ADVANCED MOTION COMMAND GENERATION PARADIGMS FOR CNC SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ULAŞ YAMAN

IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
MECHANICAL ENGINEERING

JUNE 2014

Approval of the thesis:

ADVANCED MOTION COMMAND GENERATION PARADIGMS FOR CNC SYSTEMS

submitted by **ULAŞ YAMAN** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Mechanical Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Suha Oral _____
Head of Department, **Mechanical Engineering**

Assoc. Prof. Dr. Melik Dölen _____
Supervisor, **Mechanical Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. Tuna Balkan _____
Mechanical Engineering Dept., METU

Assoc. Prof. Dr. Melik Dölen _____
Mechanical Engineering Dept., METU

Assist. Prof. Dr. A. Buğra Koku _____
Mechanical Engineering Dept., METU

Assoc. Prof. Dr. A. Özgür Yılmaz _____
Electrical and Electronics Engineering Dept., METU

Assist. Prof. Dr. Kutluk Bilge Arıkan _____
Mechatronics Engineering Dept., Atılım University

Date: 09.06.2014

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as require by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name : Ulaş YAMAN

Signature :

ABSTRACT

ADVANCED MOTION COMMAND GENERATION
PARADIGMS FOR CNC SYSTEMS

Yaman, Ulaş

Ph.D., Department of Mechanical Engineering
Supervisor: Assoc. Prof. Dr. Melik Dölen

June 2014, 203 pages

A novel motion command generation paradigm for digital motion control systems is developed with in the scope of this dissertation. In the paradigm, the tool trajectory is firstly defined with the developed programming language on a host computer and then transferred to the machine with different communication protocols. The language proposed is capable of decompressing the previously compressed motion data via ΔY10 decompression algorithm and generating curve offsets of the base curve in inner and outer directions. With these abilities of the programming language and its hardware processor (VEPRO), the tool trajectory of a machining case can be presented with a few lines of commands. The hardware complexity of the VEPRO is low compared to the ones currently used in computer numerical systems such as Siemens Sinumerik and Fanuc 0*i*.

Keywords: Command Generation, Data Compression, FPGA, Servo-motor Drives, Computer Numerical Control, Curve Offset Generation, Morphological Operations, Polygon Operations

ÖZ

BİLGİSAYAR DENETİMLİ SİSTEMLER İÇİN GELİŞMİŞ
HAREKET KOMUT ÜRETECİ ÖRNEKLERİ

Yaman, Ulaş

Doktora, Makina Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Melik Dölen

Haziran 2014, 203 sayfa

Doktora tezi kapsamında sayısal hareket denetleyici sistemleri için yeni bir hareket komut üreteç modeli geliştirilmiştir. Geliştirilen modelde, öncelikli olarak komut yörüngesi önerilen programlama dili kullanılarak bilgisayar ortamında oluşturulur. Sonrasında ise derlenen makina kodu farklı haberleşme yöntemleri aracılığıyla komut üreteci donanımına gönderilir. Önerilen programlama dili, ΔY10 sıkıştırma algoritması kullanılarak sıkıştırılmış olan komut verilerini çözebilme ve verilen temel eğrileri farklı yönlerde kaydırabilme yeteneklerine sahiptir. Program dilinin bu yetenekleri ve komut üreteç donanımı (VEPRO) ile birlikte bir parçanın üretimi için gereken takım yolları birkaç satır komut ile oluşturulabilmektedir. VEPRO'nun donanım karmaşıklığı kullanılan geleneksel bilgisayarlı sayısal denetim donanımlarına (Sinumerik ve Fanuc 0*i*) göre oldukça düşüktür.

Anahtar kelimeler: Komut Üretimi, Veri Sıkıştırma, Alan Programlanabilir Kapı Dizini, Servo Motor Sürücüleri, Bilgisayarlı Sayısal Denetim, Kaydırılmış Eğri Üretimi, Morfolojik İşlemler, Çokgen İşlemleri

Çocukların şeker de yiyebildiği bir dünyaya,

Kapıları çalan benim
kapıları birer birer.
Gözünüze görünemem
göze görünmez ölüler.

Çalıyorum kapınızı,
teyze, amca, bir imza ver.
Çocuklar öldürülmesin
şeker de yiyebilsinler.

Nazım Hikmet RAN

ACKNOWLEDGEMENTS

I am deeply grateful to my thesis supervisor Assoc. Prof. Dr. Melik Dölen for his advice, encouragement and invaluable help all throughout the study. Without his assists at the critical points of the thesis, it would be impossible to complete this study. I would also like to thank Asst. Prof. Dr. A. Buğra Koku for his help and advices especially in the first half of my doctoral study as a co-supervisor.

I would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) for the scholarship with the code BİDEB 2211.

The author also would like to gratefully thank Prof. Dr. Tuna Balkan of Middle East Technical University (METU), and Asst. Prof. Dr. Kutluk Bilge Arıkan of Atılım University for their guidance as members of thesis progress committee; Assoc. Prof. Dr. Ali Özgür Yılmaz of METU for his guidance in the paper of Robotics and Computer Integrated Manufacturing Journal; and finally all staff of the Department of Mechanical Engineering of METU for their help and support.

I would like to show gratitude to my colleagues at the department and to my other friends in the city for their sincere friendships during these four years.

I am deeply in debt to my parents Şahhanım and Azimet Yaman for their never-ending love and spiritual support at critical and opportune times.

Lastly, I would like to thank my love and my wife Gizem Yaman for her invaluable support and endless love. As the poet said that if this city is beautiful, it is because of you.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

SYMBOLS

B	Boolean (number) set
e(j)	Geometric deviation (or error)
K	Number of points in **P**
K*	Number of points in **P***
M	Grid size
M_1, M_2	Size of binary image
N	Number of points around the perimeter of the tool
P	Base (point) set
P*	Reference set
$\mathbf{p_k}$	The k^{th} element (a 2D vector) of **P**: (x_k, y_k)
Q	Polygon (vertex) set (or list)
$\mathbf{Q_i}$	The i^{th} disjoint subset associated with a curve offset
q	Point on a specific curve offset
r	Offsetting distance of radius

$\mathbf{S}$	Binary image matrix or boundary point set
$\mathbf{S_k}$	(Sub)set covering boundary points around $\mathbf{p_k}$
$\mathbf{S}_{(\alpha,\beta)}$	Sub-matrix of $\mathbf{S}$ indexed by α, β sets
$\mathbf{S_{k,i}}$	The i^{th} element (point) of subset $\mathbf{S_k}$
$\mathbf{T}$	Tool matrix/set
t_{ij}	Tool matrix element
$\mathbf{u}$	Direction vector
$\alpha(k)$	Index set for interfering base points at step k
χ	Metric registering the change in slope of the base curve
δ	Maximum distance between two consecutive points
ε	Band of error tolerance for curve offset generation
ρ	Pixel size

ABBREVIATIONS

AC	Arithmetic Coding
AF	Amplitude Field
AH	Adaptive Huffman
AMOBS	Adaptive Algorithm to Implement MOBS
CAD	Computer Aided Drawing
CAM	Computer Aided Manufacturing
CAN	Controller Area Network
CBS	Creation of Boundary Set
CCO	Creation of Curve Offsets
CG	Command Generator
CIM	Computer Integrated Manufacturing
CNC	Computer Numerical Control
COG	Curve Offset Generation
CPU	Central Processing Unit
DLL	Dynamic Link Library
DNA	Deoxyribonucleic Acid
EEPROM	Electrically Erasable Programmable Read-Only Memory
DDA	Digital Data Analyzer

FPGA	Field Programmable Gate Array
GPC	General Polygon Clipper
HC	Huffman Coding
IC	Initial Condition
IMOBS	Improved Algorithm to Implement MOBS
ISMMC	Information Source Modeling via Markov Chains
LED	Light Emitting Diode
LF	Length Field
LZ	Lempel-Ziv
LZW	Lempel-Ziv-Welch
MHC	Modified Huffman Coding
MOBI	Morphological Operation on Binary Images
MOBS	Morphological Operations on Boundary Sets
NC	Numerical Control
NURBS	Non-Uniform Rational B-Spline
PO	Polygon Operations
PLC	Programmable Logic Controller
PLL	Phase Locked Loop
RLEZ	Run Length Encoding of Zeros

RPM	Revolution-per-Minute
RT	Real-Time
SD	Secure Digital
SDRAM	Synchronous Dynamic Random Access Memory
SE	Structuring Element
SERCOS	Serial Real-time Communication System
SF	Sign Field
SOC	System-on-Chip
SOPC	System on a Programmable Chip
SRAM	Static Random Access Memory
STD	State Transition Diagram
TB	Terabyte
UART	Universal Asynchronous Receiver Transmitter
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLBN	Variable-Length Binary Numbers
ZF	Zero Field

CHAPTER 1

INTRODUCTION

1.1 Motivation

Digital motion control systems like computer numerical units (CNC) units, motor control modules, servo-motor drivers, and motion control cards do have embedded command generators (CGs). The principal task of these generators is to provide the set points of the defined trajectory to the motion controllers periodically. The trajectory of the tool is defined according to the control languages used in the literature. For instance, in a conventional CNC system the defined trajectory in G codes is parsed within the CG and the set points are generated accordingly. The CG considers various properties of the machine and the tool while generating the motion commands according to the defined G codes.

The CG strategy as briefly described above has well-known drawbacks: **i)** Due to computational complexity and time constraints in CG, complicated hardware (with parallel processors) must oftentimes be incorporated in the motion control system [1]; **ii)** Advanced RT interpolation algorithms, which generally impose considerable burden on the processors and limit servo-update rate, must be devised especially when the trajectory is to be represented by parametric curves such as B-Splines and NURBS ([2]); **iii)** The resulting software (i.e. firmware) development process is time-consuming and error prone; **iv)** Interpreted programs which describe trajectories are not portable since each manufacturer of motion control systems generally adapts her own control language and dialect [3].

The fundamental motivation of this dissertation is to find solutions to the issue described in the above paragraph. With the proposed CG paradigms, Kolmogorov complexity of the conventional approaches is aimed to be diminished. For this purpose, the high redundancy of the motion command data in manufacturing/industrial applications is to be utilized. Consequently, the size of raw motion data could be reduced via data compression and curve offset generation algorithms.

To sum up, there is a potential for devising simple yet effective CGs for industrial motion control systems by fully taking advantage of the current state-of-the-art.

1.2 Scope of the Thesis

The scope of the dissertation is illustrated in Figure 1-1. The VEPRO is the name of the command generation paradigm, whose details are given in the fifth chapter of the thesis. It can be also considered as the capstone of the thesis, since it utilizes all the algorithms developed within the study. As can be observed from the figure that there are two auxiliary fields of the VEPRO. The first one is the field of curve offset generation algorithms and the second one is about the data compression algorithms. Five different curve offset generation algorithms are developed within the scope of the thesis and they are discussed in the third chapter. On the other hand, a novel data compression algorithm, which is specialized to compress integer encoder pulses, is proposed in the fourth chapter and its performance is compared with the conventional compression techniques. The last part of the thesis is related with the data compression chapter as can be seen from the figure. In this last chapter before the conclusion, the FPGA implementation of the proposed data compression algorithm is explained.

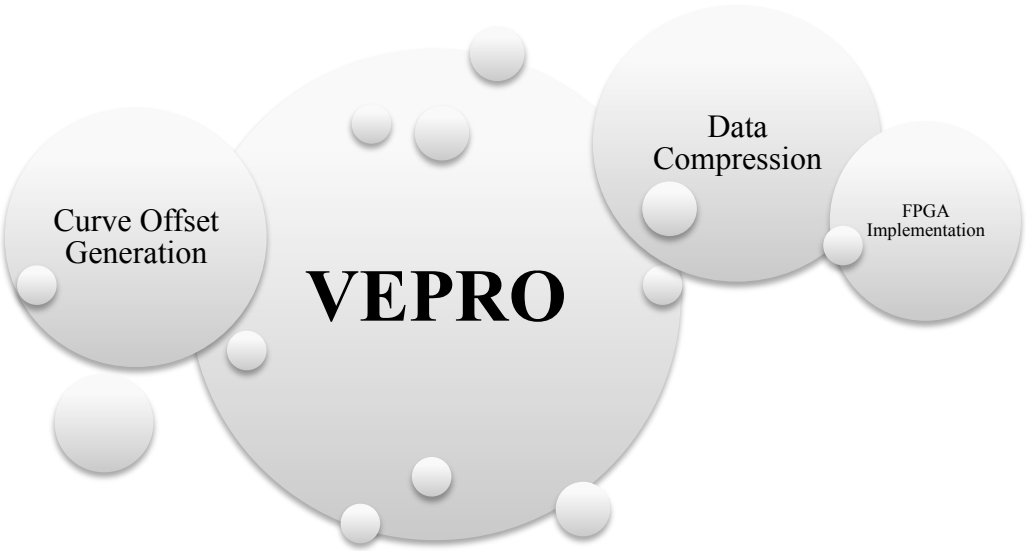


Figure 1-1 Scope of the Thesis

1.3 Organization

The dissertation starts with the Introduction chapter. The motivation behind the thesis is discussed and the scope of the thesis is given in this chapter. After the introductory chapter, the thesis continues with the Literature Survey chapter. The studies on curve offset generation algorithm, command generation methods for CNC machinery, acceleration/deceleration algorithms, data compression methods and their implementations, and FPGA-based command generation systems are summarized. The chapter is concluded with discussing the open research areas. The following chapter focuses on the developed curve offset generation algorithms. The first four of the algorithms employ morphological operations and the last one utilizes polygon operations to generate the curve offsets. After each method is described, their time and memory complexities are also discussed. The

chapter is concluded with the evaluation of the methods on the same two test cases. Then in the fourth chapter, a direct command generation paradigm for CNC machinery based on data compression methods is introduced. Beside the evaluated compression algorithms, Markov chains are also utilized to model the given trajectories. As in the previous chapter, the proposed and the conventional compression algorithms are employed on the same test cases and their results are compared. In Chapter 5, a different command generation paradigm is introduced. The best curve offset generation algorithm proposed in the third chapter and the data compression algorithm introduced in the fourth chapter are all utilized within this novel command generation paradigm. After the commands of the paradigm are given, it is emulated in MATLAB environment. To the end of the chapter, the paradigm is compared with the conventional approach via employing them on the same test cases. The last chapter before the conclusion discusses the hardware implementation details of the command generation paradigm proposed in the fourth chapter. The implementation is carried out in two different ways. In the first approach, the units required to generate motion commands from the compressed data set are designed with VHDL language and the compiled file of the overall system is downloaded onto the FPGA development board. In the second approach of the hardware implementation, a microprocessor is embedded into the FPGA chip of the development board and the command generation algorithm is processed on this microprocessor. The performances of these approaches are also evaluated within the chapter. The dissertation is concluded by pointing the key results of this study and discussing the possible future works related to the thesis.

CHAPTER 2

LITERATURE SURVEY

In this chapter of the thesis, the relevant literature topics are discussed in a detailed manner and open research areas are also highlighted for possible further study.

2.1 Curve Offset Generation Algorithms

Morphological operations are utilized in different fields of science, but there exists rare work on curve offsets with these operations. There are studies [4] in the literature on monitoring of the tool wear in which morphological operations are used to filter the acquired images, but morphological operations are firstly used by Jimeno-Morenilla et al. [5] for the computation of tool paths in manufacturing. They defined a trajectory-based dilation operation that orients the structuring element in any position on the boundary of the object. With the new morphological operation the boundary of the objects could be are computed. The major disadvantage of this operation is that it is not applicable for the objects having holes (islands). After the successful implementation of the algorithm, Carmona et al. [6] used the same idea in their study to compute offsets for contour pocketing three years later. They represented the tool as a structuring element having shapes of a circle and a rectangular. The main advantage of the algorithm they presented is that there is no need to treat self-intersections and discontinuities since these are not present when the proposed approach is employed.

