 - Independence

The first assumption, independence, is satisfied by virtue of the fact that the model responses result from optimization algorithms. The execution of each algorithm is independent of each other and, therefore, one trial cannot affect the outcome of another. Typically, the issue of independence is addressed by randomizing the trial runs. Due to the nature of the 'experiments' used in this analysis, the randomization was unnecessary and independence is still maintained. The later two assumptions are checked using routine graphical methods.
7.4.2.4.2 Assumption 2 – Normally Distributed Errors

The second assumption of normally distributed errors is checked using a normal probability plot of studentized residuals. This test is performed on the data for each of the three scenarios. The resulting plots are shown in Figure 31.

![Figure 31 - Normal probability plot of studentized residuals for scenarios 1, 2, 3](image)

**Figure 32 – Normal probability plot of studentized residuals for scenarios 1, 2, 3**

The plots are examined for a systematic deviation from linearity, which would indicate non-normality. Since each of the normal probability plots for the three scenarios exhibits data that fits a straight line, the residuals and therefore the error, are safely assumed to be normally distributed.

7.4.2.4.3 Assumption 3 – Constant Variance

Non-constant variance is assessed using a plot of studentized residuals versus the predicted or fitted values. The predicted values are defined in Equation 22, below.

\[
Fitted \ values = y_j - r_j
\]  
*(Eqn. 23)*

**Equation 23 – Definition of fitted values**

Constant variance is typically indicated by each of the treatment groups having a similar vertical spread of replicate points. Since only single replicate data are available for
the three scenarios, a general vertical spread of data is examined. The plots for the cost data of the three scenarios are shown in Figure 32.

![Figure 33 - Studentized Residuals versus predicted values for scenario 1 cost data.]

The variance appears fairly constant in the three plots, although there does seem to be a smaller variance spread with the larger predicted values. This observation was not a concern because the effect of non-constant variance on the p-value of the F-test is relatively small when \( n_i \), the number of replicates, is constant and small (in this case, all \( n_i = 1 \)).

The validity of each of the assumptions was checked and confirmed, which provides confidence in the soundness of the statistical inferences made.

### 7.5. Discussion

The ANOVA tables, shown in Appendix IV, indicate that there are five treatments in each scenario that significantly affect the cost differential between the two systems. This section discusses these results, the mechanisms involved, and the implication on future work. The latter topic includes an important comment regarding the limitations of the findings.
The five significant treatments in each scenario are the main effects A, B, C, D and the three-factor interaction BCD. Remember, these main effects correspond with the four factors: part thickness, projected area, production volume, and part variety, respectively. This result was the same for all three scenarios and, in fact, the ANOVA tables for all three scenarios were the same. This is due to the relatively high reject cost components (i.e. category 2 trials) of the treatments and the assumed estimate of error. This outcome alters the way in which the p-values can be interpreted. Although these values still reflect the effect of the treatments on the response, they now comment more on the degree to which the treatments affect the rejection of the traditional system, i.e. its ability to meet the production requirements.

7.5.1. Significant Treatments

The impact of the main effects A, B, C, and D, can be attributed to their influence on the various components of the system cost, i.e. machine cost, mold cost, station or machine cost, etc., described in Section 7.2.4. Of particular interest is the three-factor interaction BCD. In this case, the impact of the three factors together is greater than a summation of the three main effects. This result can be explained by considering the effect of each factor on the traditional and new system costs. The part variety (D) greatly affects the traditional system cost by either increasing the number of changeovers or the number of machines required. A large number of changeovers has a large impact on the available time for production, while the need for multiple machines greatly affects the total machine and processing costs. The production volume (C) increases the number of parts to be built and, therefore, has a multiplicative effect on the time required to build all the parts. The projected area (B) increases the size of the machine required, and since the size has an upper limit, the
projected area also affects the required number of runs of the mold. For a part with a large
projected area, the number of cavities must be low to ensure that the largest machine can still
handle the mold. This restriction on the number of cavities that can be used in each mold
leads to an increase in the number of runs required. This, in turn, multiplicatively increases
the amount of time required to produce the parts.

With the available time reduced by the part-variety-induced changeovers, and the
time required to build the part increased by the large production volume and projected area, it
seems plausible that factors B, C, and D interact, increasing the likelihood that the traditional
system will not meet the production requirements.

As highlighted in Section 1.4, the goal of this research was to determine the
feasibility of the new multi-station injection-molding system through a comparison with
traditional molding equipment across a variety of production requirements. Direct
observation of the results, along with an ANOVA, has indicated that there are numerous
cases where the new system outperforms the traditional system. This data provides
encouraging evidence to support further research regarding system performance and
development.

7.5.2. Limitations of the ANOVA Results

When assessing the applicability of the findings it is important to be aware of the
scope of the analytical results. Since a sensitivity analysis was not run, the data can only be
used to describe the specific cases used in the experiment, and cannot be used to determine a
the system response across a region. To achieve this, a sensitivity analysis would be required
to measure the effect of small deviations in the factor levels on the cost of the systems.
Moreover, a hyperspace contour plot must to be constructed to allow one to predict the system response given any factor-level settings.

The following chapter summarizes the steps followed during the research, from the proposed embodiment of the design, through the development of the design models, the evolutionary programming algorithms, and the collection of the results. Possibilities for future work are discussed in light of the current findings.
8. Summary and Conclusions

8.1. Introduction

The design and evaluation of the multi-station injection-molding machine followed a series of steps from problem definition through to system validation. The research began with the identification of current industry demands regarding the handling of multiple part types and volumes, as part of a more flexible production system. Traditionally, companies have geared themselves toward producing greater quantities, in less time, in order to remain financially competitive. The new demands which include the ability to handle increased product variety and smaller batch sizes formed the motivation for the new-system design. The goal, as stated previously, was to return to the conceptual design stage of the injection-molding machine, to identify system requirements based on current manufacturing demands and to develop a new system to better address those requirements.

8.2. Research Review

The proposed embodiment of the machine resulted from a functional decomposition and analysis of the injection-molding process, which highlighted key steps of the process. The resulting multi-station injection-molding system consisted of four uncoupled stations: heating, injection, cooling, and ejection. These separate stations allow each of the injection-molding-process steps to run in parallel by using multiple molds that circulate through the system.

With a newly proposed-system design in place, the research turned to the evaluation of this system compared to traditional injection-molding machines in their ability to meet the
'lean production' requirements. These systems were required to meet the customer demand rates for a variety of part types, at the lowest cost.

Design modeling provided a structured approach to the comparative evaluation of the two systems with respect to the 'lean production' criteria. Design models were built for both the traditional and new systems, and they included the identification of system inputs, outputs, objectives, and the development of design functions. The design functions for the two systems included an optimization algorithm that was used to determine the lowest-cost-system configuration that could meet the design criteria. The optimizations involved the use of Evolutionary Programming algorithms. These algorithms are computational models that use evolutionary processes as key elements in their design and implementation. The research included the development of custom representations of the input data sets, and custom evolutionary operators, for use in the respective EPs.

Once determined, the optimized system configurations were used in various cost analyses of the new machine versus the traditional equipment based on a number of factors. The system comparison was conducted using three scenarios, which represented the possible range in estimated station costs. A full factorial design of experiments was performed with three 3-level factors (A-part thickness, B-projected area, and C-production volumes) and one 1-level factor (D-part variety). Results indicate that the new system outperforms the traditional system in most cases with a lower limit of 57% of cases. Furthermore, the analysis of variance showed that for the 54 trials performed, all of the factor main effects were significant, as was the three-factor interaction between factors B, C, and D. These results support the original hypothesis that the new system would be more capable of handling multiple part types and smaller, more frequent batch sizes. The data provides
encouraging evidence to support further research regarding system performance and development.

8.3. Future Work

Additional research focused on furthering the design of a Lean-Production-based injection-molding system, can be grouped into four main groups: improving cost estimates, expanding the use of the design models, mapping of the solution space, and the development of a physical prototype.

8.3.1. Improving Cost Estimates

Additional research should focus on improving estimates of buffer and station costs in the new system. With a better understanding of the costs of these units, the need to run multiple scenarios could be eliminated, thereby reducing the computational complexity of subsequent experiments.

8.3.2. Expanding the Use of the Design Models

The development of the design models and their respective design functions has provided encouraging preliminary results. By modifying the design input variables and parameters, these design models can be used to test the effect of numerous factors on system performance. Of particular interest would be an analysis that examined the resulting effect of different part geometries on the system. Alternately, the levels of each factor can be changed to examine production requirements with a larger variety of part types. The cases covered in the initial analysis conducted here simplified the possible variations by only considering cases where all part types shared a common factor level or cases where half the part types
exhibited factor levels different from the other half. Cases where only one or two part types differ are left for future work.

As indicated previously, the scope of the results generated is limited to the specific trials evaluated. To expand the range of relevance a sensitivity analysis must be conducted. This analysis would use the design functions to test the effect of small variations in the system inputs on the system outputs.

8.3.3. Mapping the Solution Space

Ultimately, the complete solution hyperspace with all design variables as axes could be mapped. This would provide a guide by which injection molders could determine when to choose the new system over the traditional system. Based on the results gathered during this research, molders can make this decision if, and only if, their production requirements match those in any of the 54 trials. The hyperspace map would provide a more comprehensive diagnostic tool for injection molding facilities.

8.3.4. Development of a Physical Prototype

Additional exploration is needed to develop and test a physical prototype of the proposed system. The design and manufacture of the physical machine is a large undertaking, which would involve overcoming a number of technical challenges. Some of the mechanisms that need to be considered include the mold transfer system, mold temperature and clamping control, mold scheduling through the stations, etc. Due to the early stage of the study conducted here, these technical obstacles were ignored. Considerable effort would be involved in developing a prototype system, but the apparent benefits would have been well established.
8.4. Additional Considerations

While the evaluative tools were developed to address the comparison of the traditional- and new-injection-molding systems, they can be used for comparisons that are more generic. These tools can be used to assess the relative benefit of any multi-stage process versus the individual processes. For example, the traditional- and new-system design functions can be used to compare a single multi-stage-assembly station versus separate assembly stations. In the case of assembly, a trade-off exists between station and tooling costs versus the cost of labour. While the comparative evaluation of the two assembly-station setups would involve a redefinition of the design inputs, the design outputs and design functions used would be very similar. The wide range of problems that can be addressed by these evolutionary tools, dramatically improves the applicability and scope of this research.

8.5. Final Conclusions

In summary, the research presented was unique in its introduction of 'lean production' principles to the injection-molding process at the machine level. The results identified a number of cases where the new multi-station injection-molding system outperformed traditional machines in terms of overall cost and its ability to meet customer demand rates across a variety of part types. The findings may lead to additional research into the system embodiment and operation, and the design and construction of a prototype system. This research forms an important first step in the development of a new multi-station injection-molding system that can better address the increasing demands in the injection-molding industry for higher flexibility at lower cost.
9. Appendices

Appendix I – Design Parameters

<table>
<thead>
<tr>
<th>Description</th>
<th>Term used in the design model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part</td>
<td></td>
</tr>
<tr>
<td>P₁</td>
<td>No. part types</td>
</tr>
<tr>
<td>P₂</td>
<td>Production Volume i</td>
</tr>
<tr>
<td>Part Geometry</td>
<td></td>
</tr>
<tr>
<td>P₃</td>
<td>Max Wall Thickness</td>
</tr>
<tr>
<td>P₄</td>
<td>Total Projected Area i</td>
</tr>
<tr>
<td>P₅</td>
<td>Width</td>
</tr>
<tr>
<td>P₆</td>
<td>Height</td>
</tr>
<tr>
<td>P₇</td>
<td>Depth</td>
</tr>
<tr>
<td>P₈</td>
<td>Complexity *</td>
</tr>
<tr>
<td>P₉</td>
<td>Mold Base Cost i **</td>
</tr>
<tr>
<td>Part Processing Times**</td>
<td></td>
</tr>
<tr>
<td>P₁₀</td>
<td>Heating Time i</td>
</tr>
<tr>
<td>P₁₁</td>
<td>Injection Time i</td>
</tr>
<tr>
<td>P₁₂</td>
<td>Cooling Time i</td>
</tr>
<tr>
<td>P₁₃</td>
<td>Ejection Time i</td>
</tr>
</tbody>
</table>

* As defined by (Boothroyd et al., 1994) including:
  No. of Side-Pulls, No. of Internal Lifters, No. of Unscrewing Device, Surface Finish/Appearance, Tolerance, Texture, Parting Plane.

** Refers to calculated parameters. These values were not directly entered into the model but were calculated using other parameters and tracked during the design process.
| Appendix I |

<table>
<thead>
<tr>
<th>Material</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{14}$</td>
<td>Material Type $i$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{15}$</td>
<td>Buffer Cost</td>
</tr>
<tr>
<td>$p_{16}$</td>
<td>Heating Unit Cost</td>
</tr>
<tr>
<td>$p_{17}$</td>
<td>Injection Unit Cost</td>
</tr>
<tr>
<td>$p_{18}$</td>
<td>Cooling Unit Cost</td>
</tr>
<tr>
<td>$p_{19}$</td>
<td>Ejection Unit Cost</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size – Clamping Force**</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{20}$ Clamping Force</td>
<td>$F_{\text{clamp}}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processing</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{21}$</td>
<td>Processing Cost coefficient</td>
</tr>
<tr>
<td>$p_{22}$</td>
<td>Processing Cost coefficient</td>
</tr>
<tr>
<td>$p_{23}$</td>
<td>Available time per day</td>
</tr>
<tr>
<td>$p_{24}$</td>
<td>Production days per year</td>
</tr>
<tr>
<td>$p_{25}$</td>
<td>Changeover Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Amortization</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{26}$</td>
<td>Mold Amortization Period</td>
</tr>
<tr>
<td>$p_{27}$</td>
<td>Machine Amortization Period</td>
</tr>
<tr>
<td>$p_{28}$</td>
<td>Interest Rate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimization</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{29}$</td>
<td>Population size</td>
</tr>
<tr>
<td>$P_{30}$</td>
<td>Mutation probability</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$P_{31}$</td>
<td>Crossover Probability</td>
</tr>
<tr>
<td>$P_{32}$</td>
<td>Number of Generations</td>
</tr>
<tr>
<td>$P_{33}$</td>
<td>Replacement Percentage</td>
</tr>
<tr>
<td>$P_{34}$</td>
<td>Replacement Number</td>
</tr>
</tbody>
</table>
### Appendix II - Parameter Range

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>No. part types</td>
<td>(\forall i \in I, \left[1, n\right])</td>
</tr>
<tr>
<td>$P_2$</td>
<td>Production Volume $i$</td>
<td>(V_{\text{production}})</td>
</tr>
<tr>
<td>$P_3$</td>
<td>Max Wall Thickness</td>
<td>$1\ \text{mm} \leq L_{\text{thickness}} \leq 9\ \text{mm}$</td>
</tr>
<tr>
<td>$P_4$</td>
<td>Total Projected Area $i$</td>
<td>$4\ \text{cm}^2 \leq A_{\text{total projected}}$</td>
</tr>
<tr>
<td>$P_5$</td>
<td>Width</td>
<td>$2\ \text{cm} \leq L_{\text{width}}$</td>
</tr>
<tr>
<td>$P_6$</td>
<td>Height</td>
<td>$2\ \text{cm} \leq L_{\text{height}}$</td>
</tr>
<tr>
<td>$P_7$</td>
<td>Depth</td>
<td>$2\ \text{cm} \leq L_{\text{depth}}$</td>
</tr>
</tbody>
</table>