Furthermore, their offset generation algorithm is capable of dealing with the islands by recursively calling the sub algorithms when there exists other curves (islands) in the geometry. Later in 2011, Jimeno-Morenilla et al. [7] used similar morphological operations to reconstruct the computer-aided drawing of an object from point cloud obtained with the mechanical digitizers. As in the previous studies, by the use of morphological operations they were able generate inverse offsets without any self-intersection trouble. On the other hand, the computational cost may be a problem when the number of points in the cloud is enormous. Before Jimeno-Morenilla et al. [7], Yingjie et al. [8] also worked on generating offset curves from point cloud. They also used image processing based techniques in their study, but the approach is totally different. The cloud data is first segmented into layers whose thicknesses are determined based on the linear correlation of layers. Then these layers are projected on the machining plane and their gray-scale images are formed. Finally offsets are calculated using the offset filtering proposed by Chamberlain [9] in his Ph.D. dissertation. He developed various algorithms (tool path computation, offset generation) based on image processing methods for manufacturing and rapid prototyping applications.

Specific curve offset generation algorithms for NURBS curves are present in the literature. One of the outstanding curve offset algorithms for NURBS curves is suggested by Piegl et al. [10] in 1998. They used a different approach from previously suggested algorithms. The earlier algorithms start the offsetting with a few points and then add additional points till the convergence is done. On the other hand, Piegl et al. start with using many points and then eliminate these points according to the tolerance band they defined. When the results of the method are investigated, it can easily be concluded that the technique does well compared to the previous ones. One of the important properties of the algorithm is that by changing the parameters in the formula they used the performance (speed) of the technique increases significantly.

Polynomial approximation techniques are widely in used in curve offset methods in the literature. These approximation methods are reviewed and compared by Elber et al. [11]. During comparison, they counted the number of control points of

the offset to comment on the efficiency of the method. The results showed that the least square methods do well. When the polynomial based offsetting is considered, the performance of the method proposed by Li and Hsu [12] is outstanding. They presented a method for offsetting planar B-spline curves based on the use of Legendre series. They first approximate the B-spline curves by Legendre polynomials and then perform offsetting according to the offset value and error band they defined in advance. Finally they convert the Legendre series to B-spline curves. This method overwhelms the least square method when the approximation accuracy is increase in terms of stability, computation time, and the number of control points. The main disadvantage of the method is that the processing time required for the conversion from Legendre to B-spline is very high.

Voronoi diagram based approaches are generally used to remove the inner loops which are difficult to identify and remove with other techniques. The structures of these algorithms are very similar as given by Held [13]. As a first step, the Voronoi diagram is computed. Then, all of the offset-connected subareas are identified by applying a graph search. With this search the inner most points of the subareas and the bottlenecks between neighbor subareas are also determined. For each bottleneck two straight-line bisectors are inserted. Thus new Voronoi diagrams contain only one point with maximum contour clearance which enables construction of offset curves. Held only compared the processing time of the method for various examples and compared with the processing time of Voronoi diagram generation algorithms (wavefront propagation and divide-and-conquer). It is revealed that curve offsetting algorithm does not take much time when compared with the Voronoi diagram generation algorithms.

2.2 Command Generation for CNC Machinery

Since the introduction of the first NC machine tools, the interpolators based on Digital Data Analyzer (DDA) technique, where a linear or circular path is

generated incrementally via digital integrators, were commonly utilized due to their ease in hardware implementation. First detailed analysis on the subject was attributed to Danielsson [1]. In his work, non-parametric curves were generated through the DDA technique. Since the presented algorithms were not associated with error criteria, they yielded asymmetrical curves in symmetrical arrangements. As the CNC technology advanced, software interpolators, which took advantage of microprocessor technology, began to emerge [15]. Besides significant improvements in linear and circular interpolators [16], several researchers [17]-[19] have concentrated on the development of both RT and off-line interpolators for basic parametric curve (parabolic, Bezier, circular arc, etc.) generation.

As modern CAD systems have progressively gained the capability to describe a wide variety of complex shaped parts (like dies and molds) through parametric curves or surfaces like the Bezier, B-Spline or NURBS; a number of parametric curve interpolators, which have the potential to work directly with these geometric entities, have been proposed by several investigators [20]. Among these parametric curves, NURBS is one that draws considerable attention owing to the fact that NURBS offers a universal mathematical form for representing both analytical and free-form shapes [15]. In fact, most commercial CNC controller manufacturers (such as Fanuc [21] and Siemens [22]) incorporate such interpolation capabilities to their high-end CNC products. Many investigators have proposed advanced NURBS curve interpolators to address the challenges of NURBS interpolation including heavy computational burden and feedrate fluctuation due to round-off/truncation errors in the interpolator [23].

Parallel to these efforts, there are various algorithms proposed on feedrate control in order to increase the quality of the product. For instance Cheng et al. [24] employed a predictor-corrector algorithm to estimate the servo command at the next sampling time. During the prediction stage, an algorithm is used to estimate the next command value and in the corrector phase errors due to the prediction are eliminated. Cheng and Tsai [1] developed a new interpolator to produce servo commands for RT control of CNC machining at variable feedrates. The main difference of this algorithm is that acceleration/deceleration planning is performed

before the interpolation takes place. In a similar study, Xu et al. [25] presented variable interpolation schemes for planar implicit curves. They were also able to interpolate in RT to improve machining efficiency where the feedrate set by the operator is modified according to the geometrical state of the surface. In other words, it is decreased when the tool is machining curved parts and increased on planar surfaces. Another approach is proposed by Rutkowski et al. [26] in order to guarantee the high quality of machining. They pointed out that the smoothness of the trajectory profile is a necessity and employed a neuro-fuzzy based system to change the feedrate online in an adaptive manner. By this approach the machining operations become robust to changing external conditions.

Despite the considerable efforts expanded on developing various interpolation paradigms to generate command sequences for CNC machine tools, the direct storage of sampled trajectory data (and the corresponding methods) have been neglected in the industry/academia due to justified reasons: **i)** Memory capacity has always been the most scarce entity since the emergence the NC/CNC technology; **ii)** For machining applications like turning and milling, the trajectory must be modified dynamically due to changes in operating conditions such as feedrate override, cutting tools being employed, tool geometry change due to wear, etc. Fortunately, in most manufacturing machinery like abrasive water jet cutters, and rapid prototyping machines, such changes are not usually exercised during operations. Additionally, with the advancements in the solid-state electronics, the scarcity of memory is no longer the case.

In fact, the efficient data storage and retrieval techniques based on data compression have been exclusively studied in the literature [27]. However, the earlier application of the data compression techniques to CG is due to [28] where a RT command generator, which employs differencing and data compression, is introduced. The command generation performance of the proposed method is evaluated through the trajectories of a Puma 560 manipulator. A brief realization of the generator using a Field Programmable Gate Array (FPGA) is also discussed in their study. Likewise, [29] improves the method further by adding variable data-rate command generation capability and implements this novel generator on

a FPGA. Despite encouraging results, the command generation methodologies of [28]-[29] are nowhere complete and require more comprehensive assessment through real-world scenarios.

2.3 Acceleration/Deceleration Algorithms and Online Command Generation

The state of the art command generation systems consider the acceleration and deceleration limits of the machinery along with the incoming signals from the sensors and then generate and/or modify the defined motion commands accordingly. There are various studies in the literature to modify the original trajectories according to the abrupt changes in the machining conditions. For instance, the main contribution of Kröger and Wahl's study [30] is that the online generated trajectories enable the systems to adapt to the unpredictable sensor inputs by the parallel execution of the generator algorithm. The technique ensures the safe and continuous motion of the machinery in unpredicted cases. With this approach, the integration of the sensors to the robotics and CNC applications becomes easier. In another study, Haschke et al. [31] developed a real-time algorithm to update the trajectories of the robots due to the same reasons (sudden changes in the environment). Although the computational complexity of the method is high, they claim that real-time performance can be achieved. The main advantage of their algorithm is that it can overcome the arbitrary initial condition problem, which the conventional approaches cannot handle.

The study of Yong and Narayanaswami [32] concentrates on the sudden feedrate changes due to the high number of segments defined in the conventional machining approach via NC programming. They determine the feedrate sensitive corners in the machining trajectories in advance and then calculate the parameters of the acceleration and the deceleration sectors. Since they also take into consideration the capabilities of the CNC machinery, there occurs no overcut or undercut on the workpiece. In a recent similar study, Bianco and Ghilardelli [33] proposed a discrete filter to smooth the rough reference commands according to

the predefined requirements. The main advantage of their algorithm is that due to the compactness and the efficiency it can be preferable for hardware implementations.

2.4 Data Compression Methods and Their Implementations

Data compression is regarded as the crucial component in high-speed data transfer and storage. Two types of compression exist in the literature: lossless and lossy. In lossless data compression, data set is encoded to a smaller one that can later be decoded back to its original state whereas in lossy compression, the original data can only be approximated after decompression. Technical literature on data compression (i.e. Information Theory) is too vast to cite here. Readers are encouraged to refer to [27] (& [34]) for a general overview of this field.

Lossless data compression applications have increased over the past years due to the need to improve the storage capacity and transfer rate for audio/visual data [35]. There are many examples for the hardware implementations of conventional encoding techniques in the literature. Among these techniques, Huffman [36]-[37], Lempel-Ziv (LZ) [38]-[39], and Golomb [40] compression algorithms are the most popular ones for hardware implementations. For instance, Rigler et al. [36] implemented Huffman and LZ encoders on an FPGA and concluded that augmented Huffman coding (HC) uses less hardware resources than the LZ algorithm. On the other hand, Abd El Ghany et al. [38] also realized the LZ encoding and decoding algorithm on FPGA. In order to increase the efficiency, they used systolic array that resulted in a 40% decrease in the compression rate.

Since various researchers have already implemented the conventional compression algorithms, recent studies mainly focus on implementations of improved versions of the conventional techniques. As an example, the study by Koch et al. [41] can be examined. They employed additional decompression accelerators on the conventional algorithms and able to achieve comparable compression ratios with successful software solutions along with utilizing

negligible hardware resources. Rather than improving the conventional approaches, some researches also tried to combine the techniques. In one of these studies Lin et al. [42] proposed a hybrid compression algorithm composed of Adaptive Huffman (AH) and LZW coding techniques. With this approach, they achieved the compression performance of LZW coding by utilizing less hardware resources than the case when only AH coding is implemented. In another study, Lee and Park [43] implemented Huffman coding utilizing different parallel shifting algorithms. In order to satisfy the bandwidth requirement of the data acquisition system, they employed the compression algorithm to decrease the bandwidth of the output of the calorimeter utilized in their experiments.

Among conventional data compression techniques, hardware implementations of different algorithms for compressing specific data structures are also present in the literature. For instance, Yongming et al. [44] have realized the Linear Approximation Distance Threshold algorithm on FPGA to compress the Electrocardiograph signals. Similarly, Valencia and Plaza [45] developed an FPGA-based data compression technique based on the concept of spectral unmixing to compress hyperspectral data. In another study related to unconventional encoding methods, Ouyang et al. [46] combined different compressing techniques to compress the huge DNA sequences which resulted in an algorithm having fast generation time and high compression ratio.

The ΔY10 compression algorithm is similar to these methods in the sense that it is specifically developed to compress integer encoder pulses. When it is employed on image, text, sound, etc. data, it will not be successful as conventional compression algorithms.

2.5 FPGA-based Command Generation Systems

In the last decade FPGA has become widespread in the network and embedded control systems due to its high flexibility, reduced execution times, and relatively low cost [47]. For instance, Kim et al. [48] utilized FPGA technology to decrease

the decoding delays in the random linear network coding technique. With the FPGA they were able to increase the speed of the operation and decrease the power requirement. On the embedded control systems side, Cho et al. [49] developed a multiple-axis motion control chip utilizing an FPGA. The chip is capable of performing all the required tasks for industrial robots and automation systems quickly and accurately. On the other hand, FPGA-based designs are not very common in command generation parts of the control systems. One of the few implementations was carried out by Su et al. [50] with a simple controller integrated. They preferred to modify the digital convolution technique rather than employing the complex polynomial technique in order to implement trapezoidal and S-curve motion planning which resulted in an increase in the computational complexity. The digital convolution technique was also used by Jeon and Kim [51] for developing an FPGA-based acceleration and deceleration hardware for CNC machine tools and robotics. Similar to the study of Su et al. [50], the complex polynomial method is not preferred due to the computational complexity for the generation of velocity profiles having different dynamic characteristics. Instead of the current strategies, they proposed a new method to overcome the computational burden. The results reveal that the unsymmetrical profiles, which are not possible to be produced by digital convolution, can be generated by their method. In the study of Jeon and Kim [51] the error is not compensated. On the contrary the command generation schemes proposed in the thesis is capable of generating motion trajectories without any faults. Beside this advantage, it also generates the velocity and acceleration curves simultaneously provided that the order of difference is selected as two or higher during encoding. Osornio-Rios et al. [52] preferred to use profiles with higher degrees in order to generate trajectories in place of digital convolution method as Su et al. [50] and Jeon and Kim [51] employed. During the FPGA implementation, a multiplier-free recursive algorithm is designed to reduce the complexity of the profiles. Since the trajectories have jerk limitation with the utilization of higher degree polynomials, the dynamics of the machinery are also advanced with the proposed algorithm.

2.6 Open Research Areas

With a detailed literature survey on the related topics of the dissertation, the scope of the thesis is determined. Since the dissertation is supposed to have certain limits regarding the topics, some important topics are apparently left to be discovered and studied on deeply.

Although it is mentioned that the developed command generation paradigms should consider the acceleration/deceleration limits of the machine axes in the fourth chapter, the paradigms proposed in the dissertation has no units to take into consideration of such limits. The one proposed in the fourth chapter may not need such a unit since the uncompressed command trajectory may have been generated according to the acceleration/deceleration limits of the corresponding CNC machinery. On the other hand, the scheme proposed in the fifth chapter do need a unit modifying the trajectories according to the given properties of the CNC machinery. This due to the fact that the command trajectories in this approach are defined via the proposed commands in advance and there is not any opportunity to consider the motion limits during the programming stage of the paradigm. To sum up, an algorithm capable of modifying the generated trajectories according to the specifications of the CNC machinery should be developed and embedded into the proposed paradigms in the dissertation. The time complexity of the developed algorithm should be linear-in-time in order not to decrease the speed of generation in general.

Another further research area may be finding a relationship between the LZW compression algorithm and the contextual modeling of the motion trajectories. When the results obtained in the fourth chapter are investigated, it can be inferred that the LZW outperforms the other compression algorithms in the third case, where there are repetitive tool movements in all of the axes. On the contrary, it is one of the worst compression algorithms in the cases where the tool does not perform repetitive tasks. The algorithm behind the LZW can be investigated and a modified version of it can be utilized to model the tool trajectories automatically. This suggested paradigm may be more successful than the command generation

techniques proposed in the dissertation in terms of the memory requirement and Kolmogorov complexity.

CHAPTER 3

CURVE OFFSET GENERATION

For the purpose of generating 2D curve offsets used in 2.5D machining, five new methods based on morphological operations on different mathematical entities are presented in this chapter of the thesis. All of the methods, which lend themselves for parallel processing, exploit the idea that the boundaries formed by a circular structuring element whose center sweeps across the points on a generator/base curve comprise the entire offsets of the progenitor. The first approach, which is a carry-over from image processing, makes good use of morphological operations on binary images to produce 2D offsets via contour tracing algorithms. The second method, which is to rectify the high memory cost associated with the former technique, utilizes morphological operations on (boundary data) sets. The implementation of this basic technique is illustrated by two MATLAB functions given in the chapter. Despite its simplicity, the time complexity of this paradigm is found be high. Consequently, the third method, which is evolved from the preceding one, reduces the time complexity significantly with the utilization of a geometric range search method. This technique, which has a considerable margin for improvement, is found to be suitable to be used as a part of the real-time motion command generator for CNC applications. The fourth technique is the adaptive version of the previous one. It does not generate local curve offset points that are going to be eliminated due to the local problems. Unlike the previous schemes, the final approach uses polygon operations to generate such curves. The

run-time of this technique is highly governed by the complexity of the polygon overlay algorithm selected. The chapter analyzes the complexity of each technique. Finally, the presented methods are evaluated (in terms of run-time and geometric accuracy) via two test cases where most CAD/CAM packages fail to yield acceptable results.

3.1 Introduction

Curve offset generation (COG) is utilized in many different engineering applications such as computer graphics, computer numerical control (CNC), computer aided design (CAD), computer aided manufacturing (CAM), computer integrated manufacturing (CIM), industrial automation and robotics, die/mold design, rapid prototyping, and more. Planar (2D) curve offsets especially play a critical role in manufacturing such as pocket machining [53] and rapid prototyping [54] in which the tool path is generated in 2.5D where a family of planar curve offsets is produced at different elevations along the tool axis.

In fact, a curve offset is said to be the locus of points which are at constant distance along the normal from a base curve (i.e. progenitor). Despite its plain definition, the generation of (even planar) curve offsets happens to quite challenging owing to the fact that the offsets of a rational base curve are frequently in non-rational form. Due to its practical importance, there exist a wide range of research efforts on the subject. Maekawa [55] groups them into five categories: i) exact offset generation (including Pythagorian hodographs) [56]-[57]; ii) approximation techniques [58]-[59]; iii) self-intersection detection/elimination [60]-[61]; iv) geodesic offsetting [62]-[63]; v) others [64]-[65]. The emerging methods based on mathematical morphology [6]-[7] can be regarded as approximation techniques. A comprehensive literature review on this issue can be found in [55].

It is critical to note that the motivation behind this study is to devise efficient COG methods that could serve as an integral component of (discrete-time) motion command generators such as the ones presented by [29] and [66] where the tool/end-effector trajectories are essentially represented by sampled sequences of position in temporal domain rather than the ones expressed in rational forms (i.e. NURBS). Unfortunately, the literature review on the subject reveals the lack of general-purpose methods (suitable for time sequences) that effectively handle the exceptional cases arising in typical COG phase including self-intersection, unreachable/unfeasible locations, sharp turns, isolated patches, and more (for

instance, see [67]-[68]). Consequently, the main goals and eventually the contributions of this study can be summarized as follows:

i. To present some new paradigms, which are based on morphological operations on different mathematical objects including binary matrices, sets, and vertex lists, for the purpose of generating 2D curve offsets. All of the presented techniques in this chapter are inherently to deal with the afore-mentioned exceptions and must be suitable for implementation on parallel processors.
ii. To discuss important properties of these techniques in order to highlight their potential for performance improvement.
iii. To illustrate the direct implementation of a COG technique (based on morphological operations on boundary data sets) through a number of MATLAB functions and to provide the readers with the initial tools for further development of their own techniques.
iv. To present novel general-purpose algorithms, which has a reduced time- and memory complexity.

The organization of this chapter is as follows: After this introduction, the basic definitions regarding the tool trajectory (i.e. base/generator curves) are presented. In the next section, morphological operations on binary images are discussed to generate 2D curve offsets. Since this technique is found to be costly in terms of memory usage, another method based on morphological operations on the boundary data sets is introduced to circumvent that drawback. In the following section, a new algorithm that makes good use of grid search paradigm is proposed so as to reduce the time complexity of the latter method. Then in the upcoming section, the adaptive version of the method employing grid search algorithm is introduced. Following that, a different and rather unconventional approach based on polygon operations is elaborated. All of these methods are evaluated rigorously via two test cases in the eighth section. Consequently, some key results and conclusions about this study are given in the final section.

3.2 Definitions

In this chapter, 2D curves (employed in 2.5D machining) will be considered. Let the basis curve of a particular (tool) trajectory be described as an ordered set of *base* points (i.e. 2D vectors) on a plane:

$$\mathbf{P} = \{(x_k, y_k) \in \mathbb{R}^2 : \forall k \in \mathbb{N}_{\leq K}\} \tag{3-1}$$

where $|\mathbf{P}| = K \in \mathbb{Z}^+$. Here, the shorthand notation $\mathbb{N}_{\leq n}$ refers to the set $\{x \in \mathbb{N}^*: x \leq n\}$, where $0 \notin \mathbb{N}^*$. Despite the fact that the Cartesian coordinates of the tool (or end-effector) can be conveniently represented as *encoder* counts (i.e. integers) in most CNC applications, only real-valued (vector) sequences are considered to keep the chapter of general interest. A particular point $\mathbf{p_k} = (x_k,y_k)$ in this set is to be interpreted as the desired location of the tool/end-effector at time step k. For $1<k\leq K$, the following condition is presumed to be satisfied:

$$\left\|\mathbf{p_k} - \mathbf{p_{k-1}}\right\|_2 \leq \delta \tag{3-2}$$

where $\delta \in \mathbb{R}^+$ [mm], which is correlated with the sampling period of the time sequence, is a predefined upper-bound for the proximity of any two consecutive points in the set. Since the curve offsets are visualized as a combination of arcs around the base points as illustrated in Figure 3-1, one can simply write the following inequality:

$$\delta^2 \leq 8r\varepsilon \tag{3-3}$$

where $\varepsilon \in \mathbb{R}^+$ refers to the error tolerance band [mm]; $r \in \mathbb{R}^+$ is the curve offset [mm]. Within the context of this study, the constraint in (3-2) is automatically satisfied as the points on the trajectory are sampled at very high rates (>> 1 kHz) for most of the applications.

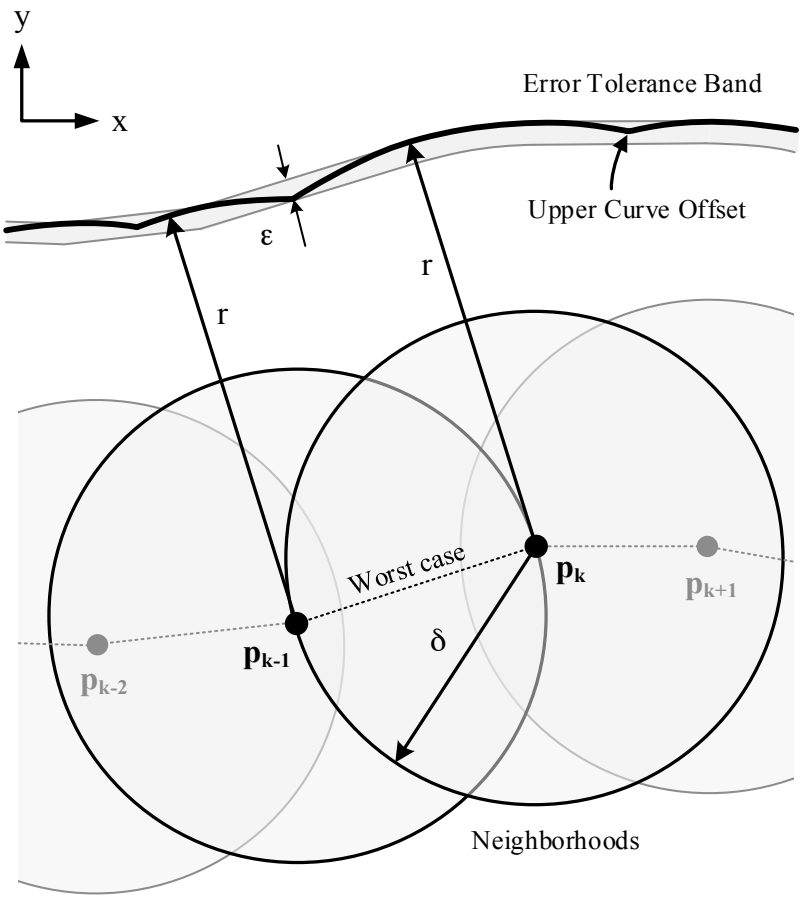


Figure 3-1 Geometric Parameters of Curve Offset Generation

3.3 Morphological Operation on Binary Images (MOBI)

In image processing, morphological operations like dilation and erosion are commonly used to enlarge or reduce certain patterns in binary images. To accomplish that, a mask or a structuring element (SE) is applied successively throughout the contour of the selected pattern. Depending on the bit patterns in the mask and the corresponding logical operations performed on the image, desired end result is obtained [69].

3.3.1 Basic Algorithm

The same idea could be easily extended to generate the curve offsets. To generate the upper or lower (a.k.a. left or right) curve offset, a binary image (i.e. a matrix), which will serve as a medium to compute/store curve offsets, is created first: $\mathbf{S} \in \mathbf{B}^{\mathrm{M}_1 \times \mathrm{M}_2}$ where $\mathbf{B} \in \{0, 1\}$ refers to Boolean number set. Depending on the range of points in **P**, the size of this matrix (image) can be determined as

$$M_1 = \left\lceil (x_{\max} - x_{\min} + 2r_{\max}) / \rho \right\rceil \tag{3-4a}$$

$$M_2 = \left\lceil (y_{\max} - y_{\min} + 2r_{\max}) / \rho \right\rceil \tag{3-4b}$$

where $\lceil . \rceil$ denotes ceiling function (i.e. rounding to the highest integer); $\rho \in \mathbb{R}^+$ [mm] refers to the pixel size of the binary image; r_{max}[mm] is the largest plausible offset distance while

$$x_{\max} \triangleq \max_k \{x_k\};\ x_{\min} \triangleq \min_k \{x_k\};\ y_{\max} \triangleq \max_k \{y_k\};\ y_{\min} \triangleq \min_k \{y_k\}.$$

Similarly, the mask

$$\mathbf{T} = [t_{ij}] \in \mathbf{B}^{N \times N} \tag{3-5}$$

could be envisioned as a binary matrix representing a (closed) circular neighborhood of a SE (a.k.a. *virtual tool*):

$$t_{ij} = \begin{cases} 1, & (\rho i - r)^2 + (\rho j - r)^2 \le r^2 \\ 0, & (\rho i - r)^2 + (\rho j - r)^2 > r^2 \end{cases} \tag{3-6}$$

Here, i, j $\in \mathbb{N}_{\le N}$; N = $\lceil 2r/\rho \rceil$. Provided that $\mathbf{S} \coloneqq \mathbf{0}^{M_1 \times M_2}$ is the initial condition, the Boolean (logical) OR operation on **S** for every point in **P** leads to overlapping images (impressions) of the tool (or SE) along the base points:

$$\mathbf{S}_{(\alpha(k),\beta(k))} \coloneqq \mathbf{S}_{(\alpha(k),\beta(k))} \vee \mathbf{T} \tag{3-7}$$

where $\mathbf{S}_{(\alpha,\beta)}$ denotes a sub-matrix of **S** that is formed by retaining the rows and columns (of **S**) indexed by the sets α and β respectively. In (3-7), the index sets, which are the functions of k, can be given as

$$\alpha(k) = \left\{ i + \left\lceil (x_k - x_{\min})/\rho \right\rceil : \forall i \in \mathbb{N}_{\le N} \right\} \tag{3-8a}$$

$$\beta(k) = \left\{ j + \left\lceil (y_k - y_{\min})/\rho \right\rceil : \forall j \in \mathbb{N}_{\le N} \right\} \tag{3-8b}$$

As the next step, the curve offsets are generated with the utilization of boundary/contour/edge tracing (or tracking) techniques that are frequently encountered in the image processing literature [6]. If a starting point and a direction vector are specified, a set of ordered points along the boundary is produced by simply tracing 0 / 1 (pixel-value) transitions on the binary image [70]. The upper- or lower curve offset is selected by simply specifying the corresponding starting point $\mathbf{p}_s(x_s,y_s)$ and direction vector (**u**):

$$x_s = x_i \mp r u_y \tag{3-9a}$$

$$y_s = y_i \pm r u_x \tag{3-9b}$$

$$\mathbf{u} = \frac{\mathbf{p}_i - \mathbf{p}_{i-1}}{\left\| \mathbf{p}_i - \mathbf{p}_{i-1} \right\|_2} \tag{3-10}$$

where i ($1 < i \leq K$) is a user-specified index; u_x, $u_y \in \mathbb{R}$ refer to the x- and y axis components of the unit vector u in (3-10), respectively .Upper and lower signs in (3-9) are used to generate upper- or lower curve offset. In this technique, an end point also needs to be specified as well to stop the contour tracing algorithm if the base curve is not closed. Otherwise, the technique will yield a single set containing points associated with both curve offsets. Furthermore, the isolated patches remaining inside the image (a.k.a. "holes"), which are the consequence of self-intersection, must be handled separately.

3.3.2 Complexity Analysis

Cost of the method (i.e. time complexity and the memory requirements) can be analyzed in two stages.

3.3.2.1 Creation of Tool Impression at each Base Point

Since N^2 Boolean operations on **S** are performed for each point in **P**, the resulting time complexity at this stage can be simply determined as $O(N^2K)$. The biggest cost is actually associated with the memory requirement where a binary image (**S** matrix) with a size of M_1-by-M_2 (pixels) needs to be created and stored. Thus, the resulting memory cost turns out to be $O(M_1M_2)$. Note that if the pixel size is

selected to be inversely proportional to K in order to maintain a certain resolution, the memory cost in a 2D curve-offset generation could easily give rise to $O(K^2)$. In such a case, the run time would be cubically bounded as $O(K^3)$.

3.3.2.2 Creation of Curve Offset via Contour Tracing Method

Time complexity of contour tracing algorithms is generally regarded as linear (in) time [71]. If there are M pixels on the boundary of a specific pattern, the time-complexity becomes O(M). Similarly, since the locations of pixels on the contour have to be stored during the processing stage, the memory requirement turns out to be O(M) as well. Note that in the technical literature, there exist advanced tracing algorithms (like [72]) that claim to reduce the time complexity [i.e. O(log M)]. However, only the basic techniques will be considered in this study. While it is very difficult to estimate M without analyzing a specific pattern; to obtain a general idea about the cost of the technique, we shall assume that there are $\lceil 2\pi ar/\rho \rceil$ pixels (on average) at the contour around each base point in **P** where the real number $a \in [0, 1]$ represents an average "contour filling" factor computed for a given case (with **P** and r at hand). In that situation, $M = \lceil 2\pi arK/\rho \rceil \leq \lceil \pi a\, NK \rceil$and thus the time complexity and the memory cost would be both O(NK).

Since the time complexity analysis (a.k.a. "big-O" analysis) in computer science deals with the upper-bound of an algorithm's run time function, the dominant costs associated with the sequential steps of an algorithm are considered. Thus, the resulting time-complexity and the memory requirement of the afore-mentioned algorithm (at least in its basic form) would be $O(N^2K)$ and $O(M_1M_2)$ respectively. Despite its shear simplicity, the morphological operations on binary images to generate curve offsets appear to be too costly in terms of memory unless the length of base curve is low to moderate.

3.4 Morphological Operations on Boundary Sets (MOBS)

In spite of the fact that the morphological operations on binary images are simple, the technique (as is) suffers from the curse of dimensionality. As a remedy for this problem, only the relevant information describing the boundaries of a SE (as it sweeps through the base points) can be stored.