### Material

<table>
<thead>
<tr>
<th>Material Type</th>
<th>Thermal Diffusivity ((\text{mm}^2/\text{s}))</th>
<th>Inject Temp. (\text{(^\circ}\text{C}))</th>
<th>Mold Temp. (\text{(^\circ}\text{C}))</th>
<th>Eject Temp. (\text{(^\circ}\text{C}))</th>
<th>Injection Pressure (bars)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-density polyethylene (HDPE)</td>
<td>0.11</td>
<td>232</td>
<td>27</td>
<td>52</td>
<td>965</td>
</tr>
<tr>
<td>High-impact</td>
<td>0.09</td>
<td>218</td>
<td>27</td>
<td>77</td>
<td>965</td>
</tr>
<tr>
<td>Acrylonitrile-butadiene-styrene (ABS)</td>
<td>0.13</td>
<td>260</td>
<td>54</td>
<td>82</td>
<td>1000</td>
</tr>
<tr>
<td>Acetal (homopolymer)</td>
<td>0.09</td>
<td>216</td>
<td>93</td>
<td>129</td>
<td>1172</td>
</tr>
<tr>
<td>Polyamide (6/6 nylon)</td>
<td>0.10</td>
<td>291</td>
<td>91</td>
<td>129</td>
<td>1103</td>
</tr>
<tr>
<td>Polycarbonate (PC)</td>
<td>0.13</td>
<td>302</td>
<td>91</td>
<td>127</td>
<td>1172</td>
</tr>
<tr>
<td>PC (30% glass)</td>
<td>0.13</td>
<td>329</td>
<td>102</td>
<td>141</td>
<td>1310</td>
</tr>
<tr>
<td>Modified polyphenylene oxide (PPO)</td>
<td>0.12</td>
<td>232</td>
<td>82</td>
<td>102</td>
<td>1034</td>
</tr>
</tbody>
</table>
### Machine – Station

<table>
<thead>
<tr>
<th>P_{15}</th>
<th>Buffer Cost</th>
<th>35 000 \leq (C_{\text{buffer}}) \leq 140 000</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_{16}</td>
<td>Heating Unit Cost</td>
<td>65 000 \leq (C_{\text{heau}}) \leq 260 000</td>
</tr>
<tr>
<td>P_{17}</td>
<td>Injection Unit Cost</td>
<td>65 000 \leq (C_{\text{injectu}}) \leq 260 000</td>
</tr>
<tr>
<td>P_{18}</td>
<td>Cooling Unit Cost</td>
<td>65 000 \leq (C_{\text{coolu}}) \leq 260 000</td>
</tr>
<tr>
<td>P_{19}</td>
<td>Ejection Unit Cost</td>
<td>65 000 \leq (C_{\text{ejectu}}) \leq 260 000</td>
</tr>
</tbody>
</table>

### Machine – Size

<table>
<thead>
<tr>
<th>Clamping Force (KN)</th>
<th>Shot Size (cm³)</th>
<th>Operating Costs ($/hr)</th>
<th>Dry Cycle Time (s)</th>
<th>Max Clamp Stroke (cm)</th>
<th>Driving Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_{20} – F_{clamp}</td>
<td>V_{shot}</td>
<td>C_{operating}</td>
<td>T_{dry cycle}</td>
<td>L_{clamp stroke}</td>
<td>P_{w driving}</td>
</tr>
<tr>
<td>300</td>
<td>34</td>
<td>28</td>
<td>1.7</td>
<td>20</td>
<td>5.5</td>
</tr>
<tr>
<td>500</td>
<td>85</td>
<td>30</td>
<td>1.9</td>
<td>23</td>
<td>7.5</td>
</tr>
<tr>
<td>800</td>
<td>201</td>
<td>33</td>
<td>3.3</td>
<td>32</td>
<td>18.5</td>
</tr>
<tr>
<td>1100</td>
<td>286</td>
<td>36</td>
<td>3.9</td>
<td>37</td>
<td>22</td>
</tr>
<tr>
<td>1600</td>
<td>286</td>
<td>41</td>
<td>3.6</td>
<td>42</td>
<td>22</td>
</tr>
<tr>
<td>5000</td>
<td>2290</td>
<td>74</td>
<td>6.1</td>
<td>70</td>
<td>63</td>
</tr>
<tr>
<td>8500</td>
<td>3636</td>
<td>108</td>
<td>8.6</td>
<td>85</td>
<td>90</td>
</tr>
</tbody>
</table>
### Appendix II

#### Processing

<table>
<thead>
<tr>
<th>$P_{21}$</th>
<th>Processing Cost coefficient</th>
<th>(K1) = 33.80 $/hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{22}$</td>
<td>Processing Cost coefficient</td>
<td>(K2) = 0.0001 $/hr/kN</td>
</tr>
<tr>
<td>$P_{23}$</td>
<td>Available time per day</td>
<td>24 hrs</td>
</tr>
<tr>
<td>$P_{24}$</td>
<td>Production days per year</td>
<td>250</td>
</tr>
<tr>
<td>$P_{25}$</td>
<td>Changeover Time</td>
<td>1.5 hrs</td>
</tr>
</tbody>
</table>

#### Amortization

<table>
<thead>
<tr>
<th>$P_{26}$</th>
<th>Mold Amortization Period</th>
<th>3 yrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{27}$</td>
<td>Machine Amortization Period</td>
<td>5 yrs</td>
</tr>
<tr>
<td>$P_{28}$</td>
<td>Interest Rate</td>
<td>10 %</td>
</tr>
</tbody>
</table>

#### Optimization

<table>
<thead>
<tr>
<th>$P_{29}$</th>
<th>Population size</th>
<th>$2 \leq \text{population_size}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{30}$</td>
<td>Mutation probability</td>
<td>$0.00 \leq \text{mutation_probability} \leq 1.00$</td>
</tr>
<tr>
<td>$P_{31}$</td>
<td>Crossover Probability</td>
<td>$\text{crossover_probability} = 0.00$</td>
</tr>
<tr>
<td>$P_{32}$</td>
<td>Number of Generations</td>
<td>$1 \leq \text{number_of_generations}$</td>
</tr>
<tr>
<td>$P_{33}$</td>
<td>Replacement Percentage</td>
<td>$0.00 \leq #\text{replacement_percentage} \leq 1.00$</td>
</tr>
<tr>
<td>$P_{34}$</td>
<td>Replacement Number</td>
<td>$#\text{replacement_number}$</td>
</tr>
</tbody>
</table>
### Appendix III - Raw Data – (minimum value set)

<table>
<thead>
<tr>
<th>Trial #</th>
<th>Traditional</th>
<th>New Best Case</th>
<th>New Nominal Case</th>
<th>New Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>234736.84</td>
<td>271668.50</td>
<td>541305.50</td>
<td>1080579.50</td>
</tr>
<tr>
<td>2</td>
<td>314683.81</td>
<td>294909.81</td>
<td>565865.81</td>
<td>1123605.63</td>
</tr>
<tr>
<td>3</td>
<td>238811.90</td>
<td>276397.34</td>
<td>546447.38</td>
<td>1085721.38</td>
</tr>
<tr>
<td>4</td>
<td>531169.57</td>
<td>359705.31</td>
<td>655891.58</td>
<td>1215573.00</td>
</tr>
<tr>
<td>5</td>
<td>242886.95</td>
<td>274998.16</td>
<td>541305.50</td>
<td>1080579.50</td>
</tr>
<tr>
<td>6</td>
<td>1E+15</td>
<td>286995.91</td>
<td>565865.81</td>
<td>1123605.63</td>
</tr>
<tr>
<td>7</td>
<td>258828.17</td>
<td>295759.81</td>
<td>587896.81</td>
<td>1172170.75</td>
</tr>
<tr>
<td>8</td>
<td>1E+15</td>
<td>315861.22</td>
<td>617231.13</td>
<td>1219970.88</td>
</tr>
<tr>
<td>9</td>
<td>516640.60</td>
<td>343897.16</td>
<td>646167.56</td>
<td>1238975.75</td>
</tr>
<tr>
<td>10</td>
<td>1E+15</td>
<td>625925.94</td>
<td>1117239.50</td>
<td>2269334.25</td>
</tr>
<tr>
<td>11</td>
<td>619862.47</td>
<td>295759.81</td>
<td>587896.81</td>
<td>1172170.75</td>
</tr>
<tr>
<td>12</td>
<td>1E+15</td>
<td>315861.22</td>
<td>617231.13</td>
<td>1219970.88</td>
</tr>
<tr>
<td>13</td>
<td>260419.50</td>
<td>297351.16</td>
<td>589488.13</td>
<td>1173762.13</td>
</tr>
<tr>
<td>14</td>
<td>1E+15</td>
<td>322226.53</td>
<td>623596.44</td>
<td>1226336.25</td>
</tr>
<tr>
<td>15</td>
<td>617378.75</td>
<td>321310.41</td>
<td>674899.38</td>
<td>1240567.00</td>
</tr>
<tr>
<td>16</td>
<td>1E+15</td>
<td>615144.38</td>
<td>1162214.63</td>
<td>2350736.25</td>
</tr>
<tr>
<td>17</td>
<td>627838.01</td>
<td>297351.16</td>
<td>589488.13</td>
<td>1177018.38</td>
</tr>
<tr>
<td>18</td>
<td>1E+15</td>
<td>330140.44</td>
<td>623596.44</td>
<td>1226336.25</td>
</tr>
<tr>
<td>19</td>
<td>236854.85</td>
<td>271668.50</td>
<td>542574.19</td>
<td>1080579.50</td>
</tr>
<tr>
<td>20</td>
<td>597691.55</td>
<td>286995.91</td>
<td>565865.81</td>
<td>1123605.63</td>
</tr>
<tr>
<td>21</td>
<td>285427.07</td>
<td>276397.34</td>
<td>551516.38</td>
<td>1085586.38</td>
</tr>
<tr>
<td>22</td>
<td>1E+15</td>
<td>349862.34</td>
<td>639391.25</td>
<td>1246491.63</td>
</tr>
<tr>
<td>23</td>
<td>289502.12</td>
<td>285845.41</td>
<td>567304.88</td>
<td>1084815.50</td>
</tr>
<tr>
<td>24</td>
<td>1E+15</td>
<td>345160.06</td>
<td>683119.06</td>
<td>1286324.00</td>
</tr>
<tr>
<td>25</td>
<td>453264.36</td>
<td>295759.81</td>
<td>587896.81</td>
<td>1174433.25</td>
</tr>
<tr>
<td>26</td>
<td>1E+15</td>
<td>323775.13</td>
<td>617231.13</td>
<td>1219970.88</td>
</tr>
<tr>
<td>27</td>
<td>1E+15</td>
<td>339211.56</td>
<td>643821.88</td>
<td>1236630.13</td>
</tr>
<tr>
<td></td>
<td>1E+15</td>
<td>610490.69</td>
<td>1158475.38</td>
<td>2210445.25</td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>-----------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>29</td>
<td>1E+15</td>
<td>321779.84</td>
<td>650848.50</td>
<td>1254150.88</td>
</tr>
<tr>
<td>30</td>
<td>1E+15</td>
<td>415921.22</td>
<td>855784.81</td>
<td>1624717.00</td>
</tr>
<tr>
<td>31</td>
<td>454855.69</td>
<td>297351.16</td>
<td>592744.44</td>
<td>1173762.13</td>
</tr>
<tr>
<td>32</td>
<td>1E+15</td>
<td>330140.44</td>
<td>623596.44</td>
<td>1226336.25</td>
</tr>
<tr>
<td>33</td>
<td>1E+15</td>
<td>345488.50</td>
<td>693365.19</td>
<td>1275153.00</td>
</tr>
<tr>
<td>34</td>
<td>1E+15</td>
<td>623237.25</td>
<td>1193252.13</td>
<td>2182067.75</td>
</tr>
<tr>
<td>35</td>
<td>1E+15</td>
<td>341633.28</td>
<td>681563.94</td>
<td>1320961.50</td>
</tr>
<tr>
<td>36</td>
<td>1E+15</td>
<td>447675.63</td>
<td>829813.88</td>
<td>1657401.88</td>
</tr>
<tr>
<td>37</td>
<td>253109.13</td>
<td>272937.16</td>
<td>542574.19</td>
<td>1080579.50</td>
</tr>
<tr>
<td>38</td>
<td>1107922.48</td>
<td>286995.91</td>
<td>565865.81</td>
<td>1123605.63</td>
</tr>
<tr>
<td>39</td>
<td>292681.35</td>
<td>276657.44</td>
<td>545380.56</td>
<td>1084654.50</td>
</tr>
<tr>
<td>40</td>
<td>1E+15</td>
<td>343680.13</td>
<td>652430.31</td>
<td>1301955.00</td>
</tr>
<tr>
<td>41</td>
<td>537177.26</td>
<td>280885.47</td>
<td>549455.56</td>
<td>1088729.63</td>
</tr>
<tr>
<td>42</td>
<td>1E+15</td>
<td>360987.94</td>
<td>737132.94</td>
<td>1391750.25</td>
</tr>
<tr>
<td>43</td>
<td>460518.64</td>
<td>298022.28</td>
<td>587896.81</td>
<td>1209102.50</td>
</tr>
<tr>
<td>44</td>
<td>1E+15</td>
<td>315961.22</td>
<td>617231.13</td>
<td>1219970.88</td>
</tr>
<tr>
<td>45</td>
<td>1E+15</td>
<td>345171.03</td>
<td>673308.06</td>
<td>1275907.38</td>
</tr>
<tr>
<td>46</td>
<td>1E+15</td>
<td>619723.63</td>
<td>1160146.25</td>
<td>2171629.00</td>
</tr>
<tr>
<td>47</td>
<td>1E+15</td>
<td>326134.22</td>
<td>652564.88</td>
<td>1285806.63</td>
</tr>
<tr>
<td>48</td>
<td>1E+15</td>
<td>408007.28</td>
<td>792473.44</td>
<td>1561405.63</td>
</tr>
<tr>
<td>49</td>
<td>904497.23</td>
<td>300607.41</td>
<td>592744.44</td>
<td>1177018.38</td>
</tr>
<tr>
<td>50</td>
<td>1E+15</td>
<td>330140.44</td>
<td>639424.25</td>
<td>1226336.25</td>
</tr>
<tr>
<td>51</td>
<td>1E+15</td>
<td>321310.41</td>
<td>626947.38</td>
<td>1941899.50</td>
</tr>
<tr>
<td>52</td>
<td>1E+15</td>
<td>595697.00</td>
<td>1158880.13</td>
<td>2217546.50</td>
</tr>
<tr>
<td>53</td>
<td>1E+15</td>
<td>341633.28</td>
<td>665736.13</td>
<td>1345970.63</td>
</tr>
<tr>
<td>54</td>
<td>2E+15</td>
<td>426261.72</td>
<td>845641.75</td>
<td>1759207.13</td>
</tr>
</tbody>
</table>
Appendix IV – ANOVA Tables

Scenario 1

<table>
<thead>
<tr>
<th>Source</th>
<th>Sum of Squares</th>
<th>DF</th>
<th>Mean Square</th>
<th>F Value</th>
<th>Prob &gt; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>1.444E+031</td>
<td>45</td>
<td>3.210E+029</td>
<td>4.33</td>
<td>0.0168</td>
</tr>
<tr>
<td>A</td>
<td>1.148E+030</td>
<td>2</td>
<td>5.741E+029</td>
<td>7.75</td>
<td>0.0134</td>
</tr>
<tr>
<td>B</td>
<td>2.704E+030</td>
<td>2</td>
<td>1.352E+030</td>
<td>18.25</td>
<td>0.0010</td>
</tr>
<tr>
<td>C</td>
<td>1.926E+030</td>
<td>2</td>
<td>9.630E+029</td>
<td>13.00</td>
<td>0.0031</td>
</tr>
<tr>
<td>D</td>
<td>4.741E+030</td>
<td>1</td>
<td>4.741E+030</td>
<td>64.00</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>AB</td>
<td>1.852E+029</td>
<td>4</td>
<td>4.630E+028</td>
<td>0.63</td>
<td>0.6579</td>
</tr>
<tr>
<td>AC</td>
<td>6.296E+029</td>
<td>4</td>
<td>1.574E+029</td>
<td>2.13</td>
<td>0.1691</td>
</tr>
<tr>
<td>AD</td>
<td>2.593E+029</td>
<td>2</td>
<td>1.296E+029</td>
<td>1.75</td>
<td>0.2342</td>
</tr>
<tr>
<td>BC</td>
<td>7.407E+028</td>
<td>4</td>
<td>1.852E+028</td>
<td>0.25</td>
<td>0.9018</td>
</tr>
<tr>
<td>BD</td>
<td>3.704E+028</td>
<td>2</td>
<td>1.852E+028</td>
<td>0.25</td>
<td>0.7847</td>
</tr>
<tr>
<td>CD</td>
<td>1.481E+029</td>
<td>2</td>
<td>7.407E+028</td>
<td>1.00</td>
<td>0.4096</td>
</tr>
<tr>
<td>ABC</td>
<td>3.704E+029</td>
<td>8</td>
<td>4.630E+028</td>
<td>0.63</td>
<td>0.7394</td>
</tr>
<tr>
<td>ABD</td>
<td>6.296E+029</td>
<td>4</td>
<td>1.574E+029</td>
<td>2.13</td>
<td>0.1691</td>
</tr>
<tr>
<td>ACD</td>
<td>1.852E+029</td>
<td>4</td>
<td>4.630E+028</td>
<td>0.63</td>
<td>0.6579</td>
</tr>
<tr>
<td>BCD</td>
<td>1.407E+030</td>
<td>4</td>
<td>3.519E+029</td>
<td>4.75</td>
<td>0.0294</td>
</tr>
<tr>
<td>Error</td>
<td>5.926E+029</td>
<td>8</td>
<td>7.407E+028</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cor Total</td>
<td>1.504E+031</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Model F-value of 4.33 implies the model is significant. There is only a 1.68% chance that a "Model F-Value" this large could occur due to noise.