3.4.1 Basic Algorithm

The method proposed in this section makes good use of the afore-mentioned idea. Note that there exist a number of studies employing morphological operations on sets to generate tool path offsets in the literature ([6] and [7]). However, the presented algorithm in this section is topologically different than its counterparts in terms of the positioning of the SE (w.r.t. the base curve), the representation (or approximation) of the generator/base curve, the boundary data formation, the detection of tool interference, and generation of tool offset curves through non-iterative boundary data sequencing. The proposed algorithm constitutes two basic steps:

i. *Formation of boundary set:* The boundary points on a circular neighborhood (with a radius of r) around each point in the base set **P** are created. Hence, a finite set $\mathbf{S} = \{\mathbf{S_k}:k\in\mathbb{N}_{\leq K}\}$, which constitutes all the feasible boundary points, are formed such that

$$\mathbf{S_k} \subset \left\{\mathbf{s}\in\mathbb{R}^2 : \left(\left\|\mathbf{s}-\mathbf{p_k}\right\|_2 = r\right) \wedge \left(\left\|\mathbf{s}-\mathbf{p_i}\right\|_2 > r\right), \forall i \in \mathbb{N}_{\leq K} / k\right\} \tag{3-11}$$

ii. *Creation of curve offsets:* Using the *nearest neighbor* technique, the elements of **S**, which happen to be all disjoint sets, are processed to yield I number of (disjoint) subsets $\mathbf{Q_i}$ that contain the points associated with the various curve offsets:

$$\mathbf{S} = \bigcup_{i=1}^{I} \mathbf{Q_i} \tag{3-12}$$

In this technique, the boundary data set **S** is built in a sequential manner. A finite structuring set **T** (at step k) is utilized for this purpose:

$$\mathbf{T}(k) = \left\{ \begin{array}{l} (x,y) \in \mathbb{R}^2 : x = x_k + r\cos\left(\frac{2\pi(n-1)}{N}\right), \\ y = y_k + r\sin\left(\frac{2\pi(n-1)}{N}\right), \forall n \in \mathbb{N}_{\leq N} \end{array} \right\} \tag{3-13}$$

where |**T**| = N is the number of elements in the structuring set. At this point, the interference between the new boundaries [to be formed by **T**(k)] and the ones previously created must be checked within a circular neighborhood of 2r:

$$\alpha(k) = \left\{ i \in \mathbb{N}_{\leq k-1} : \left\| \mathbf{p_i} - \mathbf{p_k} \right\|_2 \leq 2r \right\} \tag{3-14}$$

Here, α(k) refers to a set holding the indices for interfering base points with a particular $\mathbf{p_k}$. First, the set $\mathbf{S_k}$ is constructed using the elements of **T**(k) such that

$$\mathbf{S_k} = \left\{ \mathbf{s} \in \mathbf{T}(k) : \left(\left\| \mathbf{s} - \mathbf{p}_{\alpha(\mathbf{k})} \right\|_2 > r \right) \wedge \left(\left\| \mathbf{s} - \mathbf{p_{k+1}} \right\|_2 > r \right) \right\} \tag{3-15}$$

Then, the previous boundary points enclosed by **T**(k) has to be removed from **S**. Hence, the relevant elements of **S** (all finite sets) are modified as

$$\mathbf{S}_{\alpha(k)} := \left\{ \mathbf{s} \in \mathbf{S}_{\alpha(k)} : \left\| \mathbf{s} - \mathbf{p_k} \right\|_2 > r \right\} \tag{3-16}$$

With the initial condition $\mathbf{S} := \emptyset$, if the above-mentioned set operations are repeated for $1 \leq k \leq K$, the set **S** containing all the feasible boundary points is eventually obtained.

Finally, the boundary set **S**, which covers the entire curve offsets, is processed starting from the first subset. The set construction operation to obtain $\mathbf{Q_i}$ in (12) can be described in twelve steps:

1. Let (the offset index) i be 1.
2. Let (the time index) k be 1. Let **Q** be an empty set.
3. If $\mathbf{S_k}$ is an empty set then go to Step 10.
4. Pick the first element of $\mathbf{S_k}$ as $\mathbf{q} = \mathbf{s_{k,1}}$.
5. Determine the closest element of $\mathbf{S_k}$to **q** (i.e. the nearest neighbor of **q** using Euclidian distance as the metric):

$$\mathbf{q}' = \min_{j \in \mathbb{N}_{\leq |\mathbf{S_k}|}} \left\{ \left\| \mathbf{s_{k,j}} - \mathbf{q} \right\|_2 \right\} \tag{3-17}$$

6. If $\|\mathbf{q} - \mathbf{q}'\|_2 > r$ then go to Step 10. Note that in that case, the remaining elem ents of $\mathbf{S_k}$ are associated with the other curve offsets (upper/lower dependi ng on the case or "child" offsets).
7. Let **q** be $\mathbf{q}'$**.** Let the new subset $\mathbf{S_k}$ be $(\mathbf{S_k} - \mathbf{q})$.
8. Add the new point to the (curve offset) set **Q:** Let **Q** be $(\mathbf{Q} \cup \mathbf{q})$**.**
9. If $\mathbf{S_k}$ is not an empty set then go to Step 5.
10. Increase k by 1. If $k \leq K$, then go to Step 3.
11. If **Q** is an empty set then **end** the process.
12. Let $\mathbf{Q_i}$ be **Q**. Increase i by 1 and go to Step 2.

To implement these operations, two MATLAB functions are developed. The first function titled **`setboun`** is given in Table 3-1 and it simply constructs the boundary data set as described in the text. Similarly, the **`chain`** function, which is listed in

Table 3-2, constructs the sets associated with different curve offsets (i.e. **Q**). Notice that this function has to be called successively until all elements of **S** are empty which implies that all curve offsets are successfully extracted.

Note that the presented algorithm processes the elements of **S** (i.e. $\mathbf{S_k}$) sequentially to form curve offsets without taking into account the global distribution of the boundary points. Often times, this approach leads to a number of segmented curve offsets for self-intersecting base curves. As a remedy to this problem, these curve patches, which are represented as ordered disjoint sets, can be combined through the cross-evaluation of the proximities among the extreme elements (i.e. starting- and end points) of different sets. It is obvious that such an effort requires $2m(m-1)$ operations (e.g. the computation of Euclidian distance & a logical comparison per each operation) where m (> I) refers to the number of curve patches extracted. However, the resulting cost can be regarded insignificant if compared to the rest of algorithm since $m << K$.

Table 3-1 MATLAB Function to Construct Boundary Data Set

```
01: function S = setboun(x,y,r,N)
02:   T0 = r*exp(1i*linspace(0,2*pi,N+1)); T0 = T0(1:N);
03:   K = length(x); idx = 1:K; r2 = r*r; d2 = 4*r2;
04:   S = cell(K,1); S{1} = x(1) + 1i*y(1) + T0;
05:   for k = 2:K
06:     alpha =fliplr(idx(((x(1:k-1)-x(k)).^2 + (y(1:k-1)-y(k)).^2)<d2));
07:     Tk = x(k) + 1i*y(k) + T0;
08:     if (k<K)
09:       Tk = Tk(((real(Tk)-x(k+1)).^2+(imag(Tk)-y(k+1)).^2)>r2);
10:     end
11:     for i = 1:length(alpha)
12:       Tk = Tk(((real(Tk)-x(alpha(i))).^2 +(imag(Tk)-y(alpha(i))).^2)>r2);
13:       Sai = S{alpha(i)};
14:       S{alpha(i)} = Sai(((real(Sai)-x(k)).^2 +(imag(Sai)-y(k)).^2)>r2);
15:     end
16:     S{k} = Tk;
17:   end
18: end
```

Table 3-2 MATLAB Function to Create Curve Offset Data Sets

```
01: function [Q,S] = chain(S,r)
02:   K = length(S); Q = [];
03:   for k = 1:K
04:     qmin = 0;
05:     if (~isempty(S{k})&&isempty(Q)), q = S{k}(1); end
06:     while(and((qmin < r),~isempty(S{k})))
07:       [qmin,idx] = min(abs(S{k}-q));
08:       if (qmin < r)
09:         Q = [Q q]; q = S{k}(idx); S{k}(idx) = [];
10:       end
11:     end
12:   end
13: end
```

3.4.2 Complexity Analysis

Just like its counterpart, the cost of the algorithm can be estimated in two distinct stages.

3.4.2.1 Creation of Boundary Set (CBS)

At the k^{th} step, the set $\boldsymbol{\alpha}(k)$ [see Eq. (14) and Lines 6-7 of Table 1], which indexes the base points interfering with the creation of boundary points at the current position k, must be constructed. Unfortunately, since this set construction requires a search through the base points visited previously (i.e. $\mathbf{P_{k-1}} \subseteq \mathbf{P}$), a query on the set $\mathbf{P_{k-1}}$ ($|\mathbf{P_{k-1}}| = k-1$) can be performed in linear time [e.g. O(k-1)]. Therefore, the cumulative run time for this portion, which is expected to be quite dominant, becomes $O(K^2)$. To reduce the resulting complexity, one can employ an improved search algorithm. In applied mathematics and operations research, there are vast amount of investigations on the problem commonly referred to as range search [73]-[74]. Time complexity of advanced (circular) range search algorithms, which make good use of divide-and-conquer approach, is usually logarithmic in time. When such an algorithm is employed, the overall complexity of constructing set $\boldsymbol{\alpha}$ can reduce down to O(K log K).

With respect to the remaining steps, the total number of operations[1] at step k can be estimated as $|\boldsymbol{\alpha}(k)|(2n-1)+2N$ where n refers to the average/typical number of points in the subsets $\mathbf{S_{\alpha(k)}}$. While it is difficult to determine the cardinality of $\boldsymbol{\alpha}(k)$ without a specific analysis on the trajectory, one can conjecture[2] that $|\boldsymbol{\alpha}(k)|$ would grow in proportion to K. Furthermore, if n is treated like a constant (as it depends on neither K nor N), the corresponding complexity of this portion can be estimated as $O(K^2+NK)$. The resulting complexity for the construction of set **S**

[1] A typical operation here includes the computation of (the square of) the Euclidian distance between two points and comparing it to a certain quantity like r^2.

[2] Analysis on experimental results for a number of cases (a total of 8) is employed as the basis of assumptions in this study.

would be $O(K^2)$. Notice that the presented algorithm will have a quadratic upper bound whereas its counterpart (MOBI) will attain a cubical upper bound in case the resolution is to grow in proportion to the density of the points on the base curve. Finally, the memory requirement in this method can be simply calculated as O(NK) since sets **S** and **T** have to be stored at every step. Therefore, the resulting memory cost turns to be much smaller in practice if compared to its predecessor.

3.4.2.2 Creation of Curve Offsets (CCO)

To proceed with the analysis, an assumption on the cardinality of each subset $\mathbf{S_k}$ is needed (with **P** and r at hand): Let there be n number of points at each subset (on average) where the expectancy E{n} = 2 for a given case. When $|\mathbf{S_k}| = n$, the determination of the nearest neighbors for **q** (until the depleting set $\mathbf{S_k}$ runs out of elements) will take n(n+1)/2 operations at time step k. When n is taken as constant, the overall time complexity of the presented algorithm boils down to O(K). Similarly, the memory cost for this stage is O(NK) since **S** is needed at every step of the algorithm. Interestingly, both the time complexity and the memory cost of this algorithm are comparable to those of its counterpart (e.g. contour tracing algorithm) as described in the previous section.

The fundamental analysis on the presented algorithm reveals that the resulting algorithm (titled MOBS) could yield a better (i.e. lower) run time complexity (e.g. quadratic) than its predecessor (MOBI). However, its memory cost of MOBS would be significantly less. Generally speaking, the presented method could potentially achieve better performance while circumventing the major drawbacks of its counterpart.

3.5 Improved Algorithm to Implement MOBS (IMOBS)

Despite the fact that the memory cost of MOBS is significantly decreased as described in the previous section, its time complexity is quadratic even with the use of an advanced search algorithm. To develop an improved algorithm with reduced time complexity, one must focus on the most costly component of MOBS. Hence, relevant modifications are to be carried out on the CBS portion of the previous algorithm whereas the second part (i.e. CCO), which is already linear, will remain intact.

3.5.1 Basic Algorithm

The CBS of this new algorithm to be called IMOBS hereafter constitutes of two steps:

i. *Generation of local boundary points:* At this stage, the local boundary points, which are located at a specified distance (r) from a base point $\mathbf{p_k}$, are computed without considering the interference with the rest of the base points. To increase the efficiency of this step, the local boundary points are determined by taking into the consideration the curvature of the base curve. For this purpose, the change of slope around a base point is utilized. If the change of slope (χ) is smaller than a predetermined threshold, the boundary subset (with only two elements) for the base point $\mathbf{p_k}$ is constructed conventionally as

$$\mathbf{S_k} = \left\{ \begin{array}{r} (x,y) \in \mathbb{R}^2 : x = x_k \mp \dfrac{r y_k}{\left\| \mathbf{p}_{k+1} - \mathbf{p}_k \right\|_2}, \\ y = y_k \pm \dfrac{r x_k}{\left\| \mathbf{p}_{k+1} - \mathbf{p}_k \right\|_2} \end{array} \right\} \tag{3-18}$$

Otherwise,

$$\mathbf{S_k} = \left\{ \mathbf{s} \in \mathbf{T}(k) : \left(\left\| \mathbf{s} - \mathbf{p_{k+1}} \right\|_2 > r \right) \wedge \left(\left\| \mathbf{s} - \mathbf{p_{k-1}} \right\|_2 > r \right) \right\} \tag{3-19}$$

i. In this technique, the above-mentioned change is simply computed as the cosine of the angle between two consecutive direction vectors at a particular base point:

$$\chi = \left| \cos \Delta\theta_k \right| = \left| \mathbf{u}_\mathbf{k}^+ \cdot \mathbf{u}_\mathbf{k}^- \right| \tag{3-20}$$

where

$$\mathbf{u}_\mathbf{k}^+ = \frac{\mathbf{p_{k+1}} - \mathbf{p_k}}{\left\| \mathbf{p_{k+1}} - \mathbf{p_k} \right\|_2}, \quad \mathbf{u}_\mathbf{k}^- = \frac{\mathbf{p_k} - \mathbf{p_{k-1}}}{\left\| \mathbf{p_k} - \mathbf{p_{k-1}} \right\|_2} \tag{3-21}$$

ii. *Removal of invalid boundary points:* At this step, the invalid elements of the boundary set are removed. That is, the boundary points that interfere with the base curve must be discarded from the set **S** by checking upon the relevant Euclidian distances. Since this computation for each point covered by **S** is extremely costly (as outlined in Section 3.4), a well-known geometric range search algorithm (called grid search) is employed to improve the efficiency of the process [74].

For this purpose, the base plane is divided into a grid with M^2 ($M \in \mathbb{Z}^+$) cells. The indices of the boundary points (i.e. k and the index of the point in $\mathbf{S_k}$) residing within a particular grid cell are stored in a dynamically sized array (like MATLAB *cell* array). Boundary points, which are within the square (2r×2r) neighborhood centered on a base point $\mathbf{p_k}$, are

determined considering the grid cells enclosed within. Hence, if the Euclidian distance between a particular boundary point (within the neighborhood of concern) and a specific base point is found to be less than the offset r, that boundary point is discarded from its associated subset.

Once the feasible boundary set is constructed, the curve offsets are created by re-sequencing the elements of this set with the utilization of the CCO method described in the previous section.

3.5.2 Illustration of the Method

The quadratic time complexity of MOBS is reduced with the improved version of MOBS (a.k.a. IMOBS). In this technique, the main enhancement is on the creation of boundary set part. The second part (CCO) is the same as it is developed for MOBS. For a better description of IMOBS, it is employed on a simple self-intersecting curve (infinity sign) and its steps are illustrated in Figure 3-2. The given self-intersecting trajectory in the first step is discretized in the second step with 96 base points. Then in the third step, the main part of the algorithm is employed on the discretized points to generate inner and outer curve offsets of the given trajectory. At the start point of the trajectory, a circle is generated according to the density of the original points on the base plane. This circle can be viewed on the leftmost part of the third step. Then according to the predefined angle threshold, for each original point it is determined whether to employ the algorithm of MOBS or to generate offset points at the perpendicular directions to the line between the two consecutive trajectory points. In the figure, the red dots indicate that they are generated with the method of MOBS by utilizing the neighbor points of the corresponding point as shown in (3-19). The gray offset points on this step are the ones generated with unit normal vectors. There are now global invalid loops (at the middle of the infinity sign) and invalid

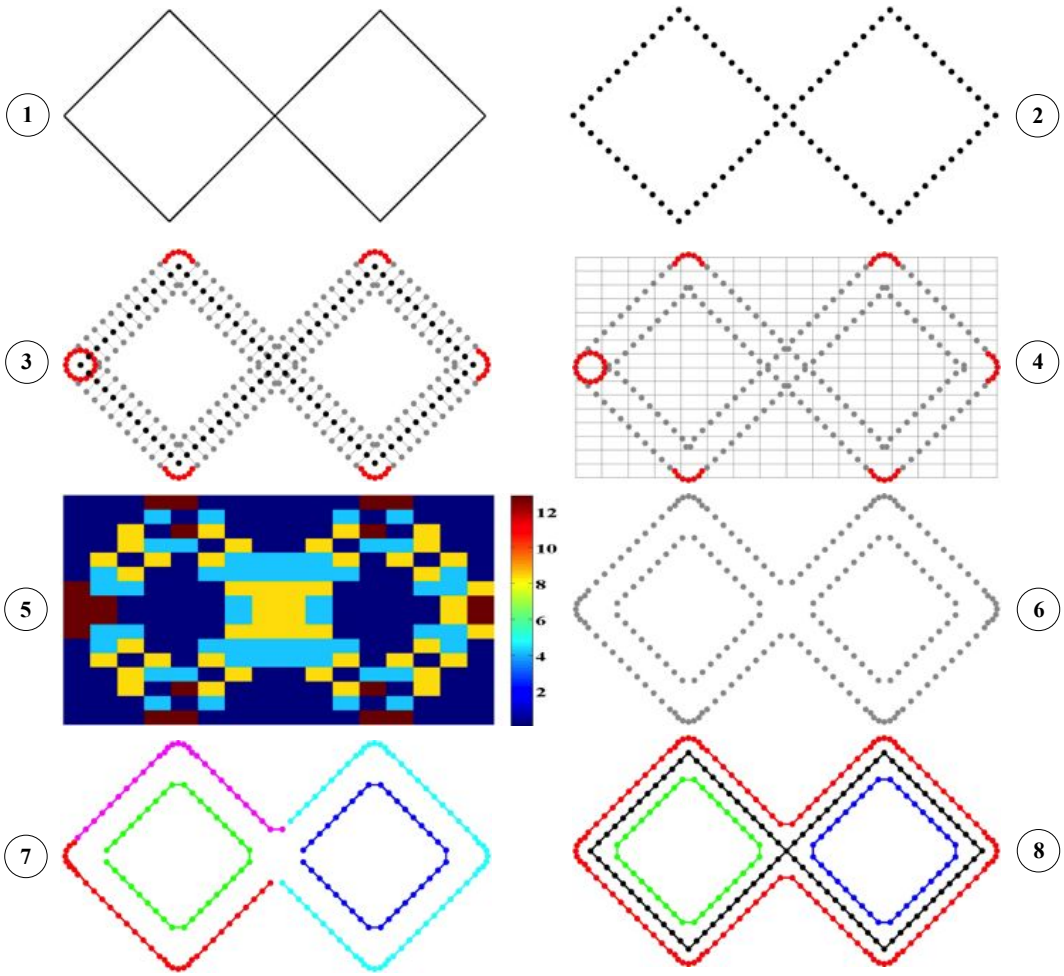


Figure 3-2 Illustration of Steps of the IMOBS

local points (at the start of the trajectory). In order to remove these points with a low computational effort, the base plane is divided into a grid according to the predefined dimensions in the fourth step, which is 16 by 16 for this case. The numbers of points in each cell are illustrated in the fifth step. The invalid points are removed according to their distances to the original trajectory points. If they are closer than the offset distance, they are regarded as invalid and removed from the resulting offset boundary data sets. These resulting offset points are shown in the sixth step of the figure. After these points are generated, the consecutive ones are connected to form curve segments by using the nearest neighbor technique.

The result is presented at the seventh step. Although there should be three closed curves, there are five different colored curve segments at this point. Thus, further processing is necessary to get proper offset curves. In the last step of the algorithm, these curve segments are connected to each other according to the distances between the start and end points of them.

3.5.3 Complexity Analysis

Since the CCO phase of this new technique is same as that of MOBS, only the run time of the first stage of the algorithm (i.e. CBS) needs to be analyzed. As a preliminary operation, all unit direction vectors [see (3-21)] must be computed (and stored) so as to determine the change of direction at a specific base point. It is self-evident that the time complexity of accompanying operation grows linearly: O(K).

At the proceeding step, the boundary points are to be assigned with the utilization of (3-18) and (3-19). In worst case, N operations need to be carried out to create the boundary data set for a particular base point (i.e. $\mathbf{S_k}$). Hence, the time complexity of this step would be O(NK).

Following that, the boundary points interfering with the base points are eliminated via a geometric grid search. To accomplish that, the domain (i.e. base plane) is divided into an M-by-M grid. Hence, a table (a.k.a. a hash table), which holds the indices of the boundary points that reside in a specific grid-cell, is formed first. Note that despite the fact that the probability of $|\mathbf{S_k}|$ being 2 is quite high[3], one can assume that the total number of points covered by **S** will be NK in the worst case. Similarly, the construction of this table requires the range check (and assignment) for every point found in **S**. Therefore, the time complexity of the corresponding operation can be inferred as O(NK).

[3] This is due to the fact that the registered directional change around a base point is usually quite small and that the boundary points are likely to be computed via (3-18) rather than (3-19).

Once the table has been created, one determines the boundary data points that are contained within the square (2r×2r) neighborhood around a base point via simply checking the grid cells located inside this region. Unfortunately, making general assumptions on the distribution of the boundary points over the domain (independent of the test case) is extremely difficult. Note that the straightforward presumption that the distribution of boundary points over the domain is uniform leads to the conclusion that the total number of test points inside a square neighborhood grows on the order of r^2K independent of the choice of M. For every point inside the neighborhood, the Euclidian distance between a base point and a boundary point needs to be computed and checked whether it is greater than r or not. Since this procedure is to be repeated for every point in **P**, the overall time complexity of the algorithm evidently becomes $O(r^2K^2) \sim O(N^2K^2)$. However, such a result appears to be inconsistent with the tests conducted in this study.

A more viable approach is to compute the length of boundaries residing inside the square neighborhood under the presumption that the base curve can be approximated as a line passing through a particular base point. Consequently, the boundary length (i.e. the total length of parallel line segments bounded by the perimeter of the square neighborhood) ranges between $4r(\sqrt{2}-1)$ and 4r as a function of the base line's slope. Provided that the density of the boundary points is in correlation with that of the base curve, the number of points in the square neighborhood can be estimated as $\lceil 4r/\delta \rceil = \lceil 2N/\pi \rceil$ [see (3-2)] in the worst case. The overall time complexity of this step (where the invalid boundary points are eliminated) will be $O(2NK/\pi) = O(NK)$. Finally, the time complexity (i.e. the growth function) of this stage simply becomes $O(\max\{K,NK,NK\}) = O(NK)$ which is more consistent with the experimental results.

With respect to the memory cost, four arrays are needed to store the base points, unit direction vectors, boundary points, and the grid-data. The corresponding memory costs are $O(\max\{K, K, NK, NK\})$. Hence, the overall memory cost for this stage is O(NK).

3.6 Adaptive Algorithm to Implement MOBS (AMOBS)

Among the developed curve offset generation algorithms up to now, IMOBS is the most suitable one for hardware implementation due to its linear in time complexity. When the method is analyzed deeply, it is noted that there is no need to generate possible boundary points that are to be eliminated in the upcoming steps of the algorithm. Thus, a new method is developed, called Adaptive Algorithm to Implement MOBS (AMOBS), to eliminate this disadvantage of the IMOBS.

3.6.1 Basic Algorithm

There exist a number of potential problems in MOBS and IMOBS when the structuring set **T(k)** has a fixed (constant) number of elements that are equally distributed around a circle (with a radius r). If the base points are densely packed while the number of elements is not specified in accordance with the distribution, some of the elements of **T(k)** might be eliminated (due to the sparsity of data) during the process of boundary set construction. As an artifact, various gaps on the boundary data set may occur. Figure 3-3 illustrates such cases on two different tool trajectories having diverse attributes. The trajectories in the figure represent a hand and a star. The number of elements of **T(k)** in these cases is 75 and the number of points in the base curves is 10000. As can be inferred from the figure that there exists no offset boundaries at some specific directions of the original trajectories. Since the number of elements in **T(k)** is selected to be 75 (low compared to the number of points in the original curves), there remains no offset points at these specific directions after the task of elimination of the invalid offset points is accomplished.

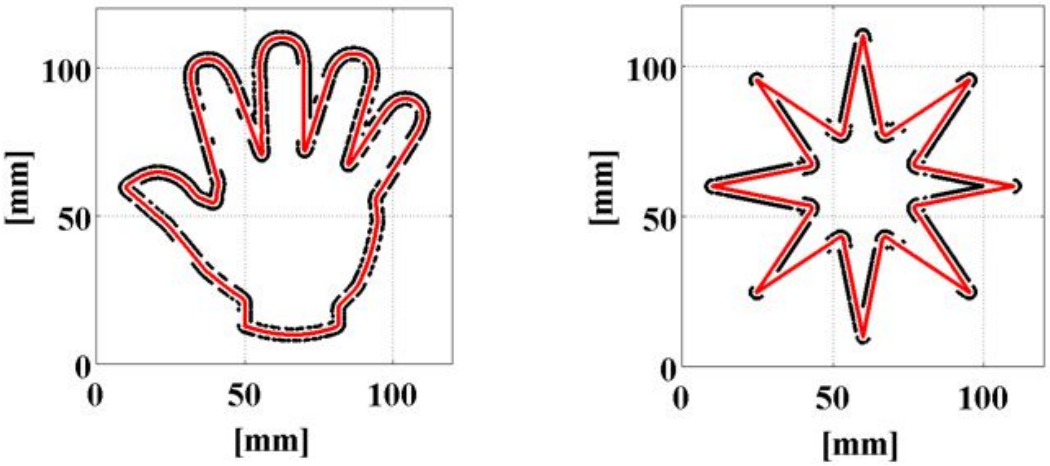


Figure 3-3 Offsets of Hand (N=125) & Star (N=75)

One obvious solution is to increase the number of elements for **T(k)** as follows:

$$N = a\left\lceil \frac{2\pi r}{\delta} \right\rceil \tag{3-22}$$

where a $\in \mathbb{N}$ is an arbitrary (user selected) constant. However, with such a choice (where N $\propto$ K), the overall time complexity of the aforementioned algorithms becomes too high $O(K^3)$ to consider the underlying methods to be practical. As a second remedy, the probability mass function for the angles of normal (unit) vectors at the base points can be calculated beforehand. For a given (constant) N, the elements of **T(k)** can then be selected with the guidance of this function. The number of elements of **T(k)** may be increased at the specific angles where the probability of valid offset points is high. Unfortunately, such an approach would still not guarantee the elimination of these "gaps" in cases where N was specified too low.

A more feasible approach would be to construct **T(k)** adaptively as a function of the gradient vectors around a particular base point $\mathbf{p_k}$. That is, as illustrated in Figure 3-4, the arcs (i.e. upper/lower curve-offset segments), centered around $\mathbf{p_k}$, which do not interfere with the boundaries of the neighboring base points ($\mathbf{p_{k-1}}$, $\mathbf{p_{k+1}}$), could be determined with the utilization of the basic trigonometric identities. The starting and the ending angles of these segments can be expressed as

$$\alpha_1^{\pm} = \tan^{-1}\left(\frac{y_{k+1}-y_k}{x_{k+1}-x_k}\right) \pm \cos^{-1}\left(\frac{\left\|\mathbf{p_{k+1}}-\mathbf{p_k}\right\|_2}{2r}\right) \tag{3-23}$$

$$\alpha_2^{\pm} = \pi + \tan^{-1}\left(\frac{y_k-y_{k-1}}{x_k-x_{k-1}}\right) \mp \cos^{-1}\left(\frac{\left\|\mathbf{p_k}-\mathbf{p_{k-1}}\right\|_2}{2r}\right) \tag{3-24}$$

After these angles are determined, the elements of the tool matrix **T(k)** are generated according to the following equation:

$$\mathbf{T}(k) = \left\{(x,y) : x = x_k + r\cos\alpha, y = y_k + r\sin\alpha, \alpha \in (A^+ \cup A^-)\right\} \tag{3-25}$$

The angle α in the above equation is defined to be in the set A^+ and/or A^-. These sets are expressed by the following equations:

$$A^+ = \left\{\alpha_1^+ + \frac{\delta}{2r} + \chi^+(i-1) : \forall i \in \mathbb{N}_{\le n^+}\right\} \tag{3-26}$$

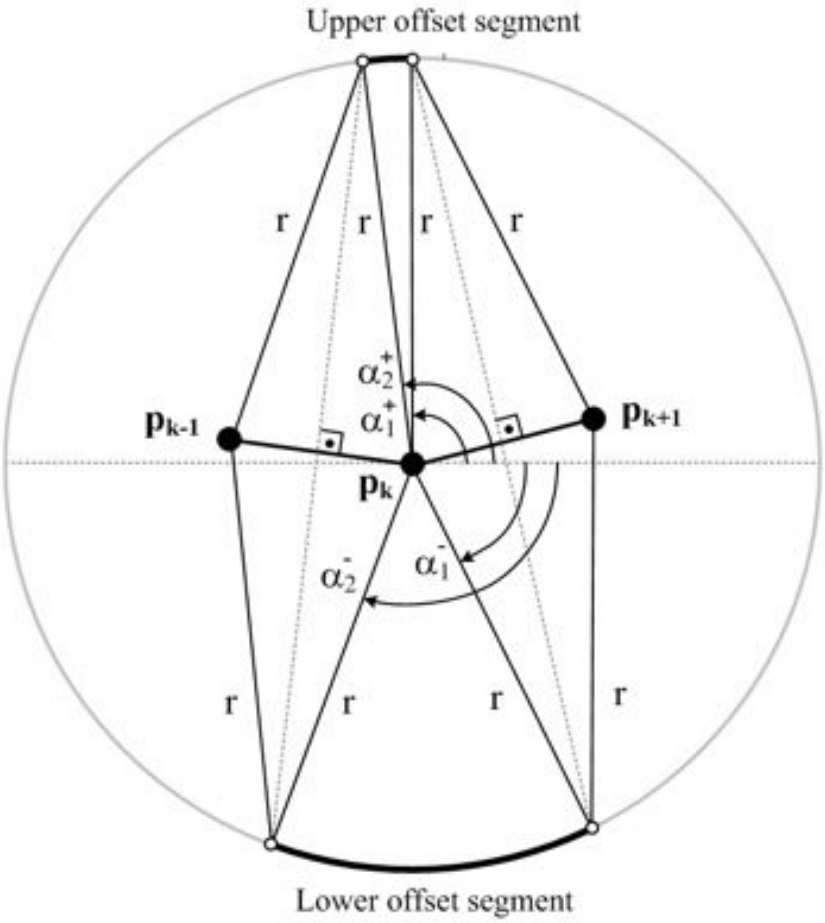


Figure 3-4 Parameters of AMOBS

$$A^- = \left\{ \alpha_2^- + \frac{\delta}{2r} + \chi^-(i-1) : \forall i \in \mathbb{N}_{\leq n^-} \right\} \tag{3-27}$$

$$\chi^\pm = \frac{\left| \alpha_2^\pm - \alpha_1^\pm \right| - \frac{\delta}{r}}{n^\pm - 1} \tag{3-28}$$

In the above equations, the number of elements in the sets is limited by the variable i. Its limit is determined by dividing the length of the arc to the specified distance between the consecutive points in the tool matrix as given in the below equation

$$n^{\pm} = \left\lceil \frac{r\left|\alpha_2^{\pm} - \alpha_1^{\pm}\right|}{\delta} \right\rceil \tag{3-29}$$

Note that when the difference between the angles of the positive or negative side is less than δ/r and greater than zero, then a point is generated at the middle of corresponding arc. On the other hand if the angle difference is less than zero, no point is generated. This exception is handled with the following equation:

$$A^{\pm} = \begin{cases} \left\{(\alpha_1^{\pm} + \alpha_2^{\pm})/2\right\}, & 0 \le \pm(\alpha_2^{\pm} - \alpha_1^{\pm}) \le \frac{\delta}{r} \\ \varnothing, & \pm(\alpha_2^{\pm} - \alpha_1^{\pm}) < 0 \end{cases} \tag{3-30}$$

The rest of the algorithm is similar to the MOBS and the IMOBS. The only difference is that there are not any local invalid points after the generation of offset boundary due to the adaptive behavior of the algorithm.

3.6.2 Complexity Analysis

Since the CCO phase of this technique is the same as that of the MOBS and the IMOBS, only the run time of the first stage of the algorithm (i.e. CBS) needs to be analyzed. For each point in the base set, four angles should be determined first before the generation of offset boundary points. This step simply has time a complexity of O(4K). At the preceding step, boundary points are generated according to the length of the upper and lower arcs. Assuming that there are *a* numbers of points in average on the arcs of a base point, the time complexity of

this step would be O(aK). Following that, the boundary points interfering with the base points are eliminated via a geometric grid search. In the previous chapter, it was concluded that the time complexity of this section is assumed to be O(NK). Finally, the time complexity (i.e. the growth function) of this stage simply becomes O(max{4K,aK,NK}) = O(NK)

With respect to the memory cost, three arrays are needed to store the base points, boundary points, and the grid-data. The corresponding memory costs are O(max{K, NK, NK}), Hence, the overall memory cost for this stage is O(NK).