Values of "Prob > F" less than 0.0500 indicate model terms are significant. In this case A, B, C, D, BCD are significant model terms.

Values greater than 0.1000 indicate the model terms are not significant.
Scenario 2

<table>
<thead>
<tr>
<th>Source</th>
<th>Sum of Squares</th>
<th>DF</th>
<th>Mean Square</th>
<th>F Value</th>
<th>Prob &gt; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>1.444E+031</td>
<td>45</td>
<td>3.210E+029</td>
<td>4.33</td>
<td>0.0168</td>
</tr>
<tr>
<td>A</td>
<td>1.148E+030</td>
<td>2</td>
<td>5.741E+029</td>
<td>7.75</td>
<td>0.0134</td>
</tr>
<tr>
<td>B</td>
<td>2.704E+030</td>
<td>2</td>
<td>1.352E+030</td>
<td>18.25</td>
<td>0.0010</td>
</tr>
<tr>
<td>C</td>
<td>1.926E+030</td>
<td>2</td>
<td>9.630E+029</td>
<td>13.00</td>
<td>0.0031</td>
</tr>
<tr>
<td>D</td>
<td>4.741E+030</td>
<td>1</td>
<td>4.741E+030</td>
<td>64.00</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>AB</td>
<td>1.852E+029</td>
<td>4</td>
<td>4.630E+028</td>
<td>0.63</td>
<td>0.6579</td>
</tr>
<tr>
<td>AC</td>
<td>6.296E+029</td>
<td>4</td>
<td>1.574E+029</td>
<td>2.13</td>
<td>0.1691</td>
</tr>
<tr>
<td>AD</td>
<td>2.593E+029</td>
<td>2</td>
<td>1.296E+029</td>
<td>1.75</td>
<td>0.2342</td>
</tr>
<tr>
<td>BC</td>
<td>7.407E+028</td>
<td>4</td>
<td>1.852E+028</td>
<td>0.25</td>
<td>0.9018</td>
</tr>
<tr>
<td>BD</td>
<td>3.704E+028</td>
<td>2</td>
<td>1.852E+028</td>
<td>0.25</td>
<td>0.7847</td>
</tr>
<tr>
<td>CD</td>
<td>1.481E+029</td>
<td>2</td>
<td>7.407E+028</td>
<td>1.00</td>
<td>0.4096</td>
</tr>
<tr>
<td>ABC</td>
<td>3.704E+029</td>
<td>8</td>
<td>4.630E+028</td>
<td>0.63</td>
<td>0.7394</td>
</tr>
<tr>
<td>ABD</td>
<td>6.296E+029</td>
<td>4</td>
<td>1.574E+029</td>
<td>2.13</td>
<td>0.1691</td>
</tr>
<tr>
<td>ACD</td>
<td>1.852E+029</td>
<td>4</td>
<td>4.630E+028</td>
<td>0.63</td>
<td>0.6579</td>
</tr>
<tr>
<td>BCD</td>
<td>1.407E+030</td>
<td>4</td>
<td>3.519E+029</td>
<td>4.75</td>
<td>0.0294</td>
</tr>
<tr>
<td>Error</td>
<td>5.926E+029</td>
<td>8</td>
<td>7.407E+028</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cor Total</td>
<td>1.504E+031</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Model F-value of 4.33 implies the model is significant. There is only a 1.68% chance that a "Model F-Value" this large could occur due to noise.

Values of "Prob > F" less than 0.0500 indicate model terms are significant. In this case A, B, C, D, BCD are significant model terms.

Values greater than 0.1000 indicate the model terms are not significant.
Scenario 3

<table>
<thead>
<tr>
<th>Source</th>
<th>Sum of Squares</th>
<th>DF</th>
<th>Mean Square</th>
<th>F Value</th>
<th>Prob &gt; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>1.444E+031</td>
<td>45</td>
<td>3.210E+029</td>
<td>4.33</td>
<td>0.0168</td>
</tr>
<tr>
<td>A</td>
<td>1.148E+030</td>
<td>2</td>
<td>5.741E+029</td>
<td>7.75</td>
<td>0.0134</td>
</tr>
<tr>
<td>B</td>
<td>2.704E+030</td>
<td>2</td>
<td>1.352E+030</td>
<td>18.25</td>
<td>0.0010</td>
</tr>
<tr>
<td>C</td>
<td>1.926E+030</td>
<td>2</td>
<td>9.630E+029</td>
<td>13.00</td>
<td>0.0031</td>
</tr>
<tr>
<td>D</td>
<td>4.741E+030</td>
<td>1</td>
<td>4.741E+030</td>
<td>64.00</td>
<td>&lt; 0.0001</td>
</tr>
<tr>
<td>AB</td>
<td>1.852E+029</td>
<td>4</td>
<td>4.630E+028</td>
<td>0.63</td>
<td>0.6579</td>
</tr>
<tr>
<td>AC</td>
<td>6.296E+029</td>
<td>4</td>
<td>1.574E+029</td>
<td>2.13</td>
<td>0.1691</td>
</tr>
<tr>
<td>AD</td>
<td>2.593E+029</td>
<td>2</td>
<td>1.296E+029</td>
<td>1.75</td>
<td>0.2342</td>
</tr>
<tr>
<td>BC</td>
<td>7.407E+028</td>
<td>4</td>
<td>1.852E+028</td>
<td>0.25</td>
<td>0.9018</td>
</tr>
<tr>
<td>BD</td>
<td>3.704E+028</td>
<td>2</td>
<td>1.852E+028</td>
<td>0.25</td>
<td>0.7847</td>
</tr>
<tr>
<td>CD</td>
<td>1.481E+029</td>
<td>2</td>
<td>7.407E+028</td>
<td>1.00</td>
<td>0.4096</td>
</tr>
<tr>
<td>ABC</td>
<td>3.704E+029</td>
<td>8</td>
<td>4.630E+028</td>
<td>0.63</td>
<td>0.7394</td>
</tr>
<tr>
<td>ABD</td>
<td>6.296E+029</td>
<td>4</td>
<td>1.574E+029</td>
<td>2.13</td>
<td>0.1691</td>
</tr>
<tr>
<td>ACD</td>
<td>1.852E+029</td>
<td>4</td>
<td>4.630E+028</td>
<td>0.63</td>
<td>0.6579</td>
</tr>
<tr>
<td>BCD</td>
<td>1.407E+030</td>
<td>4</td>
<td>3.519E+029</td>
<td>4.75</td>
<td>0.0294</td>
</tr>
<tr>
<td>Error</td>
<td>5.926E+029</td>
<td>8</td>
<td>7.407E+028</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cor Total</td>
<td>1.504E+031</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Model F-value of 6.56 implies the model is significant. There is only a 0.42% chance that a "Model F-Value" this large could occur due to noise.

Values of "Prob > F" less than 0.0500 indicate model terms are significant. In this case A, B, C, D, BCD are significant model terms.

Values greater than 0.1000 indicate the model terms are not significant.
Appendix V - Traditional-System Evolutionary Programming Algorithm

/*
TRADITIONAL SYSTEM EVALUATOR
*/

#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <string.h>

float objective(GAGenome &);  
void ArrayInitializer(GAGenome &);  
int ComboMutator(GAGenome & , float);  
float getinfomachine(int infotype, int machinesize);  
float getinfomaterial(int materialtype, int infotype, int counter, float InfoType[]);  
int getinfomaterial(int materialtype, int infotype, int counter, int InfoType[]);  
int declaration();  
int partgeometryevaluatorT(int trial, int parttype);

extern numparts, maxnomachines, machindex;  
int numparts, maxnomachines, machindex;

float *PartDepth, *PartLength, *PartWidth;  
float *AreaMoldBasePlate, *CorCavThickness;  
int *ProductionVolumes;  
float MachiningCost = 65; //$/hr  
int *InjPressureT;  
float *MoldBaseCostT, *CCCostT, *MoldBaseCCCostT;  
int MachineSizeT = 1;  
int trialno = 0;  
int Granularity = 5; //GRANULARITY measure ***************  
int *DesCavArray, *DesRunArray;  
float *TotalDist, *PartTimeTrad;  
int *MachinesUsed, *MachinesUsedBest, *PartPMachine;  
int (*StoreCavArray)[15];  
int (*BestCavArray)[15];  
int (*Runes)[15]; // 15 is the limit on the number of part types  
int (*NoCaveTrad)[15]; // 15 is the limit on the number of part types  
int (*NoRuns)[15];  
double TotalTimeTrad = 0;  
int (*MachineProductionVolumesT)[15]; // 15 is the limit on the number of part types  
int (*BestMachineProductionVolumesT)[15]; // 15 is the limit on the number of part types  
int (*BestGenome)[15]; // 15 is the limit on the number of part types  
//double BestScore = 0; // variable to store the best score across each optimization  
double BestTotalCostT = 1E20;
int main(int argc, char *argv[])
{
    cout << "TradScore
n";
    cout << "This program illustrates the use of a 3DArrayGenome
n";
    cout.flush();

    // See if we've been given a seed to use (for testing purposes). When you
    // specify a random seed, the evolution will be exactly the same each time
    // you use that seed number.

    unsigned int seed = 0;
    for(int ii=1; ii<argc; ii++)
    {
        if(strcmp(argv[ii++],"seed") == 0) {
            seed = atoi(argv[ii]);
        }
    }

    for (trialnoT = 46; trialnoT<=54; trialnoT++){

        if (((trialnoT % 2) != 0) { // if trialno is odd
            numparts = 2;
        }
        else if (((trialnoT % 2) == 0) { // if trialno is even
            numparts = 8;
        }

        maxnomachines = numparts*1; //TEST
        declaration();
        BestMachineProductionVolumesT = new int[maxnomachines][15];
        BestGenome = new int[maxnomachines][15];
        BestCavArray = new int[maxnomachines][15];
        MachinesUsedBest = new int[maxnomachines];
        BestTotalCostT = 1E20;

        // ***** Read in Part Geometry Info *****
        for (int part = 1; part<=numparts; part++){
            partgeometryevaluatorT(trialnoT, part);
        }

        GAAlleleSet<int> alleles; // create an allele with an enumerated list of integers from 0 to Grnularity-1
        for (int granindex = 0; granindex <= Granularity; granindex++){
            alleles.add(granindex);
        }

        GA3DArrayAlleleGenome<int> genome(maxnomachines, numparts, 2, alleles, objective);
        genome.initialize(ArrayInitializer);
        genome.mutator(GA3DArrayAlleleGenome<int>::FlipMutator);
        genome.crossover(GA3DArrayGenome<int>::OnePointCrossover);
GASteadyStateGA ge(genome);

/*
 GAParameterList params;
 GASteadyStateGA::registerDefaultParameters(params); //**** A STEADYSTATE GA is used (see pg 32)
 params.set(gaNpopulationSize, 30); // population size
 params.set(gaNpCrossover, 0.0); // probability of crossover
 params.set(gaNpMutation, 0.02); // probability of mutation
 params.set(gaNnGenerations, 500); // number of generations
 params.set(gaNpReplacement, 0.5); // how much of pop to replace each gen
 params.set(gaNscoreFrequency, 1); // how often to record scores
 //params.set(gaNnReplacement, 4); // how much of pop to replace each gen
 params.set(gaNnReplacement, 5); // how often to record scores
 params.set(gaNscoreFrequency, 1); // how often to record scores
 params.set(gaNnReplacement, 4); // how often to record scores
 params.set(gaNscoreFrequency, 1); // how often to record scores
 params.set(gaNscoreFilename, "bog1.dat");
 params.read("settings.txt"); // grab values from file first
 params.parse(argc, argv, gaFalse); // parse command line for GAlib args
 */

//GASigmaTruncationScaling trunc;
//ga.scaling(trunc);
//ga.set(gaNnPopulationSize, 30);
//ga.set(gaNpCrossover, 0);
//ga.set(gaNpMutation, 0.02);
//ga.set(gaNnGenerations, 500);
//ga.set(gaNselectScores, 7);
//ga.set(gaNpReplacement, 0.3);
//ga.set(gaNscoreFrequency, 1); // how often to record scores
//ga.set(gaNnReplacement, 4); // how often to record scores
//ga.set(gaNnReplacement, 4); // how often to record scores
params.set(gaNscoreFilename, "bog1.dat");
params.read("settings.txt"); // grab values from file first
params.parse(argc, argv, gaFalse); // parse command line for GAlib args

FILE *streamH;
if( (streamH = fopen("BestScoresTra1d.txt", "a")) == NULL )
  fprintf("Couldnt open file
n");
else{
  fprintf( streamH, "%s
", "Trial Number: ");
  fprintf( streamH, "%s
", trialNoT);
  fprintf( streamH, "%s
", "The Best Score for this trial is: ");
  fprintf( streamH, "%s
", BestTotalCostT);
  fprintf( streamH, "%sn
", "Without:");
  for(int Fa=0; Fa<genome.height(); Fa++)
    for(int Fb=0; Fb<genome.width(); Fb++)
      {
fprintf(streamH, "%s\n", BestGenome[Fb][Fc]);
}  
fprintf(streamH, "%s\n", "");  
printf(streamH, "%s\n", "The Machine Production Volumes are as follows (rows=parts, col=machines)\n");  
for(int Fc=0; Fc<numparts; Fc++){
    for(int Fd=0; Fd<maxnomachines; Fd++){
        if (MachinesUsedBest[Fd] == 1) {
            fprintf(streamH, "%d\n", BestMachineProductionVolumesT[Fd][Fc]);
        }
        else{
            fprintf(streamH, "%s\n", "n");
        }
    }
    fprintf(streamH, "%s\n", "");
}
printf(streamH, "%s\n", "The Best NoCav combination used is as follows (rows=parts, col=machines)\n");  
for(int Fc=0; Fc<numparts; Fc++){
    for(int Ff=0; Ff<maxnomachines; Ff++){
        if (MachinesUsedBest[Ff] == 1) {
            fprintf(streamH, "%d\n", BestCavArray[Ff][Fc]);
        }
        else{
            fprintf(streamH, "%s\n", "n");
        }
    }
    fprintf(streamH, "%s\n", ");
}
printf(streamH, "%s\n", "");
fclose(streamH);
}  
cout << "The global best score is: " << BestTotalCostT << "\n";

return 0;

/* Objective Function */
/*                        */
/*                        */
/*                        */

float
objective(GAGenome & c)
{
    GA3DArrayGenome<int> & genome = (GA3DArrayGenome<int> & )c;
    /* */
** Traditional Mold Config Evaluator **