3.7 Polygon Operations (PO)

Polygons based techniques, in which a number of Boolean operations such as union, intersection, and difference could be performed on two polygons (with arbitrary shape) for various specific purposes, find extensive use in computer-aided design [75], computer graphics [76]-[77], geodesic sciences [78]-[79], and more. However, it is somewhat neglected in the area of curve offset generation.

3.7.1 Basic Algorithm

Polygon operations are utilized to form curve offsets in this study since there exist various algorithms suitable for the generation of curve offsets and their software packages are also available online. These algorithms are named as polygon clipping/intersection [79], polygon overlay [78], Boolean operations on polygons [77], etc.

With an efficient processing technique, the polygon operations could be a viable alternative to generate curve offsets using a base set. To illustrate the application, let us first describe the polygon as an ordered set of vertices listed in counter-clockwise order:

$$\mathbf{Q} = \{\mathbf{q_1}, \mathbf{q_2}, \ldots, \mathbf{q_N}\} = \{(x_{q,1}, y_{q,1}), (x_{q,2}, y_{q,2}), \ldots, (x_{q,N}, y_{q,N})\} \tag{3-22}$$

Here, the edges $\overline{\mathbf{q_1q_2}}, \overline{\mathbf{q_2q_3}}, \mathrm{K}, \overline{\mathbf{q_{N-1}q_N}}, \overline{\mathbf{q_Nq_1}}$ implicitly define the boundaries (edges) of this polygon. To produce the curve offset, two polygons are needed: the first polygon (**T)** describes the contour of a virtual tool (with a radius r) which is to travel along the base curve. The second polygon (**Q**) describes the boundaries that are created dynamically by this tool **T** as it sweeps through the points in the base data set **P** (see Section 3.2 for definitions). The *union* of **T** at every point in **P** will yield the desired the result:

$$\mathbf{Q} = \bigcup_{k=1}^{K} \mathbf{T}(k) \tag{3-23}$$

Or, as a casual relationship,

$$\mathbf{Q}(k) := \mathbf{Q}(k-1) \cup \mathbf{T}(k) \tag{3-24}$$

where **Q**(k), **Q**(k-1) refer to the states of **Q** polygon set/list (in growth) at steps k and k-1 where **Q**(0) := ∅. Similarly, **T**(k) indicates the tool polygon set (i.e. vertex list) defined around $\mathbf{p_k}$ and can formed using (13).

It is critical to notice that the Boolean operations defined on polygons (such as union, intersection, difference, complement, etc.) are fundamentally different than the oneson sets. The algorithms in the literature [77], [80] generally perform the union of polygons in three consecutive steps: **i)** determination of vertices lying inside each polygon (to be purged from both polygons); **ii)** calculation of intersection points (to be added); **iii)** creation of vertices for the new polygon using the information generated at former steps.

The following stage is the extraction of the desired curve offset from the polygon **Q**. Once the starting- and ending points for the curve offset is specified [see (9)],

one can directly identify subsequence of **Q** lying between these points. However, care on the direction of the progression for the vertices must be exercised to determine the correct order of points on the curve offset. The identified subsequence frequently requires inversion/reversion (or "flipping over"). It is critical to note that for self-intersecting curves, independent polygons enclosed by **Q** oftentimes emerge as the union of polygons proceeds. Most algorithms handle this instance by creating an extra polygon (a.k.a. interior polygon/"hole") automatically.

3.7.2 Complexity Analysis

Just like the morphological operations, the cost associated with this method can be analyzed in two stages.

3.7.2.1 Union Operations on Polygons

The run time of the proposed method depends on the selected algorithm performing Boolean operations on polygons. General purpose algorithms with different complexities are available in the literature [76], [77], [78]. For instance, Martínez et al. [78] reports the time complexity of their method as O((n+m)log n)where n refers to the total number of edges for all the polygons involved in the operation while m denotes the number of intersections. Assuming **Q**(k) constitutes [N + 2(k-1)] edges on average while **T**(k) has N edges, the total number of edges becomes n = 2(N+k–1). If the expectancy for the number of intersections (m) is 2, the time complexity of the algorithm at step k will be O((N+k)log(2(N+k–1))). Since polygon operations on **Q** are to be performed for each point in **P**, the upper bound of the polygon based technique in that case could be estimated as $O(K^2\log(K))$ provided that K>>N.

In this study, the general purpose technique presented by Murta [81] (a.k.a. GPC) that employs the polygon clipping algorithm of Vatti [82] (in part) is preferred as

this method is well-established in the literature. Furthermore, GPC intrinsically handles the interior polygons formed as a consequence of self-intersection. Unfortunately, since the Vatti's algorithm is very complex, its "asymptotical" analysis has not been performed, to this day [83]. However, considering the numerical assessments in the literature (along with the ones conducted in this study), one can postulate the complexity of this algorithm as $O(n^2) \sim O((N+k)^2)$. Consequently, the overall complexity of PO technique with the above-mentioned algorithm becomes $O(K(N+K)^2) \sim O(K^3)$.

With respect to the memory cost, since all polygons (**Q** and **T**) need to be stored (independent of the polygon overlay algorithm selected), the resulting cost simply becomes O(K+N).

3.7.2.2 Generation of Curve Offsets

Once the union operations on **Q** are complete, the determination of correct offset can be done by querying the starting- and ending positions of a specific curve offset. Assuming that the polygon **Q** have (2K + N) vertices on average, each query can be performed in linear time: O(2(2K+N)). Hence, since K>>N, the resulting time complexity would be O(K). At this step, only **Q** needs to be stored. Therefore, the corresponding memory cost will be $O(2K+N) \sim O(K)$.

3.8 Experimental Evaluation

The COG techniques along with their implementation/complexity issues have been elaborated in the previous sections. Hence, Table 3-3 summarizes the important attributes of the techniques discussed in this chapter. To evaluate the above-mentioned methods, two test cases (titled Club and Doodle) are considered as illustrated in Figure 3-5. Geometric parameters of the trajectories associated with two cases are given in Table 3-4. Note that the test case "Club" describes a complex closed trajectory whose offsets are likely to self-intersect for certain

offsetting distances while the one titled “Doodle”, which is adapted from Liu et al. [84], represents a self-intersecting tool path.

Table 3-3 Important Attributes of the Methods Considered

Method	Memory Cost	Time Complexity
MOBI	$O(M_1M_2)$	$O(N^2K)$
MOBS	O(NK)	$O(K^2)$
IMOBS	O(NK)	O(NK)
AMOBS	O(NK)	O(NK)
PO	O(K+N)	$O(K^3)$

Table 3-4 Summary of the Test Cases and Selected Parameters

Case	Size [mm×mm]	K	δ [μm]
Club	100 × 100	10000	47
Doodle	100 × 60	20000	37.4

The algorithms MOBS, IMOBS, and PO are implemented and tested in MATLAB 2012b running on two different platforms: iMac (OS X 10.9) and PC (Windows 7/32 bit). However, due to high memory cost associated with MOBI, its performance could not be evaluated experimentally. To be specific, (for a pixel size of 1×1 μm) the memory required to store the binary images for the test cases

alone are 1.82 GB (Club) and 1.23 GB (Doodle) respectively whereas the memory available for all arrays in MATLAB (for both platforms) was limited to 1.37 GB.

With respect to the implementations, for the techniques MOBS, IMOBS and AMOBS, the number of points on the SE is determined via $N = \lceil 2\pi r/\delta \rceil$ in order to avoid gaps on the contour due to insufficient number of boundary points. In the tests, the grid size for IMOBS and AMOBS is selected as 256×256 (M = 256). It is critical to note that the evaluation of PO is carried out with the utilization of the method of Murta [81] where the C++ source code developed to implement the algorithm is available online. To utilize this (open-source) C++ code, it has to be compiled to create a dynamic link library (DLL) so that corresponding functions could be directly called from the MATLAB environment.

First of all, successive curve offsets, which are 2.5 mm apart at the left- and right-hand side of the base curve, are generated by different methods. The results are presented in Figure 3-5. Since the curve offsets produced by a certain technique do not exhibit any visually discernable geometric deviations if compared to the others (at least within the given scale), only the curves produced by the MOBS method are presented in Figure 3-5 to avoid cluttering. It is critical to note that some commercial CAD packages do not yield acceptable results for these two cases. According to the experiments conducted in this chapter, SolidWorks (2010) does not produce any inner offsets for the Club case when the offset curves commence to self-intersect at specific distances. Similarly, for the Doodle case, it totally fails to generate the offset curves at any distance and direction. Liu et al. [84] reports some missing offsets in AutoCAD for the very same case.

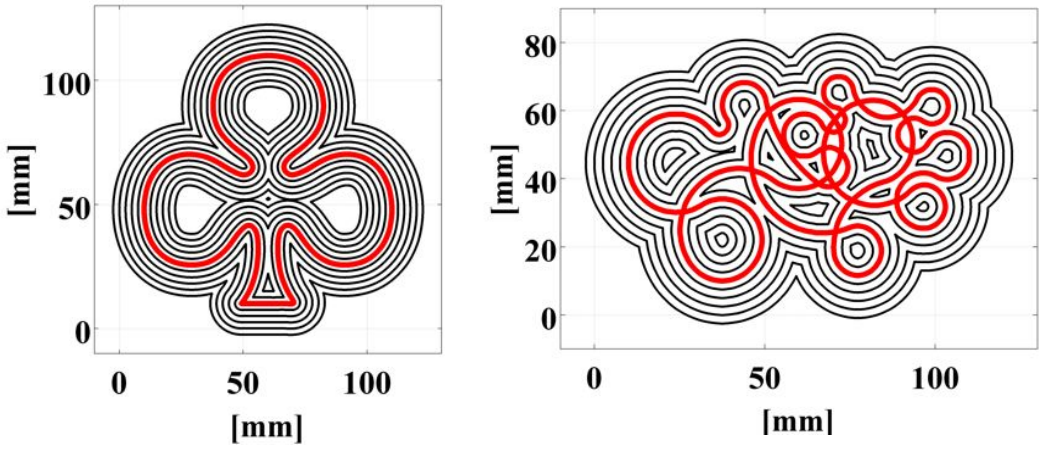


Figure 3-5 Base curves (tool trajectories) and curve offsets produced by MOBS for two test cases

It is critical to note that as mentioned in Section 3.4, the MATLAB function **`chain`** (used by MOBS, IMOBS, and AMOBS) generally yields segmented curve offsets for self-intersecting cases due to sequential processing of the set elements. For instance, 17 offsets are to be produced for the Doddle case when r = 2.5 mm while this function yields 39 segments. Consequently, these segmented offsets are simply merged via the algorithm described in Section 3.4.

For quantitative cost analysis, the memory requirements and the execution times of the afore-mentioned methods are recorded for the test case Doodle (with r = 1 mm) as shown in Figure 3-6, Figure 3-7 and Figure 3-8. As can be observed from the figures, the obtained results appear to be in good agreement with the formal analysis conducted in the previous sections despite the fact the developed MATLAB code (employing advanced data structures), by no means, can be regarded as high-fidelity implementations of the presented algorithms.

For quantitative cost analysis, the AMOBS is evaluated under the same conditions with IMOBS. Figure 3-7 represents the results of time complexity analysis. As can be inferred from the figure that the AMOBS is also linear in time like IMOBS as expected and its slope is a bit less than the slope of IMOBS for the CBS parts

of the algorithms. On the other hand, the difference between the slopes of the CCO parts of the algorithms is high compared to the difference between the slopes of CBS parts. Figure 3-7 also represents the memory costs of the CBS parts of the algorithms IMOBS and AMOBS. The memory requirement of IMOBS is seven times the requirement of AMOBS. This arises from the fact that variable structure of the CCO part of IMOBS is the same as with the previous version of the algorithm (MOBS). With the introduction of AMOBS, another CCO algorithm is also developed and this new CCO now requires different type of variable structures. In the previous versions of the CCO, for each base points (K) an array of valid points are supplied to the CCO via cell arrays. On the other hand, in the CCO of the AMOBS the valid boundary points are supplied to the function in an M by M cell structure. With this approach, the size of the cell structure given to the CCO is kept constant as opposed to the CCO of the IMOBS and MOBS.

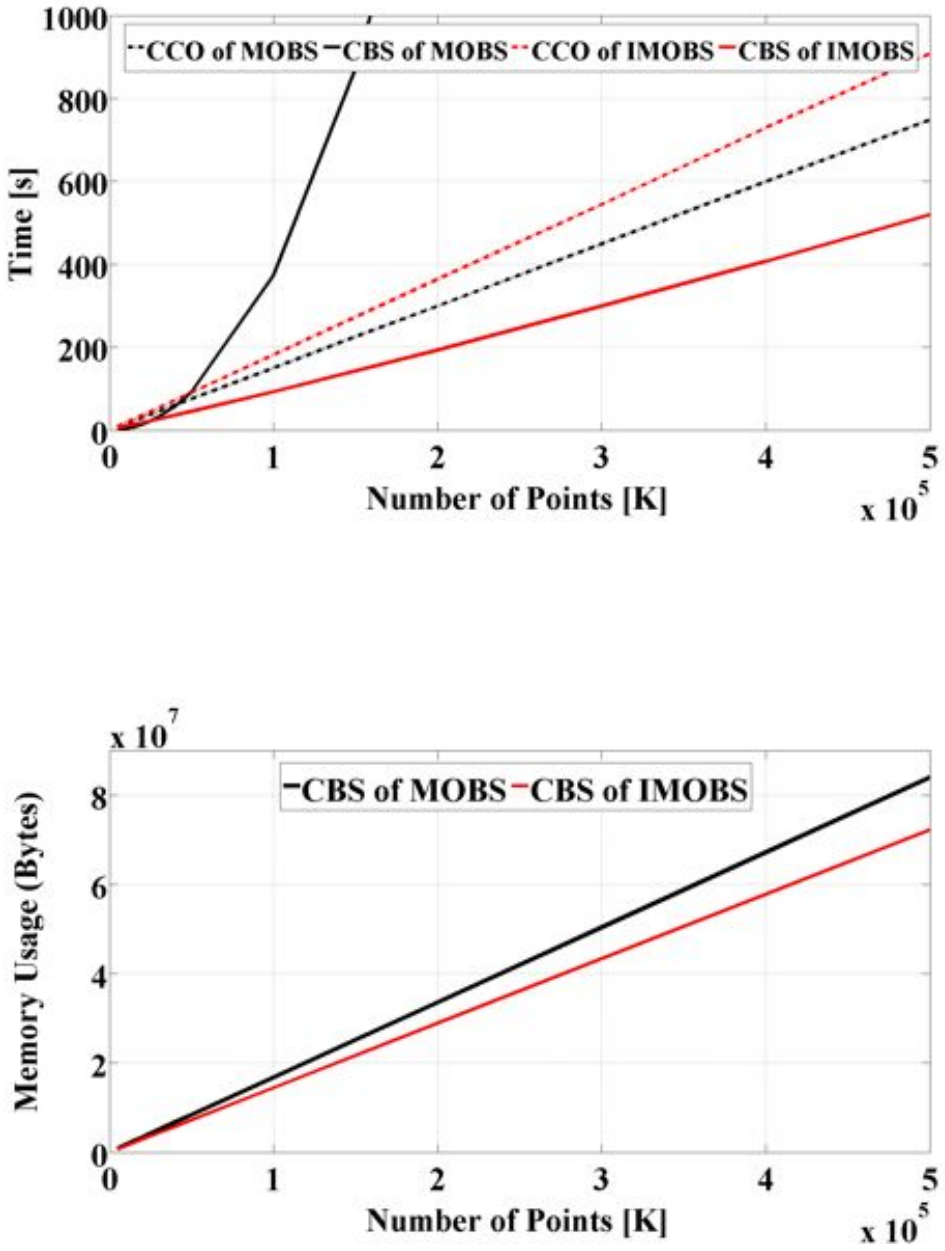
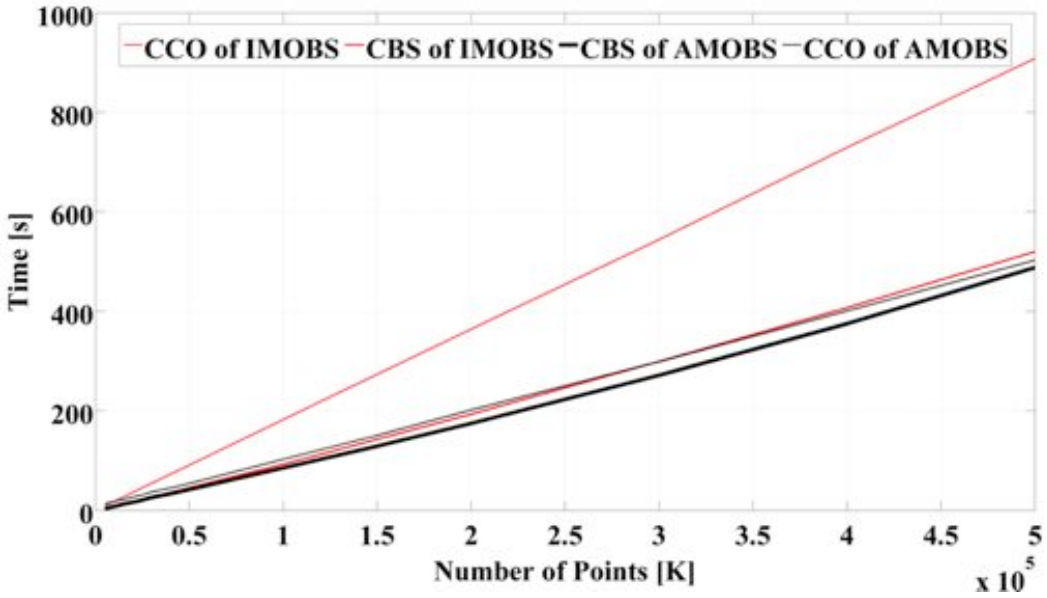