---

MachineProductionVolumes = new int[5][15];
TotalDist = new float[15];
float ChangeoverTime = 1.5; //hours
float AvailableTimeTrad = 24;
float MaxInjPressure, MaxCavPressure, ClampFT = 0, ClampFValue, MaxSeparatingFT;
int *InjPressure = new int[15];
InjPressure = new int[15];
rejectcost1T = new double[5];
rejectcost2T = new double[5];
MachinesUsed = new int[5];
PartPMachine = new int[5];
int TotalMachinesUsed = 0;
moldcostT = new double[5];
processingcostT = new double[5];
machinecostT = new double[5];
MachineCostT = new double[5];
double TotalCostT = 0, BestTradScore;
for (int clearR = 0; clearR < 5; clearR++) {
    rejectcost1T[clearR] = 0;
    rejectcost2T[clearR] = 0;
    MachinesUsed[clearR] = 0;
    PartPMachine[clearR] = 0;
    moldcostT[clearR] = 0;
    machinecostT[clearR] = 0;
    processingcostT[clearR] = 0;
    MachineCostT[clearR] = 0;
}

/* Find the total number of machines used and the number of parts per machine (PartPMachine) */
---

for (int yy = 0; yy < 5; yy++) {
    for (int zz = 0; zz < 15; zz++) {
        if (genome.gene(yy, zz, 0) != 0) {
            MachinesUsed[yy] = 1;
            PartPMachine[yy]++;
        }
    }
    TotalMachinesUsed = TotalMachinesUsed + MachinesUsed[yy];
}
---

int MachineSizeT;
int PowerDatatypeT = 5; // Driving power in the sixth column (0,1,2,3,4,5)

float CoolTimeTypeT = 0;
int DiffusivitytypeT = 1, InjectTempTypeT = 2, MoldTempTypeT = 3, EjectTempTypeT = 4;

InjectTempT = new int[numparts];
MoldTempT = new int[numparts];
EjectTempT = new int[numparts];
float *DiffusivityT;
DiffusivityT = new float[numparts];

PartTimeTrad = new float[numparts];

double piT = 3.1415926535;
int DryCTTypeT = 3, MaxClampStypeT = 4;
float DryCycleTimeT, MaxClampStrokeT;

int BufferCostT, HeatUCostT, InjUCostT, CoolUCostT, EjectCostT;
float ProcCostK1T, ProcCostK2T;
int value1 = 0, value2 = 0;

//get cost data
FILE *streamcostT;
if( (streamcostT = fopen( "CostData.txt", "rb" )) == NULL )
    printf( "Couldn't open file"n ");
else{
    fseek(streamcostT, 0, 0);
    fscanf( streamcostT, "%f", &ProcCostK1T);
    fscanf( streamcostT, "%f", &ProcCostK2T);
    fscanf( streamcostT, "%d", &BufferCostT);
    fscanf( streamcostT, "%d", &HeatUCostT);
    fscanf( streamcostT, "%d", &InjUCostT);
    fscanf( streamcostT, "%d", &CoolUCostT);
    fscanf( streamcostT, "%d", &EjectCostT);
}

fclose(streamcostT);
}

for (int clearB = 0; clearB<maxnomachines; clearB++){
    rejectcost2T[clearB] = 0;
}

for (int clearD=0; clearD < numparts; clearD++){
    TotalDist[clearD] = 0;
}
for (int li=0; li < maxnMachines; li++){
    for (int mm=0; mm < numParts; mm++){
        value1 = genome.gene(li,mm,0);
        TotalDist[mm] = TotalDist[mm] + value1;
    }
}

.showMessage("REJECT Condition: if any part is not built by any of the machines e.g. M1 M2 M3 part A 1 2 1 => TotalDist[A] = 4");

for (int rejectA=0; rejectA < numParts; rejectA++){
    if (TotalDist[rejectA] == 0){
        BestTradScore = 1E20;
        return BestTradScore;
    }
}

for (int qq=0; qq < maxnMachines; qq++){
    for (int rr=0; rr < numParts; rr++){
        MachineProductionVolumesT[qq][rr] = 0;
    }
}

for (int nn=0; nn < maxnMachines; nn++){
    if (MachinesUsed[nn] == 1)//add test to see if machines used array has a 1 or 0 for the element referenced by maxnMachines
        for (int pp=0; pp < numParts; pp++){
            value2 = genome.gene(nn,pp,0);
            if (value2 == 0){
                MachineProductionVolumesT[nn][pp] = 0;
            } else if (value2 != 0){
                value2/TotalDist[pp]*ProductionVolumes[pp];
            }
        }
}

.showMessage("Create arrays to STORE the current cavity combo for a machine. ");

StoreCavArray = new int[maxnMachines][15];

for (int clearJ = 0; clearJ < maxnMachines; clearJ++){
    for (int clearK = 0; clearK < numParts; clearK++){
        StoreCavArray[clearJ][clearK] = 0;
        //BestCavArray[clearJ][clearK] = 0;
    }
}
/**
 * Find the number of run possibilities for each part type
 * based on the machine production volumes
 * -> NoRuns[maxNoMachines][nParts]
 */

for (int mindex=0; mindex<maxNoMachines; mindex++) {
    for (int Partindex=0; Partindex<nParts; Partindex++) {
        NoRuns[mindex][Partindex] = 0;
        for (int run=1; run <= (floor(MachineProductionVolumesT[mindex][Partindex]/2)+1); run++) {
            // Integer
            NoRuns[mindex][Partindex]++;
        }
        NoRuns[mindex][Partindex]++;
    }
}

/**
 * Cycle through all machines
 */

for (int machindex =0; machindex <maxNoMachines; machindex++) {
    if (MachinesUsed[machindex] == 1) {
        // ***** MAXIMUM CAVITY PRESSURE ******

        int infoTypeInjPressureT = 5; // Inj. Pressure data appears in the 6th column in
        materialdatafile

        for (int IPindex=0; IPindex<nParts; IPindex++) {
            getInfomaterial(Material[IPindex], infoTypeInjPressureT, IPindex, InjPressureT); // InjPressure in
            bars

            if (InjPressureT[IPindex] > MaxInjPressureT)
                MaxInjPressureT = InjPressureT[IPindex];
        }
        MaxCavPressureT = 0.5 * MaxInjPressureT * 100000; // N/m2

        Runs = new int[MaxNoRuns][15];
        NoCavsTrad = new int[MaxNoRuns][15];
    }
for (int clearL = 0; clearL<MaxNoRuns; clearL++){
    for (int clearM = 0; clearM<numparts; clearM++){
        Runs[clearL][clearM] = 0;
        NoCavsTrad[clearL][clearM] = 0;
    }
}

for (int ind=0; ind <numparts; ind++){
    int runindex = 0;
    for (int runtype = 1; runtype <= MachineProductionVolumesT[machindex][ind]; runtype++){
        if (MachineProductionVolumesT[machindex][ind] == 0){
            Runs[runindex][ind]=0;
            NoCavsTrad[runindex][ind] = 0;
            runindex++;
        } else if (MachineProductionVolumesT[machindex][ind] % runtype == 0){
            Runs[runindex][ind]=runtype;
            //Find the corresponding #cav required
            NoCavsTrad[runindex][ind] = MachineProductionVolumesT[machindex][ind]/runtype;
            runindex++;
        }
    }
}

/*****************************************************************************/
// Define and fill the array that stores the designated NoCav value (DesCavArray) and the array that stores the designated Run value for each part type (DesRunArray).
/*****************************************************************************/
DesCavArray = new int[numparts];
DesRunArray = new int[numparts];
for (int clearC = 0; clearC<numparts; clearC++){
    DesCavArray[clearC] = 0;
    DesRunArray[clearC] = 0;
}

int Desdumbvar = 0;
int geneval;
for (int DesInd = 0; DesInd<numparts; DesInd++){
    geneval = genome.gene(machindex, DesInd, 1);
    if (((NoRuns[machindex][DesInd] - 1) * geneval) % Granularity) < (Granularity/2)){
        Desdumbvar = floor((NoRuns[machindex][DesInd] - 1) * geneval / Granularity);
    } else{
        Desdumbvar = ceil((NoRuns[machindex][DesInd] - 1) * geneval / Granularity);
    }
    DesCavArray[DesInd] = NoCavsTrad[Desdumbvar][DesInd];
    DesRunArray[DesInd] = Runs[Desdumbvar][DesInd];
}

/*****************************************************************************/
// TRADITIONAL - MACHINE SIZE CALCULATION
/*****************************************************************************/
int TotalProjAreaT = 0;

// ******** TOTAL PROJECTED SHOT AREA ********
for (int TPApart = 0; TPApart < numparts; TPApart++) {
    int CalcTotalProjAreaT = ProjectedAreaPerPart[TPApart] * DesCavArray[TPApart];
    //TotalAreaPerPart is a function of the part design (includes runner vol @15%)
    if (CalcTotalProjAreaT > TotalProjAreaT)
        TotalProjAreaT = CalcTotalProjAreaT;
}

// ************************************************************************************************************

// ******** MAX SEPARATING FORCE ********
MaxSeparatingFT = TotalProjAreaT * 0.0001 * MaxCavPressureT / 1000; //TotalProjArea in cm2, Max Cav Pressure in N/m2, MaxSeparatingForce in kN

// ******** CLAMP FORCE ********
ClampFT = 0;
MachineSizeT = 1;
int MachineDatatypeT = 0; // Clamp Force is the leftmost column
ClampFValueT = getimachine(MachineDatatypeT, MachineSizeT); // read data from column 1, row 1
(info type range starts at 0, machinesize range starts at 1)
while (ClampFT != ClampFValueT){
    if (MaxSeparatingFT < ClampFValueT) {
        ClampFT = ClampFValueT; //rounded to the next machine size, from
        machinedata.txt
    } else if (MaxSeparatingFT >= 8500){ //8500kN max clamping force
        // ******* Reject mold Combo *******
        ClampFT = 9999;
        rejectcostT[machindex] = 1E15;
        ClampFValueT = ClampFT;
        MachineSizeT = 7;
    } else{ // ******* read next value
        MachineSizeT++;
        ClampFValueT = getimachine(MachineDatatypeT, MachineSizeT);
        if (ClampFValueT == MaxSeparatingFT) {
            MaxSeparatingFT = MaxSeparatingFT + 1;
        }
    }
}

// ************************************************************************************************************

"" TRADITIONAL - CYCLE TIME CALCULATIONS ""

TotalTimeTrad = 0;
for (int tt = 0; tt < numparts; tt++) {........

// Injection Time
    cycletimesT[tt][1] = 2 * (PartVol[tt] * 0.000001) * pow(DesCavArray[tt], 0.7) * 
                (InjPressureT[tt] * 100000) / (DrivingPowerT[1] * 1000); //Inj[1] = row rr, column 2
// the correction factor of raising the NoCavs to the power of 0.7 was used instead of pure multiplication

// Cooling Time and Heating Time
getinfomaterial(Material[tt], DiffusivityTypeT, tt, DiffusivityT);
getinfomaterial(Material[tt], InjectTempTypeT, tt, InjectTempT);
getinfomaterial(Material[tt], MoldTempTypeT, tt, MoldTempT);
getinfomaterial(Material[tt], EjectTempTypeT, tt, EjectTempT);

// cooling time
CoolTimeT = pow(MaxThickness[tt],2)/(pow(pIT,2)*DiffusivityT[tt])*log((4*(InjectTempT[tt] -
MoldTempT[tt]))/(pIT*(EjectTempT[tt] - MoldTempT[tt])));
if (CoolTimeT >= 3){
cycletimes[T[tt][2] = CoolTimeT;
}
else if (CoolTimeT <3){
cycletimes[T[tt][2] = 3; // 3 sec minimum
}

// heating time = % of cooling time
cycletimes[T[tt][0] = 0.5 * cycletimes[T[tt][2];

// Mold-Reset Time
DryCycleTimeT = getinfomachine(DryCTypeT, MachineSizeT); //seconds
MaxClampStrokeT = getinfomachine(MaxClampStypeT, MachineSizeT); //cm

cycletimes[T[tt][3] = 1+1.75*DryCycleTimeT*pow(((2*PartDepth[tt]+5)/MaxClampStrokeT),0.5);

// Total Time
PartTimeTrad[tt] = (cycletimes[T[tt][0] + cycletimes[T[tt][1] + cycletimes[T[tt][2] +
cycletimes[T[tt][3]]*DesRunArray[tt];

TotalTimeTrad = TotalTimeTrad + PartTimeTrad[tt]; // this is the time it takes to produce all parts on machine 'machindex'
}

/******************************************************************************
** Check to see if the TotalTimeTrad                                           **
** (which equals (the sum of cycles times)#runs                                **
** is less that the Available Time for the specific machine based on the number **
** of parts run on the machine and the respective number of changeover times  **
*******************************************************************************/

AvailableTimeTrad = 24 - ChangeoverTime * (PartPMachine[machindex]-1); //available

hours per day with the changeover times accounted for

if (TotalTimeTrad/60/60 > AvailableTimeTrad){
    rejectcost2T[machindex] = 1E15;
}
else if (TotalTimeTrad/60/60 <= AvailableTimeTrad){
    rejectcost2T[machindex] = 0;
}
// Calculate MOLD COST and store current cavity combination

for (int uu=0; uu < numparts; uu++){
    pow(DesCavArray[uu],0.7);
    StoreCavArray[machindex][uu]=DesCavArray[uu];
}

// Calculate PROCESSING COST

processingcostT[machindex] = (ProcCostK1T+(ProcCostK2T*ClampnT))*AvailableTimeTrad + ChangeoverTime * (PartPMachine[machindex]-1)*250; //250 days per year [use k1 = 25.103, k2=0.0075]

// Calculate MACHINE COST

machinecostT[machindex] = 260000; //see CostData.txt file

// AMMORTIZATION

float interestT = 0.1;
int moldperiodT = 3, machineperiodT = 5;

moldcostT[machindex] = moldcostT[machindex] * (interestT * pow((1+interestT),moldperiodT))/(pow((interestT+1),moldperiodT)-1);
machinecostT[machindex] = machinecostT[machindex] * (interestT * pow((1+interestT),machineperiodT))/(pow((interestT+1),machineperiodT)-1);

//calc the TotalCost per run = moldcost(#cavs) + processing cost(#cavs) + reject cost
MachineCostT[machindex] = (moldcostT[machindex] + processingcostT[machindex] +
machinecostT[machindex] + rejectcostT[machindex] + rejectcost2T[machindex]);

TotalCostT = TotalCostT + MachineCostT[machindex];

} //end machindex <maxnomachines>

if (BestTotalCostT > TotalCostT)
    BestTotalCostT = TotalCostT;
    for (int Mind = 0; Mind < maxnomachines; Mind++)
        for (int PInd = 0; PInd < numparts; PInd++)
            BestCavArray[Mind][PInd] = StoreCavArray[Mind][PInd];

    BestMachineProductionVolumesT[Mind][PInd] =

    MachineProductionVolumesT[Mind][PInd];

    BestGenome[Mind][PInd] = genome.gene(Mind, PInd, 0);

    MachinesUsedBest[Mind] = MachinesUsed[Mind];

} //end printind < maxnomachines; printind++)

if (MachinesUsed[printind] == 1)
    char costsfileTrad[30] = "CostsTrad";
char triatypestringTrad[] = ".";
itoa(triatnoT, triatypestringTrad, 10);
strcat(costsfileTrad, triatypestringTrad, 5);
strcat(costsfileTrad, ".txt");
FILE *streamG;
if (streamG = fopen(costsfileMTrad, "a")) == NULL)
    print("Couldn't open file\n");
else{
    fprintf( streamG, "%s\n", "***** Machine Number: ");
    fprintf( streamG, "%d\n", printind);
    fprintf( streamG, "%s\n", "***** Process Cost ***** Machine Cost 

   Reject 1  Reject 2

   Costs Trad);.
    fprintf( streamG, "%g\n", moldcostT[printind]);
    fprintf( streamG, "%g\n", processingcostT[printind]);
    fprintf( streamG, "%g\n", machinecostT[printind]);
    fprintf( streamG, "%g\n", rejectcost1T[printind]);
    fprintf( streamG, "%g\n", rejectcost2T[printind]);
    fprintf( streamG, "%s\n", "**** Total Time Trad ***** Available Time Trad 

   Clamp Force");
    fprintf( streamG, "%g\n", TotalTimeTrad/60/60);
    fprintf( streamG, "%g\n", AvailableTimeTrad);
    fprintf( streamG, "%g\n", ClampFT);
    fprintf( streamG, "%s",

   "----------------------------------------------------------------------

   ** PRINT stuff **

  ----------------------------------------------------------------------

   char costsfileMTrad[30] = "CostsTrad";
   char trialtypestringMTrad[] = "\"
   _itoa(trialnoT, trialtypestringMTrad, 10);
   strcat(costsfileMTrad, trialtypestringMTrad , 5);
   strcat(costsfileMTrad, ".txt");
   FILE *streamF;
   if (streamF = fopen(costsfileMTrad, "a")) == NULL)
    print("Couldn't open file\n");
else{
    fprintf( streamF, "%s\n", "***** Total Machines Used: ");
    fprintf( streamF, "%d\n", TotalMachinesUsed);
    fprintf( streamF, "%s\n", "*****

   TOTAL COST ********* Clamp FT **********

   for(int Fp=0; Fp<genome.height(); Fp++)
     for(int Fr=0; Fr<genome.width(); Fr++){
       fprintf( streamF, "%d", genome.gene(Fr,Fp));
     }
    fprintf( streamF, "%s\n", "\n
   \n
126"
for(int Fs=0; Fs<numparts; Fs++) {
    for(int Ft=0; Ft<maxomachines; Ft++) {
        if (MachinesUsed[Ft] == 1) {
            fprintf (streamF, "%d\n", MachineProductionVolumesT[Ft][Fs]);
        } else {
            fprintf (streamF, "%s\n", "n");
        }
    }
    fprintf (streamF, "%s\n", "");
}

fprintf (streamF, "%s\n", "The Best NoCav combination used is as follows (rows=parts, col=machines):");
for(int Fu=0; Fu<numparts; Fu++) {
    for(int Fv=0; Fv<maxomachines; Fv++) {
        if (MachinesUsedBest[Fv] == 1) {
            fprintf (streamF, "%d\n", StoreCavArray[Fv][Fu]);
        } else {
            fprintf (streamF, "%s\n", "n");
        }
    }
    fprintf (streamF, "%s\n", "");
}

fprintf (streamF, "%s\n", "The Best TradScore = %f\n", BestTradScore);
void 
ArrayInitializer(GAAlleleGenome & g) {
    GA3DArrAyAlleleGenome<int> & genome = (GA3DArrAyAlleleGenome<int> & ) g;  // may have to change the word genome to something else
    int value3 = 0;
    int value4 = 0;
    // **** fill with zeroes ******
    for (int i = 0; i < genome.width(); i++){
        for (int j = 0; j < genome.height(); j++){
            for (int p = 0; p < 2; p++){
                genome.gene(i,j,p,0);
            }
        }
    }
    // *** OPTION 1 - fill the genome with randomly selected values from the allele set
    for (int k = 0; k < maxnornachines; k++) { // maxnornachines = genome.width()
        for (int i = 0; i < numparts; i++) { // numparts = genome.height()
            for (int m = 0; m < 2; m++) { // 2 = genome.depth()
                value3 = GARandomInt(0, Granularity);
                genome.gene(k,i,1,value3);
            }
        }
    }
    // *** OPTION 2 - fill the first machine with all parts to start, random cavity setting ******
    for (int m = 0; m < numparts; m++) {
        //value4 = GARandomInt(0, Granularity);
        genome.gene(0,m,0,1);
        // genome.gene(0,m,1,value4);
    }
}

// Here we override the built-in write method for the 3DArray genome so
// that we get better spacing. The default just stacks the characters one
// after another. Here we do fixed spacing so that the -1 and 1 don't screw
// each other up.
int
GA3DArrAyAlleleGenome<int>::write(ostream & os) const
{
    for(unsigned int j=0; j<ny; j++){
        for(unsigned int i=0; i<nx; i++){
            os.width(3);
            os << gene(i,j,0);
        }
        os << endl;
    }
    return os.fail() ? 1 : 0;
}

// Here we define the function that gets INTEGER information from the material datafile
// regarding injection pressures etc. based on the materialtype.
int getinfomaterial(int materialtype, int infotype, int counter, int InfoType[]){
FILE *stream;
float testfloat = 0, dumbvar = 0;
if (stream = fopen("materialdata.txt", "rb")) == NULL )
    printf("Couldn't open file\n");
else{
    fseek(stream, 0, 0);
    while (!feof(stream)){
        fscanf (stream, "%f", &testfloat);
        if (testfloat == -materialtype){
            for (int m = 1; m <= infotype; m++){
                fscanf (stream, "%f", &dumbvar);
            }
            fscanf( stream, "%d", &InfoType[counter]);
            break;
        }
        else if (testfloat == -9999){
            break;
        }
    }
    fclose(stream);
}
return 0;

// Here we define the function that gets FLOAT information from the material datafile
// regarding injection pressures etc. based on the materialtype.

float getinfomaterial(int materialtype, int infotype, int counter, float InfoType[]){
    FILE *stream;
    float testfloat = 0, dumbvar = 0;
    if (stream = fopen("materialdata.txt", "rb")) == NULL )
        printf("Couldn't open file\n");
    else{
        fseek(stream, 0, 0);
        while (!feof(stream)){
            fscanf (stream, "%f", &testfloat);
            if (testfloat == -materialtype){
                for (int m = 1; m <= infotype; m++){
                    fscanf (stream, "%f", &dumbvar);
                }
                fscanf( stream, "%d", &InfoType[counter]);
                break;
            }
            else if (testfloat == -9999){
                break;
            }
        }
    }
    fclose(stream);
}
return 0;

// Here we define the function that gets information from the machine datafile
// regarding Clamping Forces, Processing Costs, etc. based on the machine type.

float getinfomachine(int infotype, int machinesize){
FILE *stream;
float dumbvar = 0;
float machinedatavar = 0;
if ( (stream = fopen("machinedata.dat", "rb")) == NULL )
  printf("Could\'t open file\n");
else{
  fseek(stream, 0, 0);
  for (int hh = 0; hh < infotype; hh++){
    fscanf(stream, "%f", &dumbvar);
  }
  if (machinesize == 1){
    fscanf (stream, "%f", &machinedatavar);
    fclose (stream);
    return (machinedatavar);
  }
  else{
    for (int jj = 1; jj <= (6 * (machinesize - 1)); jj++){
      fscanf (stream, "%f", &dumbvar);
    }
    fclose (stream);
  }
  return (machinedatavar);
}

/************************************************************
==
==   Here we declare a bunch of DYNAMIC arrays needed later.  ==
==
************************************************************/

int declaration() {
    ProjectedAreaPart = new float[numparts];
    Material = new float[numparts];
    PartVol = new float[numparts];
    MaxThickness = new float[numparts];
    PartDepth = new float[numparts];
    PartLength = new float[numparts];
    PartWidth = new float[numparts];
    //float *AreaMoldBasePlate, *CorCavThickness;
    AreaMoldBasePlate = new float[numparts];
    CorCavThickness = new float[numparts];
    Nspi = new int[numparts];
    Napo = new int[numparts];
    Nhd = new int[numparts];
    Npull = new int[numparts];
    Nilift = new int[numparts];
    Nunscrew = new int[numparts];
    NpulScrewSides = new int[numparts];
    NscrewRear = new int[numparts];
SF\text{finish} = \text{new float}[\text{numparts}];
\text{Tolerance} = \text{new float}[\text{numparts}];
\text{Texture} = \text{new float}[\text{numparts}];
\text{PartPlane} = \text{new float}[\text{numparts}];
// \text{Nspi} = \text{number of inner surface patches}
// \text{Nso} = \text{number of outer surface patches}
// \text{Nhd} = \text{number of holes and depressions}
// \text{Ns pull} = \text{number of side pulls}
// \text{Nlift} = \text{number of internal lifter}
// \text{Nunscrew} = \text{number of unscrew devices}
// \text{Ns pull screw Sides} = \text{number of sides of part requiring side pulls or unscrew devices}
// 11 = \text{left or right}
// 21 = \text{left and right}
// 22 = \text{top and bottom}
// 23 = \text{left or right and (top or bottom)}
// 31 = \text{left and right and (top or bottom)}
// 32 = \text{top and bottom and (left or right)}
// 40 = \text{left and right and top and bottom}
\text{Texture} = \text{surface texture}
\text{PartPlane} = \text{paring plane factor}

// \text{Me} = \text{new float}[\text{numparts}];
\text{Mx} = \text{new float}[\text{numparts}];
\text{Xi} = \text{new float}[\text{numparts}];
\text{Xo} = \text{new float}[\text{numparts}];
\text{Mpo} = \text{new float}[\text{numparts}];
\text{Ms pull} = \text{new float}[\text{numparts}];
\text{Mlli ft} = \text{new float}[\text{numparts}];
\text{Munscrew} = \text{new float}[\text{numparts}];
\text{Ms finish} = \text{new float}[\text{numparts}];
\text{Mtol} = \text{new float}[\text{numparts}];
\text{Mtexture} = \text{new float}[\text{numparts}];
\text{Mpartplane} = \text{new float}[\text{numparts}];
\text{Mtotal} = \text{new float}[\text{numparts}];
\text{ProductionVolumes} = \text{new int}[\text{numparts}];
// \text{MoldBaseCost}, \text{CCCost}, \text{MoldBaseCCCost}
\text{MoldBaseCostT} = \text{new float}[\text{numparts}];
\text{CCCostT} = \text{new float}[\text{numparts}];
\text{MoldBaseCCCostT} = \text{new float}[\text{numparts}];
\text{InjPressureT} = \text{new int}[\text{numparts}];
\text{MachineSize} = 1;

// \text{Xi} = \text{inner complexity}
// \text{Xo} = \text{outer complexity}
// \text{Me} = \text{manufacturing hours due to ejector pins}
// \text{Mx} = \text{manufacturing hours due to geometrical features}
// \text{Mpo} = \text{manufacturing hours due to size}
// \text{Ms pull} = \text{manufacturing hours due to side pulls}
// \text{Mlli ft} = \text{manufacturing hours due to internal lifter}
// \text{Munscrew} = \text{manufacturing hours due to unscrew devices}
// \text{Ms finish} = \text{manufacturing hours due to surface finish}
// \text{Mtol} = \text{manufacturing hours due to tolerance}
// \text{Mtexture} = \text{manufacturing hours due to surface texture}
 Appendix V

// Mpartplane = manufacturing hours due to parting plane factor

        return 0;
    }

/**************************************************************************
** Here we read part geometry information from respective part files.    **
** These files are labeled "part1.txt", "part2.txt", etc.               **
/**************************************************************************

// ***** Read values from file ******

int partgeometryevaluatorT(int trial, int parttype){
    int dumbvar;
    char filename[30] = "part";
    int type = trial;
    char typestring[] = "";
    int ptype = parttype;
    char ptypestring[] = "";
    _itoa(type, typestring, 10);
    _itoa(ptype, ptypestring, 10);
    strcat(filename, typestring , 5);
    strcat(filename, "-" );
    strcat(filename, ptypestring, 5);
    strcat(filename, ".txt" );
    printf( filename );
    FILE *stream;
    if( (stream = fopen(filename, "rb" )) == NULL )
        print( "Couldn't open file
" );
    else{
        fseek(stream, 0, 0);
        fscanf (stream, "%d", &dumbvar);
        if (dumbvar != parttype){
            printf( "Incorrect file or format
" );
            return(0);
        }
        fscanf (stream, "%f", &ProjectedArea[parttype - 1]);
        fscanf (stream, "%f", &Material[parttype - 1]); //value 1-11
        fscanf (stream, "%f", &PartVol[parttype - 1]); //incl runners - cm3
        fscanf (stream, "%f", &MaxThickness[parttype - 1]); //mm
        fscanf (stream, "%f", &PartDepth[parttype - 1]); //cm
        fscanf (stream, "%f", &PartLength[parttype - 1]); //cm
        fscanf (stream, "%f", &PartWidth[parttype - 1]); //cm
        fscanf (stream, "%d", &Nspil[parttype - 1]);
        fscanf (stream, "%d", &Nspol[parttype - 1]);
        fscanf (stream, "%d", &Nhd[parttype - 1]);
        fscanf (stream, "%d", &Nspull[parttype - 1]);
        fscanf (stream, "%d", &Nillf[parttype - 1]);
        fscanf (stream, "%d", &Nunscrew[parttype - 1]);
        fscanf (stream, "%d", &NpulcrewSides[parttype - 1]);
        fscanf (stream, "%d", &NcrewRear[parttype - 1]);
        fscanf (stream, "%f", &SFinish[parttype - 1]);
        fscanf (stream, "%f", &Tolerance[parttype - 1]);
        fscanf (stream, "%f", &Texture[parttype - 1]);
        fscanf (stream, "%f", &PartPlane[parttype - 1]);
    }

132
Il

' Single Cavity Mold Base Cost *

// Area of Mold Base Cavity Plate
// side pull or screwing device (top, bottom, left, right) => double clearance
float ClearanceLength = 15; ClearanceWidth = 15; // cm (7.5cm on each side)

if (NpullscrewSides[partype - 1] == 11)
{
    ClearanceWidth = ClearanceWidth + 7.5;
}
else if (NpullscrewSides[partype - 1] == 12)
{
    ClearanceLength = ClearanceLength + 7.5;
}
else if (NpullscrewSides[partype - 1] == 21)
{
    ClearanceWidth = ClearanceWidth + 15;
}
else if (NpullscrewSides[partype - 1] == 22)
{
    ClearanceLength = ClearanceLength + 15;
}
else if (NpullscrewSides[partype - 1] == 23)
{
    ClearanceWidth = ClearanceWidth + 7.5;
    ClearanceLength = ClearanceLength + 7.5;
}
else if (NpullscrewSides[partype - 1] == 31)
{
    ClearanceLength = ClearanceLength + 15;
    ClearanceLength = ClearanceLength + 7.5;
}
else if (NpullscrewSides[partype - 1] == 32)
{
    ClearanceWidth = ClearanceWidth + 15;
    ClearanceLength = ClearanceLength + 15;
}
else if (NpullscrewSides[partype - 1] == 40)
{
    ClearanceWidth = ClearanceWidth + 15;
    ClearanceLength = ClearanceLength + 15;
}

AreaMoldBasePlate[partype - 1] = (PartLength[partype - 1] + ClearanceLength) * (PartWidth[partype - 1] + ClearanceWidth);

// Thickness
// screwing device (behind) => double clearance
float ClearanceDepth = 15; // cm (7.5cm in front and 7.5cm behind part)
// NscrewRear = rear unscrew devices
// 1 = present 0 = NOT present
if (NscrewRear[partype - 1] == 1)
{
    ClearanceDepth = ClearanceDepth + 7.5;
}

CorCavThickness[partype - 1] = (PartDepth[partype - 1] + ClearanceDepth);

/* N.B. the increase to the clearance spaces of 0.5cm for every 100cm2 of Total Area is ignored in this case due to the fact that the costs for a SINGLE cavity mold are calculated. This is then used as the
basis for a multicavity mold */

\[ \text{MoldBaseCostT}[\text{parttype - 1}] = 1000 + 0.45 \times \text{AreaMoldBasePlate}[\text{parttype - 1}] \times \text{pow}((\text{CorCavThickness}[\text{parttype - 1}], 0.4); \]

// "Single Core and Cavity Cost"

// Part Complexity
// inner complexity
\[ X_1[\text{parttype - 1}] = 0.01 \times N_1[\text{parttype - 1}] + 0.04 \times \text{Nhd}[\text{parttype - 1}]; \]
// outer complexity
\[ X_0[\text{parttype - 1}] = 0.01 \times N_0[\text{parttype - 1}]; \]

// Ejector pins
\[ M_1[\text{parttype - 1}] = \text{pow}(\text{ProjectedAreaperPart}[\text{parttype - 1}], 0.5) \times 2.5; \]

// Geometrical Features
\[ M_2[\text{parttype - 1}] = 45 \times \text{pow}(X_1[\text{parttype - 1}] + X_0[\text{parttype - 1}], 1.27); \]

// Size
\[ M_3[\text{parttype - 1}] = 5 + 0.085 \times \text{pow}(\text{ProjectedAreaperPart}[\text{parttype - 1}], 1.2); \]

// Side pulls
\[ M_{spull}[\text{parttype - 1}] = N_{spull}[\text{parttype - 1}] \times 65; \]