Figure 3-6 Numerical Cost Evaluation for MOBS and IMOBS



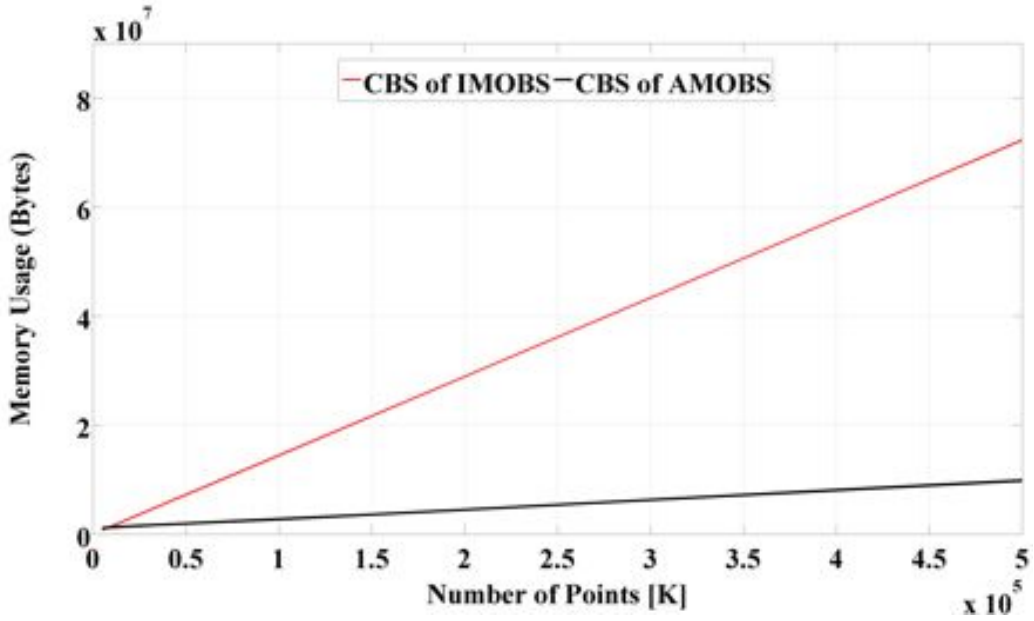


Figure 3-7 Comparison of IMOBS and AMOBS

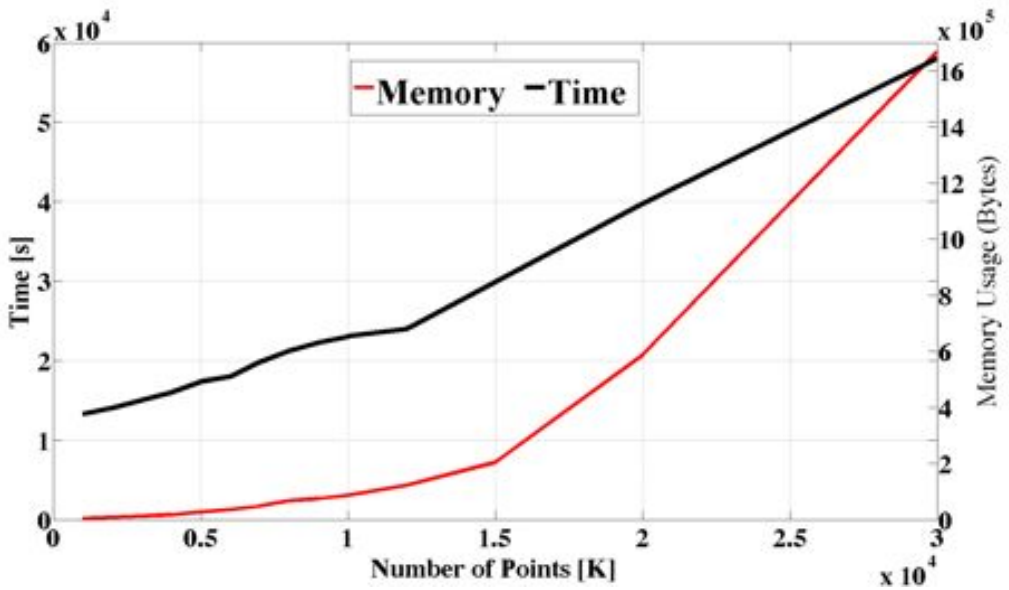


Figure 3-8 Numerical Cost Evaluation of the PO Technique

Since the geometric accuracy of the curve offsets produced by different techniques are of extreme importance for CNC machine tool applications, the deviations of curve offsets from the ideal geometry must be rigorously assessed. However, a reference curve (representing the ideal geometry) for each offset is needed for this purpose. Unfortunately, a well established and general purpose technique to generate these reference offsets (to desired accuracy) does not exist in the current state-of-the art. Therefore, the base curve is again utilized to assess the geometric accuracy. This time, a smooth base curve set (**P***), which constitutes greater number of points than the original (e.g. |**P***| = K* > K), is generated for each case not only to minimize the quantization effects/noise but also to detect the tool interference (i.e. undercut/overcut on the contour) that might arise in between two successive points of the original base set. To produce such a reference set, the original base sequence is sampled at a higher rate (i.e. 4 times) via linear interpolation techniques. Furthermore, a third order Butterworth filter is utilized to filter the resulting time sequence twice (i.e. forward and

backward in time) to eliminate the phase distortion. The deviation from the ideal geometry is determined via shear computation as

$$e(j) = \min_{k \in \mathbb{N}_{\leq K^*}} \left\{ \left\| \mathbf{q}_{\mathbf{i,j}} - \mathbf{p}_{\mathbf{k}}^{*} \right\|_2 \right\} - r \qquad (3\text{-}25)$$

where $\mathbf{p}_{\mathbf{k}}^{*}$ refers to a point in $\mathbf{P}^{*}$; $\mathbf{q}_{\mathbf{i,j}}$ is the j^{th} element of the offset curve set $\mathbf{Q}_{\mathbf{i}}$ ($j \in \mathbb{N}_{\leq |\mathbf{Qi}|}$). Note that if e(j) < 0, the surface (at some point) is expected to be overcut when the tool (with a radius r) is located at the tested offset point. Similarly, e(j) ≥ 0 denotes that some excess material is to be left on the surface.

For r ∈ {-12.5, -10, -7.5, -5, -2.5, 2.5, 5, 7.5, 10, 12.5} (in mm), the geometric deviations associated with each offset are computed and 2D-interpolated to create a geometric error field. Figure 3-9, Figure 3-10, Figure 3-11, and Figure 3-12 illustrate these results for MOBS, IMOBS, AMOBS, and PO. As can be observed from the contour plots, the errors, which indicate by-and-large overcuts on the contour, generally tend to grow with the increasing radius. Similarly, the error is on the rise where sharp changes in the slope of the base curve take place.

For quantitative comparison, the statistical attributes (minimum, maximum, mean, and standard deviation) of these geometric errors at different offsetting distance are shown in Figure 3-13 and Figure 3-14 for the test cases Club and Doodle. Note that the plots of the second case (Doodle) start at -10 mm owing to the fact that no offset points are produced when r = -12.5 mm. As can be seen from the figures, the geometric errors (mean, min) for all techniques are well within the accepted tolerance band of 10 μm. In general, the MOBS technique yields the lowest errors (almost in every category) if compared to the rest due to the fact that more densely populated boundary data set is produced in the CBS phase of MOBS. Another conclusion can be drawn from the figure that the geometric errors of the test case Club are usually higher than those of the other case regardless of the algorithm. This is due to the fact that the Club curve constitutes swifter turns than the Doodle.