// Internal lifters
\[ M_{lilift}[\text{parttype - 1}] = N_{ilift}[\text{parttype - 1}] \times 150; \]

// Unscreeving devices
\[ M_{unscrew}[\text{parttype - 1}] = N_{unscrew}[\text{parttype - 1}] \times 250; \]

// Surface finish
// find SFinish
\[ M_{sfinish}[\text{parttype - 1}] = S_{finish}[\text{parttype - 1}] \times (M_1[\text{parttype - 1}] + M_2[\text{parttype - 1}] + M_3[\text{parttype - 1}]); \]

// Tolerance
\[ M_{tol}[\text{parttype - 1}] = \text{Tolerance}[\text{parttype - 1}] \times M_2[\text{parttype - 1}]; \]

// Texture
\[ M_{texture}[\text{parttype - 1}] = \text{Texture}[\text{parttype - 1}] \times (M_1[\text{parttype - 1}] + M_2[\text{parttype - 1}] + M_3[\text{parttype - 1}]); \]

// Parting Plane
\[ M_{partplane}[\text{parttype - 1}] = \text{PartPlane}[\text{parttype - 1}] \times \text{pow}(\text{ProjectedAreaperPart}[\text{parttype - 1}], 0.5); \]

// * Total number of manufacturing hours *
\[ M_{total}[\text{parttype - 1}] = M_1[\text{parttype - 1}] + M_3[\text{parttype - 1}] + M_2[\text{parttype - 1}] + M_{spull}[\text{parttype - 1}] + M_{lilift}[\text{parttype - 1}] + M_{unscrew}[\text{parttype - 1}] + M_{sfinish}[\text{parttype - 1}] + M_{tol}[\text{parttype - 1}] + M_{texture}[\text{parttype - 1}] + M_{partplane}[\text{parttype - 1}]; \]

// * Core and Cavity Cost *
\[ \text{CCCostT}[\text{parttype - 1}] = M_{total}[\text{parttype - 1}] \times \text{MachiningCost}; \]

// * Total Single Mold Base and Core & Cavity Cost *
\[ \text{MoldBaseCCCottT}[\text{parttype - 1}] = \text{MoldBaseCostT}[\text{parttype - 1}] + \text{CCCottT}[\text{parttype - 1}]; \]

return 0;
// If your compiler does not do automatic instantiation (e.g. g++ 2.6.8),
// then define the NO_AUTO_INST directive.
#endif
#include <ga/GAAAllele.C>
#include <ga/GA3DArrayGenome.C>
#if defined(__GNUG__)
  template class GAAAlleleSet<int>;
  template class GAAAlleleSetCore<int>;
  template class GAArray<int>;
  template class GA3DArrayGenome<int>;
  template class GA3DArrayAlleleGenome<int>;
#else
  GAAAlleleSet<int>;
  GAAAlleleSetCore<int>;
  GAArray<int>;
  GA3DArrayGenome<int>;
  GA3DArrayAlleleGenome<int>;
#endif
#endif
Appendix VI - New-System Evolutionary Programming Algorithm

/*
 * NEW SYSTEM EVALUATOR
 */
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <ga/ga.h>
#include <math.h>
#include <string.h>

// The objective function tells how good a genome is. The initializer defines
// how the lists should be initialized.
float objective(GAGenome &);
void ListInitializer(GAGenome &);
int SplitJoinMutator(GAGenome & float);
int getinfodata(int product, int ctvare);
int getinfomaterial(int materialtype, int infotype, int counter, int InfoType[]);
float getinfomaterial(float materialtype, int infotype, int counter, float InfoType[]);
int declaration();
int partgeometryevaluator(int trial, int parttype);
float getinfomachine(int infotype, int machinesize);
int GCD(int m, int n);
int FindNoFreq(int gcd);
int FrequencyFill(int gcd);
extern NoFreq, numparts;
int frequencyval;
int *Frequency, NoFreq;
int numparts;
float *PartDepth, *PartLength, *PartWidth;
float *AreaMoldBasePlate, *CorCavThickness;
int *ProductionVolumes;
float MachiningCost = 65; //$/hr
int *InjPressure;
float *MoldBaseCost, *CCost, *MoldBaseCCost;
int MachineSize = 1;
int trialno = 0;
int BestWorst = 1; //BestWorst = 1 for Best Case Scenario //BestWorst = 4 for Worst Case Scenario

// *****************************************************
// ******** MAIN ********
//**********************************************************

int main(int argc, char *argv[])
```c
int trials = 54; // the number of trials to be run

for (trialno = 1; trialno<=trials; trialno++){
    if ((trialno % 2) != 0){ // if trialno is odd
        numparts = 2;
    } else if ((trialno % 2) == 0){ // if trialno is even
        numparts = 8;
    }

    declaration();

    // ***** Read in Part Geometry info *****
    for (int part = 1; part<=numparts; part++){
        partgeometryevaluator(trialno, part);
    }

    double BestScore = 0; // variable to store the best score across each optimization
    double TrackScore = 0;
    int FreqCount = 0; // var to keep track of the frequency corresponding to the best score

    // ***** Find the Greatest Common Divisor *****
    int TestGCD = ProductionVolumes[0];
    for (int nn = 2; nn<=numparts; nn++){
        TestGCD = GCD(TestGCD, ProductionVolumes[nn-1]);
    }

    // ***** Find the Minimum Part Set *****
    int *MPSet;
    MPSet = new int[numparts];
    for (int gg=0; gg<numparts; gg++){
        MPSet[gg] = ProductionVolumes[gg]/TestGCD;
    }

    // ***** Find the Number of Frequencies *****
    NoFreq = FindNoFreq(TestGCD);
    //NoFreq = 1; // for TESTING purposes only

    // *** define and fill the Frequency array with all the possible frequency values
    Frequency = new int[NoFreq];
    FrequencyFill(TestGCD);

    for (int ff=0; ff<NoFreq; ff++){
        //OPTION 1 - all frequencies tested
        //Int ff = GARandomInt(0, NoFreq-1); //OPTION 2 - random frequency tested (see line 223)
        //Int ff = -1; //OPTION 3 - highest allowable frequency
        while (((BestScore == 0) || (BestScore > 1E10) || (TrackScore <= BestScore)) && (Frequency[ff] != 1)) {
            //OPTION 3 - continues if a feasible option is not found
            // ff++;
            //OPTION 3 - increment ff (starts at NoFreq-1)
            frequencyval = Frequency[ff]; // Find frequency value from an array
        }
    }
```
cout << "Lean Injection Molding \n\n";
cout << "This program finds the (close to) optimal solution for the number of \n";
cout << "molds and the number of cavities per mold. This genetic algorithm \n";
cout << "takes into account the mold base costs, the processing costs and the\n";
cout << "costs due to the addition of processing stations. The mold combination and \n";
cout << "frequency ensure that the Takt time for the parts is always met. \n";
cout.flush();

// See if we've been given a seed to use (for testing purposes). When you
// specify a random seed, the evolution will be exactly the same each time
// you use that seed number.

for(int ii=1; ii<argc; ii++) {
    if(strcmp(argv[ii++], "seed") == 0) {
        GARandomSeed((unsigned int)atoi(argv[ii]));
    }
}

// Set the default values of the parameters.

GAParameterList params;
GASteadyStateGA::registerDefaultParameters(params);  //***** A STEADYSTATE GA is used (see pg32)
params.set(gaNpopulationSize, 30);  // population size
params.set(gaNpCrossover, 0.0);  // probability of crossover
params.set(gaNpMutation, 0.2);  // probability of mutation
params.set(gaNnGenerations, 500);  // number of generations
params.set(gaNpReplacement, 0.5);  // how much of pop to replace each gen
params.set(gaNscoreFrequency, 1);  // how often to record scores
params.set(gaNsreplacement, 4);  // how much of pop to replace each gen
params.set(gaNscoreFrequency, 1);  // how often to record scores
params.set(gaNflushFrequency, 1);  // how often to dump scores to file
params.set(gaNscoreFilename, "bog.dat");
params.read("settings.txt");  // grab values from file first
params.parse(argc, argv, gaFalse);  // parse command line for GAlib args

// Now create the GA and run it. We first create a genome with the
// operators we want. Since we're using a template genome, we must assign
// all three operators.

GAListGenome<int> genome(objective);
genome.initializer(InitListInitializer);
genome.mutator(SplitJoinMutator);

// Now that we have our genome, we create the GA (it clones the genome to
// make all of the individuals for its populations). Set the parameters on
// the GA then let it evolve.

GASteadyStateGA ga(genome);
ga.crossover(GAListGenome<int>::OnePointCrossover);
ga.parameters(params);
ga.minimize();  // by default we want to minimize the objective
ga.selectScores(GAStatistics::AllScores);
ga.evolve();

// Assign the best that the GA found to our genome then print out the results.
// write the best scores to a file called BestScore

FILE *stream;
if (stream = fopen("BestScores.txt", "a")) == NULL )
    printf("Couldn't open file\n");
else{
    fprintf( stream, "%s\n", genome.score());
    for (int rr = 1; rr <= genome.size()-9; rr++)
        fprintf( stream, "\n%ld\n", node);
    fprintf( stream, "%s\n", char(fmod("\n", genome.warp(rr)),100));
}
    fprintf( stream, "%s\n", "");
    for (int ss = genome.size()-8; ss <= genome.size()-1; ss++)
        fprintf( stream, "%ld\n", node);
    fprintf( stream, "%s\n", char(fmod("\n", genome.warp(ss)),100));
}
    fprintf( stream, "\n%ld\n", frequencyval);
    fprintf( stream, "%s\n", "");
    fclose(stream);
}

cchar costsfileM[30] = "Costs\n";
cchar trialtypestringM[] = "*";
}.itoa(trialno, trialtypestringM, 10);
strcat(costsfileM, trialtypestringM, .5);
strcat(costsfileM, ".txt\n");
FILE *streamC;
if (streamC = fopen(costsfileM, "a")) == NULL )
    printf("Couldn't open file\n");
else{
    fprintf( streamC, "%s\n", "");
    fprintf( streamC, "%s\n", "");
    fprintf( streamC, "%s\n", "");
    fprintf( streamC, "%s\n", "");
    fprintf( streamC, "%s\n", "");
    fclose(streamC);
}

// Store the best result for each frequency to be compared later
TrackScore = genome.score();
if (ff==0){
    BestScore = genome.score();
    FreqCount = ff;
}
else if (genome.score() < BestScore){
    BestScore = genome.score();
}
FreqCount = 0;
}

//} // OPTION 1 - test all frequencies
}
// OPTION 3 - choose highest feasible value

FILE *streamD;
if((streamD = fopen("BestScores.txt", "a")) == NULL)
    printf("Couldn't open file\n");
else{
    fprintf(streamD, "%sn", "Frequencies:");
    fprintf(streamD, "%sn", "MaxCost: 0, MaxMcValue = 0, MaxPressure = 0, MaxCost = 0, RejectCost = 0, RejectCost2 = 0, RejectCost3 = 0;

Appendix VI
Appendix VI

```c
genome.head();
double score = 0;

FILE *streamcost;
if( (streamcost = fopen("CostData.txt", "rb")) == NULL )
    printf("Couldn't open file\n");
else{
    fseek(streamcost, 0, 0);
    fscanf( streamcost, "%f", &ProcCostK1);
    fscanf( streamcost, "%f", &ProcCostK2);

    fscanf( streamcost, "%d", &BufferCost);
    fscanf( streamcost, "%d", &HeatUCost);
    fscanf( streamcost, "%d", &InjUCost);
    fscanf( streamcost, "%d", &CoolUCost);
    fscanf( streamcost, "%d", &EjUCost);

    fclose(streamcost);
}

/******************************************************************************
** CYCLE THROUGH MOLDS - mold cost calculation
** - maximum cavity per part type calc
*******************************************************************************/

Match = new int[breakpoint-1];
//set all elements to zero
Match[0] = -1; //set the zeroth element to a non zero value so that it is effectively
//ignored. This allows the Match array to be used consistently with the
//indexing in the list of nodes.

for (int init=1; init<=breakpoint - 1; init++){
    Match[init] = 0;
}

for (int indexa= 1; indexa <= breakpoint - 1; indexa++){   
    if (Match[indexa] == 0){
        Match[indexa] = 1;
        for (int indexb= indexa+1; indexb <= breakpoint - 1; indexb++){  
            if (Match[indexb] == 0){
                if (((fmod("genome.warp(indexa),100)-64) ==
                    (fmod("genome.warp(indexb),100)-64))&&
                    (floor("genome.warp(indexa)/100) ==
                        (\"genome.warp(indexb)/100))){
                    Match[indexa]++;
                    Match[indexb]--;
                }
            }
        }
    }
    
}

for(int i=1; i <= breakpoint-1; i++){   // cycle through list and look at nodes
    ProductNo = fmod("genome.warp(i)),100)-64;  //fmod("genome.warp(i),100)-64 is the product number

    // ******** MOLD COST CALCULATION *********
    if (Match[i] != -1){
        moldcost = moldcost + (MoldBaseCCost[ProductNo-1] * 
            pow(floor("genome.warp(i)/100),0.7)) * pow(Match[i],0.7);
```
m
MACHINE
SIZE

CALCULATION

int TotalProjArea = 0;

// ******** TOTAL PROJECTED SHOT AREA ********
for(int j=1; j<=numparts; j++){
    int CalcTotalProjArea = ProjectedAreaPart[j-1] * MaxNoCavs[j-1]; //TotalAreaPerPart is a function
    if (CalcTotalProjArea > TotalProjArea)
        TotalProjArea = CalcTotalProjArea;
}

// ***** MAXIMUM CAVITY PRESSURE *****

int infotypelnjPressure = 5; // Inj. Pressure data appears in the 6th column in materialdatafile
for (int k=0; k<numparts; k++)
    getInfromaterial(Material[k], infotypelnjPressure, k, InjPressure); //injPressure in bars

if (InjPressure[k] > MaxInjPressure)
    MaxInjPressure = InjPressure[k];

MaxCavPressure = 0.5 * MaxInjPressure * 100000; // N/m²

// ***** MAX SEPARATING FORCE *******
MaxSeparatingF = TotalProjArea * 0.0001 * MaxCavPressure / 1000; // TotalProjArea in cm², Max Cav Pressure in N/m², MaxSeparatingForce in kN

// ***** CLAMP FORCE **********

MachineSize = 1;
int MachineDatatype = 0; // Clamp Force is the leftmost column
ClampFValue = getInfromachine(MachineDatatype, MachineSize); // read data from column 1, row 1 (info type range starts at 0, machinesize range starts at 1)

while (ClampF != ClampFValue){
    if (MaxSeparatingF < ClampFValue){
        // ******** Reject mold Combo ********
        ClampF = ClampFValue; //rounded to the next machine size, from machinedata.txt
    }
    else if (MaxSeparatingF >= 8500){ //8500kN max clamping force
        ClampF = 8500;
        rejectcost1 = 1E15;//acts as a reject cost
        ClampFValue = ClampF;
        MachineSize = 7;
    }
    else{
        // ******** read next value
}
MachineSize++;  
ClampFValue = getinfomachine(MachineDatatype, MachineSize);  
if (ClampFValue == MaxSeparatingF){  
    MaxSeparatingF = MaxSeparatingF + 1;  
}

//cout << "MaxSepF:" << MaxSeparatingF=" << "ClampF:" << ClampF;  

/******************************************************************************  
** CYCLE TIME CALCULATIONS **  
*******************************************************************************/  

float (*cydetimes)[4];  
cydetimes = new float[MachineSize][4];  
int PowerDatatype = 5;  // Driving power in the sixth column (0,1,2,3,4,5)  
float DrivingPower = getinfomachine(PowerDatatype, MachineSize);  //in kW  
float CoolTime = 0;  
int Diffusivitytype = 1, InjectTemptype = 2, MoldTemptype = 3, EjectTemptype = 4;  
InjectTemp = new int[numparts];  
MoldTemp = new int[numparts];  
EjectTemp = new int[numparts];  
float *Diffusivity;  
Diffusivity = new float[numparts];  
double pi = 3.145926535;  
int DryCttype = 3, MaxClampStype = 4;  
float DryCycleTime, MaxClampStroke;  
for (int i=0, r<numparts; r++){  
// Injection Time (Single Cavity Mold)  
cydetimes[r][1] = 2* (PartVol[0]*0.000001) * (InjPressure[r]*100000)/(DrivingPower*1000);  //[r][1] => row r, column 2  
// Cooling Time and Heating Time  
getinfomaterial(Material[r], Diffusivitytype, r, Diffusivity);  
getinfomaterial(Material[r], InjectTemptype, r, InjectTemp);  
getinfomaterial(Material[r], MoldTemptype, r, MoldTemp);  
getinfomaterial(Material[r], EjectTemptype, r, EjectTemp);  
//cooling time  
CoolTime = pow(MaxThickness[r],2)/pow(pi,2)*Diffusivity[r]*log((4*(InjectTemp[r] - MoldTemp[r])/pi*(EjectTemp[r] - MoldTemp[r])));  
if (CoolTime >= 3){  
cydetimes[r][2] = CoolTime;  
}  
else if (CoolTime <3){  
cydetimes[r][2] = 3;  // 3 sec minimum  
}  
//heating time = % of cooling time  
cydetimes[r][0] = 0.5 * cydetimes[r][2];  
// Mold-Reset Time  
DryCycleTime = getinfomachine(DryCttype, MachineSize); //seconds
MaxClampStroke = getinformachine(MaxClampStype, MachineSize); //cm

cycletimes[r][3] = 1+1.75*DryCycleTime*pow(((2*PartDepth[r]+5)/MaxClampStroke),0.5);

/**
  \* LONGEST CYCLE TIME CALCULATIONS
  \*/
  
for(int t=1; t <= breakpoint-1; t++){
    // cycle through list and look at nodes
    //getinfodata(fmod("genome.warp(i),100)-64, cycletimes); //fmod("genome.warp(i),100)-64 is the
    product number
    int partnumber = fmod("genome.warp(t),100)-65;
    //cycletimes[fmod("genome.warp(i),100)-64][0] = cycletimes[fmod("genome.warp(i),100)-64][0]*floor("genome.warp(i)/100); // based on heating time
    cycletimes[partnumber][1] = cycletimes[partnumber][1]*pow(\"genome.warp(t)/100\", 0.7); // based on injection time (NB the original inj time is calculated for a single cavity mold)
    
    the injection time is not simply multiplied by the NoCav, but rather multiplied by
    
    NoCav raised to the power of 0.7.
    //cycletimes[fmod("genome.warp(i),100)-64][2] = cycletimes[fmod("genome.warp(i),100)-64][2]*floor("genome.warp(i)/100); // based on cool time
    //cycletimes[fmod("genome.warp(i),100)-64][3] = cycletimes[fmod("genome.warp(i),100)-64][3]*floor("genome.warp(i)/100); // based on ejection time
    
    for(int pp=0; pp < 4; pp++){
      if (floor("genome.warp(breakpoint + 2*pp)/100)+floor("genome.warp(breakpoint+1+ 2*pp)/100)
        < (genome.size() -8-1)) //the number of nodes is larger than the station capacity
        if (cycletimes[partnumber][pp]/(floor("genome.warp(breakpoint+
        2*pp)/100)+floor("genome.warp(breakpoint+1+ 2*pp)/100)) > longestCT) //compare the cyclctime/station capacity to
        longestCT = cycletimes[partnumber][pp]/(floor("genome.warp(breakpoint+
        2*pp)/100)+floor("genome.warp(breakpoint+1+ 2*pp)/100));
      } else if (floor("genome.warp(breakpoint + 2*pp)/100)+floor("genome.warp(breakpoint+1+ 2*pp)/100) > (genome.size() -8-1)) // the station capacity is larger than the number of nodes
        if (cycletimes[partnumber][pp]/(genome.size() -8-1) > longestCT) //compare the cycltime/number of nodes to the longestCT
          longestCT = cycletimes[partnumber][pp]/(genome.size() -8-1);
      }
    }
  }
  
for (int counta = 0; counta<numparts; counta++){
  cout << MaxNoCavs[counta] << " / ";
}
  cout << "n";

(['/
** TEST TO SEE IF THE LONGEST CT IS LESS THAN THE CYCLE TAKT TIME **

```cpp
int CycleTaktTime;
int AvailableTime = 24; // available hours per day

CycleTaktTime = (AvailableTime*60*60/frequency); // in seconds
if (longestct > CycleTaktTime)
    rejectcost2 = 1E15;
else if (longestct < CycleTaktTime)
    rejectcost2 = 0;
```

** PROCESSING COST CALCULATION - BEST AND WORST CASE **

```cpp
// PROCESSING COST - Linear
processingcost = 1*(ProcCostK1+(ProcCostK2*ClampF))*AvailableTime*250; // 250 days per year [use k1 = 25.103, k2=0.0075]
    // ^
    // | WORST CASE SCENARIO

// processingcost = (ProcCostK1+(ProcCostK2*MaxSeparatingF))*AvailableTime*250; // 250 days per year [uses MaxSeparating force as the ClampF]

// PROCESSING COST - Exponential
// processingcost = (ProcCostK1*exp(ProcCostK2*ClampF))*AvailableTime*250; // 250 days per year [use k1 = 33.801, k2=0.0001]

// processingcost = (ProcCostK1*exp(ProcCostK2*MaxSeparatingF))*AvailableTime*250; // 250 days per year [uses MaxSeparating force as the ClampF]
```

** ADDITIONAL MACHINE COST CALCULATION **

```cpp
int TotalBuffer = 0;
TotalBuffer = floor("genome.warp(breakpoint)/100") + floor("genome.warp(breakpoint+2)/100")
    + floor("genome.warp(breakpoint+4)/100") + floor("genome.warp(breakpoint+6)/100");
// Addition of all the buffers i.e. stations S, U, W, and Y

addmachinecost = TotalBuffer*BufferCost + floor("genome.warp(breakpoint+1)/100")*HeatUCost
    + floor("genome.warp(breakpoint+3)/100")*InjUCost + floor("genome.warp(breakpoint+5)/100")*CoolUCost
    + floor("genome.warp(breakpoint+7)/100")*EjUCost; // Heating unit cost (capacity of 1 each) per year

addmachinecost = BestWorst*addmachinecost; // WORST CASE SCENARIO
    // ^
    // | WORST CASE SCENARIO
```

** CHECK FOR DEADLOCK **

```cpp
```
int TotalNumberOfStations = 0;
for (int p = breakpoint; p <= size-1; p++)
{
    TotalNumberOfStations = TotalNumberOfStations + floor(genome.warp(p)/100);
}
if (breakpoint > TotalNumberOfStations)
    rejectcost3 = 1E15;
else if (breakpoint <= TotalNumberOfStations)
    rejectcost3 = 0;

/**
 * AMORTIZATION
 */

float interest = 0.1;
int moldperiod = 3, machineperiod = 5;
moldcost = moldcost*(interest*pow(1+interest,moldperiod))/(pow((interest+1),moldperiod)-1);
addmachinecost = addmachinecost*(interest*pow(1+interest,machineperiod))/(pow((interest+1),machineperiod)-1);

/**
 * FIND SCORE
 */

score = moldcost + processingcost + addmachinecost + rejectcost1 + rejectcost2 + rejectcost3;

++longestc << " ";
char costsfile[30] = "Costs";
int trialtypes = trialex;
char trialtypestring[] = " ";
_itoa(trialtype, trialtypestring, 10);
strcat(costsfile, trialtypestring, 5);
strcat(costsfile, ".txt ");
//printf(costsfile);
FILE *streamC;
if ((streamC = fopen(costsfile, "a" )) == NULL )
    printf("Couldn't open file\n ");
else{
    fprintf( streamC, "%g\n", moldcost);
    fprintf( streamC, "%g\n", processingcost);
    fprintf( streamC, "%g\n", addmachinecost);
    fprintf( streamC, "%g\n", rejectcost1);
    fprintf( streamC, "%g\n", rejectcost2);
    fprintf( streamC, "%g\n", rejectcost3);
    fprintf( streamC, "%g\n", score);
    for (int parti = 0; parti<numparts; parti++)
        fprintf( streamC, "%d\n", cycletimes[pariti][0]);
    fprintf( streamC, "%d\n", cycletimes[pariti][1]);
    fprintf( streamC, "%d\n", cycletimes[pariti][2]);
    fprintf( streamC, "%d\n", cycletimes[pariti][3]);
}
}
/* List Genome Operators */

The initializer creates a list with n elements in it and puts a unique digit in each one. After we make the list, we scramble everything up.

void ListInitializer(GAGenome & c)
{
    GAList Genome<int> & child = GAList Genome<int> & c;
    while (child.head()) child.destroy(); // destroy any pre-existing list

    int *ProductionV;
    ProductionV = new int[numparts];
    for (int qq = 0; qq < numparts; qq++)
    {
        ProductionV[qq] = ProductionVolumes[qq]/(frequencyval); // Frequency will always take the value of 1, and usually others
    }

    // int max num cavities = 15; // max number of cavities per mold
    int max capacity = 10; // max station capacity
    int counter = 0; // counts the number of nodes in the list
    int index = 0; // index for entering values
    child.insert(0, GAListBASE::HEAD); // the head node contains a '0'
    counter++; // start counting the size of the list
    for (int i = 1; i < numparts; i++)
    {
        // each subsequent node contains a letter representing the part type and a numeric value for the number of cavities in the mold.
        while (ProductionV[i-1] != 0)
        {
            int insertvol = GARandomInt(1, ProductionV[i-1]);
            child.insert(insertvol*100 + 64 + i);
            counter++; // add one to the size of the list after a new node is added
            ProductionV[i-1] = ProductionV[i-1] - insertvol;
        }
        // child.insert(ProductionV[i-1]*100 + 64 + i);
        // child.insert(GARandomInt(1, max num cavities)*100+(65+GARandomInt(0,numparts-1)));
    }

    int size = counter;
    int breakpoint = counter-8;

    for (int j = breakpoint; j <= size-1; j++)
    {
        // each node after the breakpoint contains a letter ranging from S(83) to Z(90) representing the fact that the list element holds the value for the station capacity.
        // child.insert(max capacity*100 + 83 + index); // max capacity
        child.insert(GARandomInt(1, max capacity)*100 + 83 + index); // Random (1, max capacity)
```c
for(int k=1; k<=breakpoint-1; k++) { // scramble everything up (until breakpoint)
    child.swap(GARandomInt(1, breakpoint-1), GARandomInt(1, breakpoint-1));
}

int SplitJoinMutator(GAGenome & c, float pmut)
{
    GAListGenome<int> &child=DYN_CAST(GAListGenome<int> &, c);
    register int n, i, oldvalue, oldletter, randomval, newnode1, newnode2, currentletter;
    register int currentnumber, nextnumber, newnode, splitjoinvar, indecvar, jump, randomjump;

    jump = 10; // ***** Set the jump limit for the increase and decrease mutator *****
    // ***********************************************************************

    if(pmut <= 0.0) return 0;

    n = child.size()-8; // final eight nodes reserved for station capacities
    float nMut = pmut * STA_CAST(float, n);
    if(nMut < 1.0){ // we have to do a flip test for each node
        nMut = 0;
        for(i=0; i<n; i++)
            if(GAFlipCoin(pmut) && child.warp(i))
                // randomly select a node vs. visit node i
                child.warp(i);

        splitjoinvar = GARandomInt(0,1);
        if (splitjoinvar == 0) { // SPLIT
            if (floor(*child.current()/100) > 1)
            {
                cout << "splitjoinvar == 0, not splitting

                oldvalue = floor(*child.current()/100);
                oldletter = fmod(*child.current(), 100);
                /* OPTION 1 - random division */
                randomval = GARandomInt(1, oldvalue-1);

                /* OPTION 2 - Even division (Odd e.g. 9=> 4+5)
                if (oldvalue % 2 == 0) { //even
                    randomval = oldvalue/2; //even division, NOT random
                }
                else if (oldvalue % 2 != 0) { //odd
                    randomval = floor(oldvalue/2); // makes the # cav even
                }
            }

            /* OPTION 3 - Even division, Odd values become even +1 e.g. 9=> 8+1
            if (oldvalue % 2 == 0) { //even
            */
        }
}
```
randomval = oldValue/2;       //even division, NOT random
}
else if (oldValue % 2 != 0){  //odd
    randomval = 1;           // makes the # cav even
}

newnode1 = randomval * 100 + oldletter;
newnode2 = (oldValue - randomval) * 100 + oldletter;
child.destroy();
child.insert(newnode1, GListBASE::AFTER);
child.insert(newnode2, GListBASE::AFTER);

n = child.size()-8;  // final eight nodes reserved for station capacities
child.swap(GARandomInt(1,n-1),GARandomInt(1,n-1)); //Swap nodes around
nMut++;
}

} else if (splitpinvar == 1){  // JOIN
    if (fmod(*child.current(),100) == fmod(*child.next(),100)){
        cout << "1";
        currentletter = fmod(*child.prev(),100);
        currentnumber = floor(*child.current())/100;
        nextnumber = floor(*child.next())/100;
        newnode = ((currentnumber + nextnumber)*100 + currentletter);
        child.prev();
        child.destroy();
        child.next();
        child.destroy();
        child.insert(newnode, GListBASE::AFTER);
        n = child.size()-8;  // final eight nodes reserved for station capacities
        child.swap(GARandomInt(1,n-1),GARandomInt(1,n-1)); //Swap nodes around
        nMut++;
    }

    n = child.size()-8;
    child.swap(GARandomInt(n,child.size()-1)); //INCREASE/DECREASE
    incdecvar = GARandomInt(0,1);
    currentnumber = floor(*child.current())/100;
    currentletter = fmod(*child.current(),100);
    if (incdecvar == 0){  //INCREASE
        newnode = (currentnumber + GARandomInt(1,jump))*100 + currentletter; //increase by
    Random (1-jump), jump is set above
        //newnode = (currentnumber + 1)*100 +currentletter;   //increase by 1
        child.destroy();
        child.insert(newnode, GListBASE::AFTER);
    }
    else if (incdecvar == 1){  //DECREASE
        randomjump = GARandomInt(1,jump); //needed for randomizing the decreasing jump
        if (currentnumber > randomjump){
            newnode = (currentnumber - randomjump)*100 + currentletter; //decrease by
    Random (1-jump), jump is set above
        //newnode = (currentnumber - 1)*100 + currentletter;   //decrease by 1
        child.destroy();
        child.insert(newnode, GListBASE::AFTER);
    }
}
dse(n); // only make the number of nodes we need to

for(i=0; i<nMut; i++)
    n = child.size()-8;

    // randomly select a node vs.
    // visit node i

    child.warp(GARandomInt(1, n-1));
    child.warp();

    splitjoinvar = GARandomInt(0, 1);

    if (splitjoinvar == 0) { //SPLIT

        if (floor(*child.current())/100) > 1){
            cout << "S2\n";
            oldvalue = floor(*child.current())/100;
            oldletter = fmod(*child.current(), 100);
            /* OPTION 1 - random division */
            randomval = GARandomInt(1, oldvalue-1);

            /* OPTION 2 - Even division (Odd e.g. 9=>4+5)
             if (oldvalue % 2 == 0){
                even randomval = oldvalue/2; //even division, NOT random
            } else if (oldvalue % 2 != 0){ //odd
                randomval = floor(oldvalue/2); // makes the # cavs even
            }
            */

            /* OPTION 3 - Even division, Odd values become even +1 e.g. 9=>8+1
             if (oldvalue % 2 == 0){
                even randomval = oldvalue/2; //even division, NOT random
            } else if (oldvalue % 2 != 0){ //odd
                randomval = 1; // makes the # cavs even
            }
            */
            newnode1 = randomval * 100 + oldletter;
            newnode2 = (oldvalue - randomval) * 100 + oldletter;
            child.destroy();
            child.insert(newnode1, GASTInt::AFTER);
            child.insert(newnode2, GASTInt::AFTER);
        }
        n = child.size()-8; // final eight nodes reserved for station capacities
        child.swap(GARandomInt(1, n-1), GARandomInt(1, n-1)); //Swap nodes around

    } else if (splitjoinvar == 1){ //JOIN
        if (fmod(*child.current(), 100) == fmod(*child.next(), 100)) {
            cout << "2\n";
            currentletter = fmod(*child.prev(), 100);
            currentnumber = floor(*child.current()/100);
            nextnumber = floor(*child.next()/100);
            newnode = ((currentnumber + nextnumber)*100 + currentletter);
            child.prev();
            child.destroy();
        }
    }
child.next();
child.destroy();
child.insert(newnode, GAListBASE::AFTER);
n = child.size()-8;  // final eight nodes reserved for station capacities
child.swap(GARandomInt(1,n-1),GARandomInt(1,n-1));  //Swap nodes around
}

n = child.size()-8;
child.warp(GARandomInt(n,n+8-1));  //INCREASE/DECREASE
incdecvar = GARandomInt(0,1);
currentletter = fmod("child.current()",100);
currentnumber = floor("child.current()"/100);
if (incdecvar == 0){  //INCREASE
    cout << "" << "n"
    newnode = (currentnumber + GARandomInt(1,jump))"100 +currentletter; //increase by
Random (1-jump), jump is set above
    //newnode = (currentnumber + 1)*100 +currentletter;  //increase by 1
    child.destroy();
    child.insert(newnode, GAListBASE::AFTER);
}
else if (incdecvar == 1){  //DECREASE
    cout << "" << "n"
    randomjump = GARandomInt(1,jump);  //needed for randomizing the decreasing jump
    if (currentnumber > randomjump){
        newnode = (currentnumber - randomjump)*100 +currentletter; //decrease by
Random (1-jump), jump is set above
        //newnode = (currentnumber - 1)*100 +currentletter;  //decrease by 1
        child.destroy();
        child.insert(newnode, GAListBASE::AFTER);
    }
}
//cout << "..." << n << "."
}

child.head();  // set iterator to root node
return(STA_CAST(int, nMut));

// Here we specialize the write method for the List class. This lets us see
// exactly what we want (the default write method dumps out pointers to the
// data rather than the data contents).
// This routine prints out the contents of each element of the list,
// separated by a space. It does not put a newline at the end of the list.
// Notice that you can specialize ANY function of a template class, but
// some compilers are more picky about how you do it than others. For the
// metrowerks compiler this specialization must come before the forced
// instantiation.
int
GAListGenome<int>::write(ostream & os) const
{
    int *cur, *head, listsize = 1, totalend = 1;
    GAListIter<int> tmpiter("this");
    if((head=tmpiter.head()) != 0)
        os << floor(*head/100) << char(fmod(*head,100)) << ".";
for (cur = tmpiter.next(); cur && cur != head; cur = tmpiter.next())
{
    os << floor(*cur/100) << char(mod(*cur, 100)) << " ";
    listsize ++;
    totalend += (floor(*cur/100));
}
cout << "in" << listsize << " ";

return os.fail() ? 1 : 0;
}

int declaration()
{
    ProjectedAreaPerPart = new float[numparts];
    Material = new float[numparts];
    PartVol = new float[numparts];
    MaxThickness = new float[numparts];
    PartDepth = new float[numparts];
    PartLength = new float[numparts];
    PartWidth = new float[numparts];
    //float *AreaMoldBasePlate, *CorCavThicknes;
    AreaMoldBasePlate = new float[numparts];
    CorCavThickness = new float[numparts];
    Nspi = new int[numparts];
    Nspo = new int[numparts];
    Nhd = new int[numparts];
    Nspull = new int[numparts];
    Nilift = new int[numparts];
    Nunscrew = new int[numparts];
    NpullscrewSides = new int[numparts];
    NscrewRear = new int[numparts];
    SFinish = new float[numparts];
    Tolerance = new float[numparts];
    Texture = new float[numparts];
    PartPlane = new float[numparts];
    // Nspi = number of inner surface patches
    // Nspo = number of outer surface patches
    // Nhd = number of holes and depressions
    // Nspull = number of side pulls
    // Nilift = number of internal lifters
    // Nunscrew = number of unscrew devices
    // NpullscrewSides = number of sides of part requiring side pulls or unscrew devices
    // 11 = left or right 12 = top or bottom
    // 21 = left and right 22 = top and bottom 23 = (left or right) and (top or bottom)
    // 31 = left and right and (top or bottom) 32 = top and bottom and (left or right)
// 40 = left and right and top and bottom
// Sfinish = surface finish
// Tolerance = tolerance
// Texture = surface texture
// PartPlane = parting plane factor

Me = new float[numparts];
Mx = new float[numparts];
Xi = new float[numparts];
Xo = new float[numparts];
Mspull = new float[numparts];
Munscrew = new float[numparts];
Msfinish = new float[numparts];
Mtol = new float[numparts];
Mtexture = new float[numparts];
Mpartplane = new float[numparts];
Mtotal = new float[numparts];

//int *ProductionVolumes;
ProductionVolumes = new int[numparts];
//float *MoldBaseCost, *CCost, *MoldBaseCCCost;
MoldBaseCost = new float[numparts];
CCost = new float[numparts];
MoldBaseCCCost = new float[numparts];

//int *InjPressure;
InjPressure = new int[numparts];

//int MachineSize = 1;

// Xi = inner complexity
// Xo = outer complexity
// Me = manufacturing hours due to ejector pins
// Mx = manufacturing hours due to geometrical features
// Mpo = manufacturing hours due to size
// Mspull = manufacturing hours due to side pulls
// Mlifit = manufacturing hours due to internal lifter
// Munscrew = manufacturing hours due to unscrew devices
// Msfinish = manufacturing hours due to surface finish
// Mtol = manufacturing hours due to tolerance
// Mtexture = manufacturing hours due to surface texture
// Mpartplane = manufacturing hours due to parting plane factor

return 0;

/*****************************/
** Here we read part geometry information from respective partfiles.  **
** These files are labeled "part1.txt", "part2.txt", etc.  **
***************************************************************************/

// ***** Read values from file *****
int partgeometryevaluator(int trial, int parttype){
  int dumbvar;
char filename[30] = "part";
int type = trial;
char typestring[] = "";
int ptype = parttype;
char ptypestring[] = "";
float(10, type, typestring, 10);  
float(10, ptype, ptypestring, 10);
strcat(filename, typestring, 5);
strcat(filename, ptypestring, 5);
strcat(filename, ".");
printf(filename);
FILE *stream;
if (stream = fopen(filename, "rb")) == NULL)
    printf("Couldn't open file\n");
else{
    fseek(stream, 0, 0);
fscanf (stream, "%d", &dumbvar);
    if (dumbvar != parttype)
        printf("Incorrect file or format\n");
        return(0);
}
fscanf (stream, "%f", &1ProjectedAreaPart[parttype - 1]);
fscanf (stream, "%f", &Material[parttype - 1]); // Value 1 - 11
fscanf (stream, "%f", &PartVol[parttype - 1]); // Incl runners - cm3
fscanf (stream, "%f", &MaxThickness[parttype - 1]); // mm
fscanf (stream, "%f", &PartDepth[parttype - 1]); // cm
fscanf (stream, "%f", &PartLength[parttype - 1]); // cm
fscanf (stream, "%f", &PartWidth[parttype - 1]); // cm
fscanf (stream, "%d", &Nsp[parttype - 1]);
fscanf (stream, "%d", &Nsp[parttype - 1]);
fscanf (stream, "%d", &Nsp[parttype - 1]);
fscanf (stream, "%d", &Nspull[parttype - 1]);
fscanf (stream, "%d", &Nspull[parttype - 1]);
fscanf (stream, "%d", &Nspull[parttype - 1]);
fscanf (stream, "%d", &Nspull[parttype - 1]);
fscanf (stream, "%d", &Nspull[parttype - 1]);
fscanf (stream, "%f", &1Project[parttype - 1]);
fscanf (stream, "%f", &1FillVolume[parttype - 1]);
fscanf (stream, "%f", &1Finish[parttype - 1]);
fscanf (stream, "%f", &1Tolerance[parttype - 1]);
fscanf (stream, "%f", &1Texture[parttype - 1]);
fscanf (stream, "%f", &1PartPlane[parttype - 1]);

fscanf (stream, "%d", &ProductionVolumes[parttype - 1]);
}
fclose(3ream);

// Single Cavity Mold Base Cost

// Area of Mold Base Cavity Plate
// side pull or screwing device (top, bottom, left, right) => double clearance
float ClearanceLength = 15, ClearanceWidth = 15; // cm (7.5cm on each side)

if (NpullScrewSides[parttype - 1] == 11) {
    ClearanceWidth = ClearanceWidth + 7.5;
}
else if (NpullscREW[Sides][parttype - 1] == 12) {
    ClearanceLength = ClearanceLength + 7.5;
}
}
else if (NpullscREW[Sides][parttype - 1] == 21) {
    ClearanceWidth = ClearanceWidth + 15;
}
else if (NpullscREW[Sides][parttype - 1] == 22) {
    ClearanceLength = ClearanceLength + 15;
}
else if (NpullscREW[Sides][parttype - 1] == 23) {
    ClearanceWidth = ClearanceWidth + 7.5;
    ClearanceLength = ClearanceLength + 7.5;
}
else if (NpullscREW[Sides][parttype - 1] == 31) {
    ClearanceWidth = ClearanceWidth + 15;
    ClearanceLength = ClearanceLength + 7.5;
}
else if (NpullscREW[Sides][parttype - 1] == 32) {
    ClearanceWidth = ClearanceWidth + 7.5;
    ClearanceLength = ClearanceLength + 15;
}
else if (NpullscREW[Sides][parttype - 1] == 40) {
    ClearanceWidth = ClearanceWidth + 15;
    ClearanceLength = ClearanceLength + 15;
}

AreaMoldBasePlate[parttype - 1] = (PartLength[parttype - 1] + ClearanceLength) * (PartWidth[parttype - 1] + ClearanceWidth);

    // Thicknes
    // screwing device (behind) => double clearance
    float ClearanceDepth = 15; // cm (7.5 cm in front and 7.5 cm behind part)
    // NscrewRear = rear unscrew devices
    // 1 = present  0 = NOT present
    if (NscrewRear[parttype - 1] == 1) {
        ClearanceDepth = ClearanceDepth + 7.5;
    }

    CorCavThickness[parttype - 1] = (PartDepth[parttype - 1] + ClearanceDepth);

    /* N.B. the increase to the clearance spaces of 0.5 cm for every 100 cm2 of Total Area is ignored in this case due to the fact that the costs for a SINGLE cavity mold are calculated. This is then used as the basis for a multicavity mold */

    MoldBaseCost[parttype - 1] = 1000 * 0.45 * AreaMoldBasePlate[parttype - 1] * pow(CorCavThickness[parttype - 1], 0.4);

    // **Single Core and Cavity Cost**

        // Part Complexity
        // inner complexity
        X1[parttype - 1] = 0.01*Nspill[parttype - 1] + 0.04*Nhd[parttype - 1];

        // outer complexity
        X0[parttype - 1] = 0.01*Nsp1[parttype - 1];

    // Ejector pins
    Me[parttype - 1] = pow(ProjectedAreaPart[parttype - 1], 0.5)*2.5;
Appendix VI

// Geometrical Features
Mx[partype - 1] = 45^*pow(X[partype - 1] + Xo[partype - 1], 1.27);

// Size
Mpo[partype - 1] = 5 + 0.085^*pow(ProjectedAreaPart[partype - 1], 1.2);

// Side pull
Mspull[partype - 1] = Nspull[partype - 1] * 65;

// Internal lifters
Mllift[partype - 1] = Nllift[partype - 1] * 150;

// Unscrewing devices
Mnunscrew[partype - 1] = Nnunscrew[partype - 1] * 250;

// Surface finish
// find SFinish

// Tolerance
Mtol[partype - 1] = Tolerance[partype - 1] * Mx[partype - 1];

// Texture

// Parting Plane
Mpartplane[partype - 1] = PartPlane[partype - 1] * pow(ProjectedAreaPart[partype - 1], 0.5);

// * Total number of manufacturing hours *

// * Core and Cavity Cost *
CCCost[partype - 1] = Mtotal[partype - 1] * MachiningCost;

// * Total Single Mold Base and Core & Cavity Cost *
MdbaseCCCost[partype - 1] = MoldBaseCost[partype - 1] + CCCost[partype - 1];

return 0;

// Here we define the function that gets INTEGER information from the material datafile
// regarding injection pressures etc. based on the materialtype.

int getinfomaterial(int materialtype, int infotype, int counter, int InfoType[]){
  FILE *stream;
  float testfloat = 0, dumbvar = 0;
  if( (stream = fopen( "materialdata.txt", "rb" )) == NULL )
    printf( "Couldn't open fileIn" );
  else{
    fseek(stream, 0, 0);
    while (!feof(stream)){
      fscanf( stream, "%f", &testfloat);
    }
  }
  return 0;
}
if (testfloat == -materialtype){
    for (int m = 1; m <= infotype; m++){
        fscanf (stream, "%f", &dumbvar);
    }
    fscanf (stream, "%d", &InfoType[counter]);
    break;
}
else if (testfloat == -9999){
    break;
}
}
fclose(stream);
}
return 0;

// Here we define the function that gets FLoAT information from the material datafile
// regarding injection pressures etc. based on the material type.
float getinformaterial(int materialtype, int infotype, int counter, float InfoType[]){
    FILE *stream;
    float testfloat = 0, dumbvar = 0;
    if (stream == fopen("materialdata.txt", "rb")) == NULL )
        printf("Couldn't open file\n");
    else{
        fseek(stream, 0, 0);
        while (!feof(stream)){
            fscanf (stream, "%f", &testfloat);
            if (testfloat == -materialtype){
                for (int m = 1; m <= infotype; m++){
                    fscanf (stream, "%f", &dumbvar);
                }
                fscanf (stream, "%d", &InfoType[counter]);
                break;
            }
            else if (testfloat == -9999){
                break;
            }
        }
    }
    fclose(stream);
}
return 0;

// Here we define the function that gets information from the machine datafile
// regarding Clamping Forces, Processing Costs, etc. based on the machine type.
float getinformachinem(int infotype, int machinesize){
    FILE *stream;
    float dumbvar = 0;
    float machinedatatavar = 0;
    if (stream == fopen("machinedata.txt", "rb")) == NULL )
        printf("Couldn't open file\n");
    else{
        fseek(stream, 0, 0);
        for (int hh = 0; hh < infotype; hh++){
```c
fscanf (stream, "%f", &numbvar);
}
if (machinewidth == 1){
fscanf (stream, "%f", &machinedatavar);
fclose(stream);
return (machinedatavar);
} else {
    for (int jj = 1; jj <= (machinewidth-1); jj++){
        fscanf (stream, "%f", &numbvar);
    }
    fscanf (stream, "%f", &machinedatavar);
}
fclose(stream);
return (machinedatavar);
}

// Here we define the function that finds the greatest common divisor or GCD

int GCD(int m, int n){
    if (m%n==0){
        return n;
    } else{
        return GCD(n, m%n);
    }
}

// This function returns the number of possible frequencies given the GCD

int FindNoFreq(int gcd){
    int count = 0;
    for (int fnf = 1; fnf <= gcd; fnf++){
        if (gcd%fnf == 0){
            count = count + 1;
        }
    }
    return count;
}

// This function fills the Frequency array with the possible frequencies given the GCD

int FrequencyFill(int gcd){
    int FreqCount = 0;
    for (int hh = 1; hh <= gcd; hh++){
        if (gcd%hh == 0){
            Frequency[FreqCount] = gcd/hh;
            FreqCount = FreqCount + 1;
        }
    }
    return 0;
}
```
// If your compiler does not do automatic instantiation (e.g. g++ 2.6.8),
// then define the NO_AUTO_INST directive.
#define NO_AUTO_INST
#include <ga/GAList.C>
#include <ga/GAListGenome.C>
#if defined(__GNUC__)
template class GAList<int>;
template class GAListGenome<int>;
#else
GAList<int>;
GAListGenome<int>;
#endif
#endif
10. References


Cochran, D. S., 1999, “Production System Design Course,” MIT.


Schut, J. H., 1999, “Why are these men smiling?” Plastics Technology, 45/1:100.