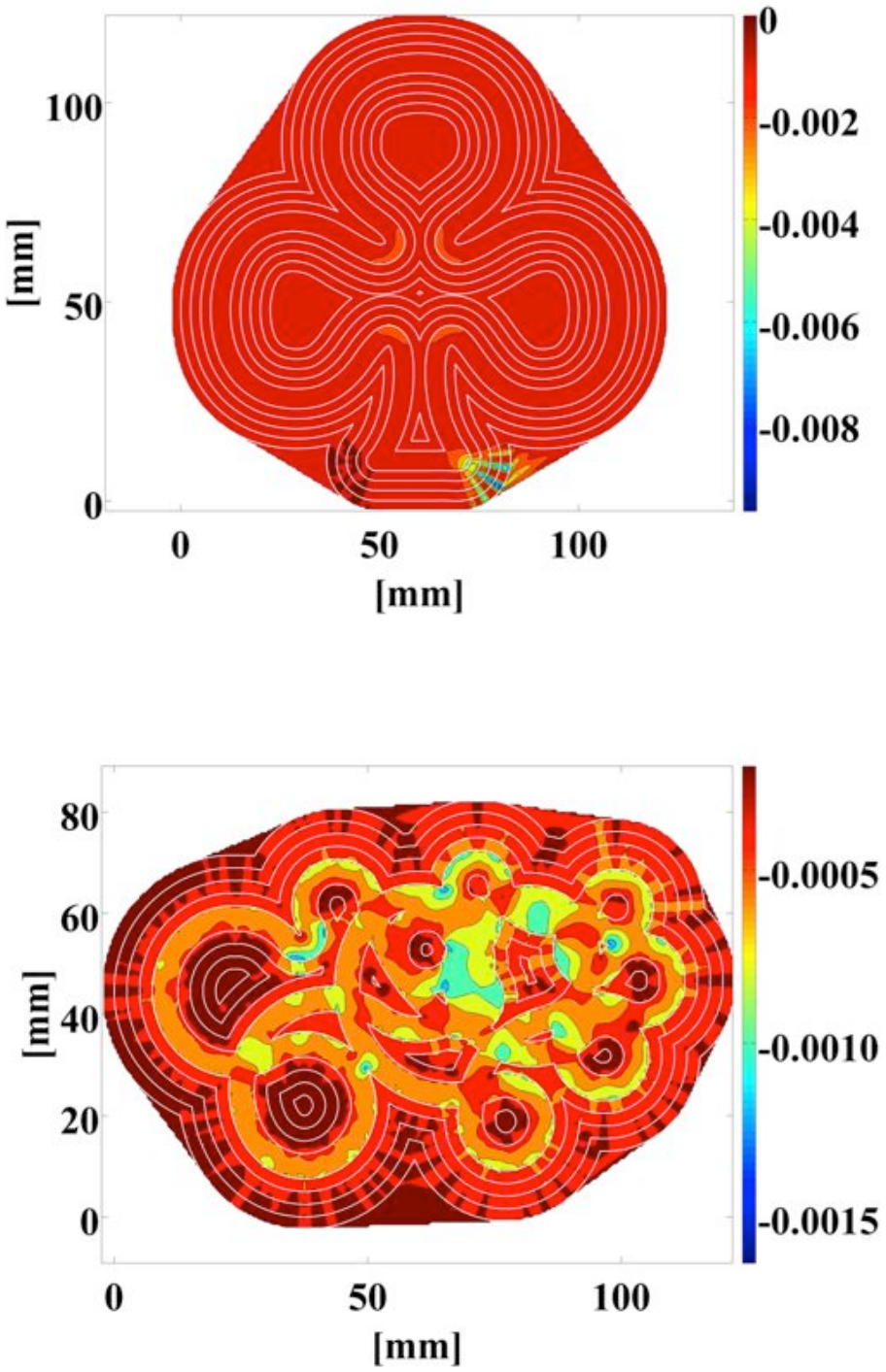


Figure 3-9 Geometric Errors on All Curve Offsets Produced by MOBS

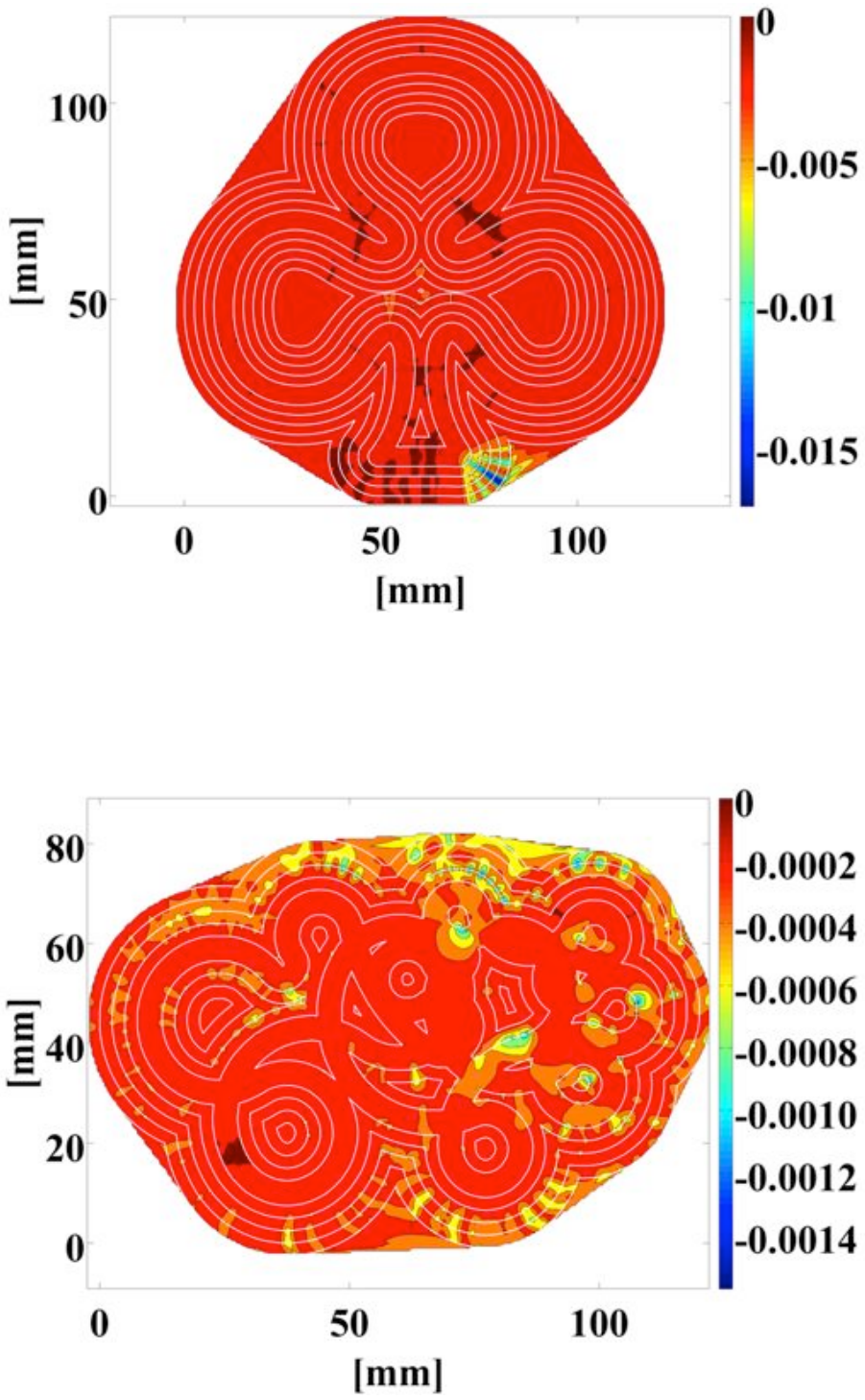


Figure 3-10 Geometric Errors on All Curve Offsets Produced by IMOBS

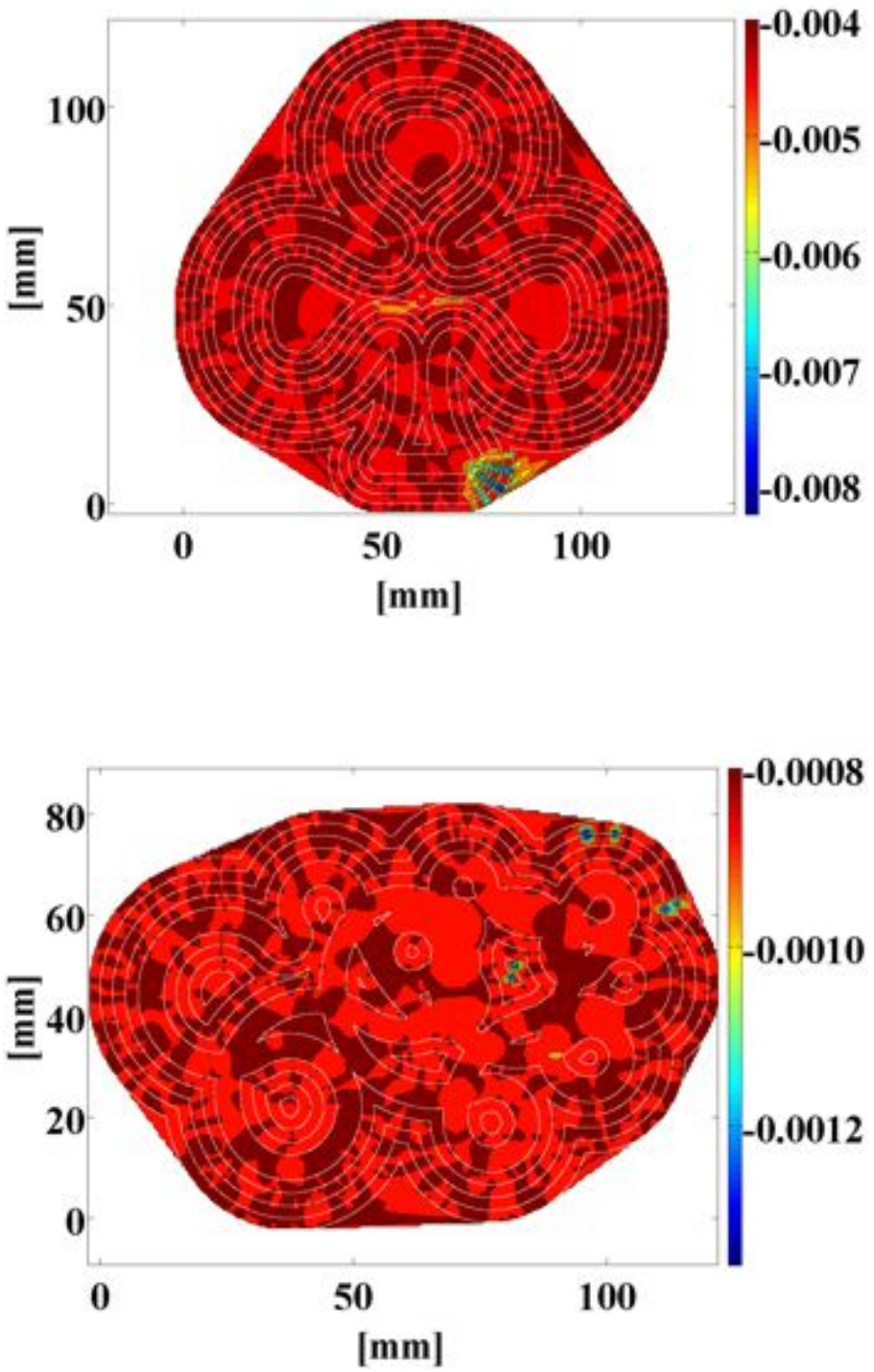


Figure 3-11 Geometric Errors on All Curve Offsets Produced by AMOBS

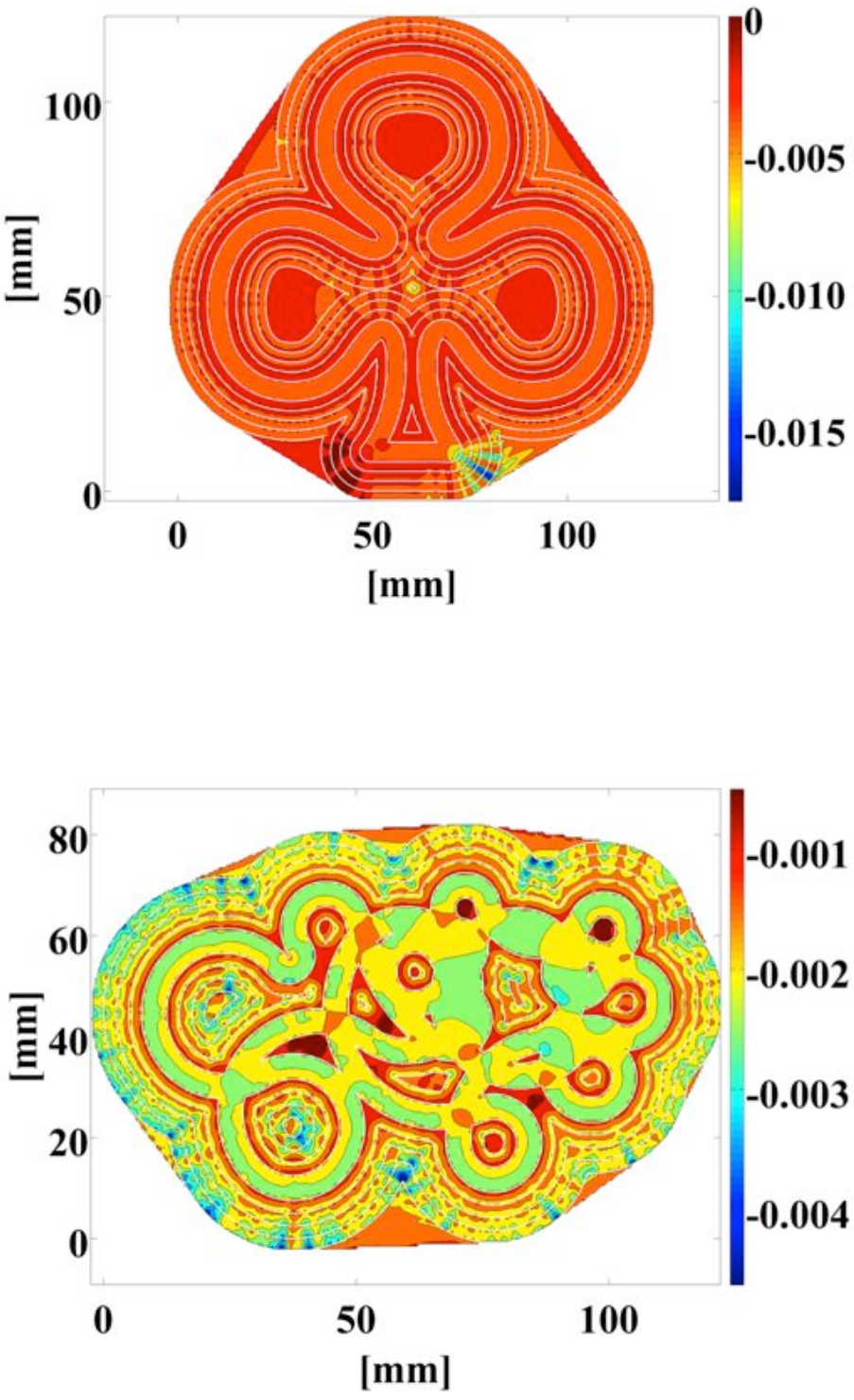


Figure 3-12 Geometric Errors on All Curve Offsets Produced by PO

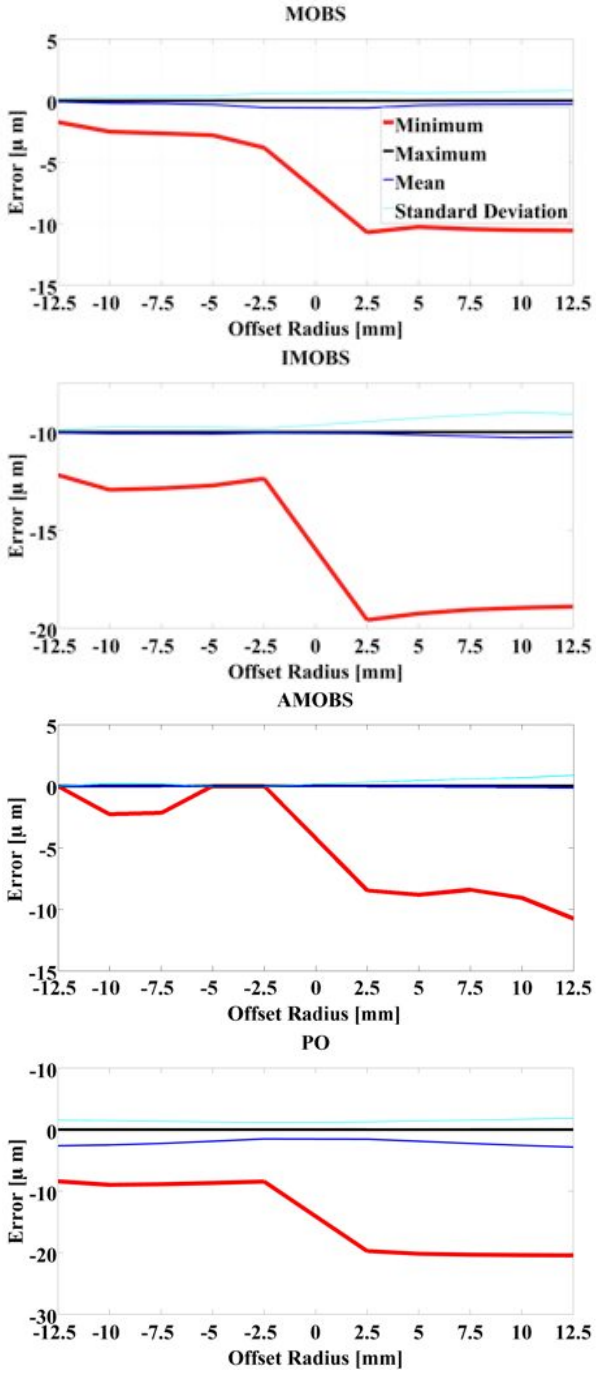


Figure 3-13 Statistical Attributes of Geometric Errors at Different Offsetting Distance for the Test Case Club

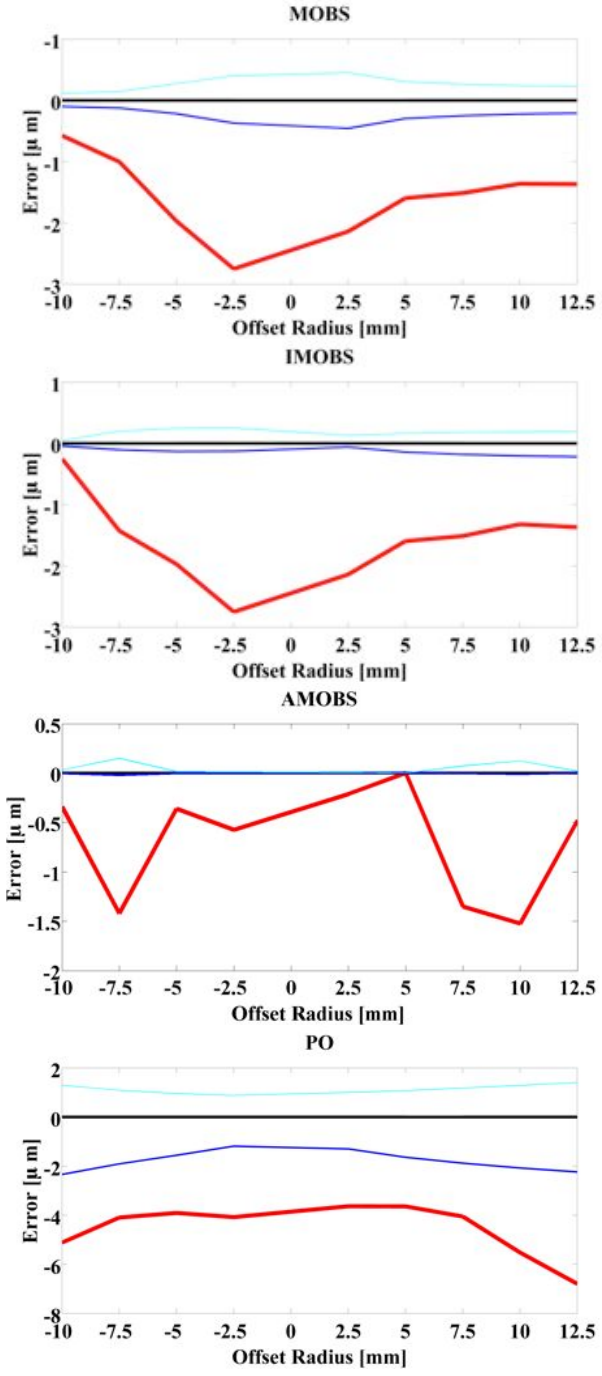


Figure 3-14 Statistical Attributes of Geometric Errors at Different Offsetting Distance for the Test Case Doodle

3.9 Discussion and Conclusions

In this chapter, various unconventional techniques are considered to generate curve offsets. The algorithms, which are analyzed in terms of run-time complexity and memory cost, are rigorously tested via two experimental cases. The key points of the study can be summarized as follows:

- All methods presented in this chapter employ the basic idea that the envelopes of a circular SE whose center sweeps across a generator curve constitute all curve offsets of that progenitor. This approach implicitly eradicates the need for iterative offset calculation at "sharp" turns on the base curve.
- Despite its ease of implementation, the MOBI method suffers heavily from the curse of dimensionality. For most real-world applications, the memory requirement of this technique, which could be on the order of TB, surpasses the resources of most computing platforms today.
- In MOBS, this memory problem is surmounted by employing the morphological operations on the boundary data sets (rather than the binary image itself). Even though the memory requirement for this has considerably decreased, the upper bound of its run-time complexity is found to be quadratic. However, the technique, which affectively handles all exceptional cases in COG, is still viable to generate offsets for practical applications where the size of tool trajectories is moderate.
- The performance of MOBS, to a certain extent, depends on the number of elements (N) on the structuring set (**T**). Even though N can be arbitrarily selected by the users, the CBS portion may yield unpopulated boundaries in the domain if the density of points in the structuring set is low. To overcome this problem, N needs to be adjusted dynamically based on the point density in the base set.
- The time complexity problem of MOBS is further reduced by the new technique proposed in this study. The enhanced method called IMOBS takes a different approach to generate feasible boundary points than its predecessor

and eventually employs a grid search technique to eliminate invalid boundary points in a global sense. The technique is found to be adequate for the tool trajectories with large number samples (>>100,000).

- The CCO portion of MOBS (and IMOBS) processes the boundary points in **S** sequentially to form curve offsets. Since this approach does not take into account the global distribution of the boundary points, some segmented curve offsets may occur for self-intersecting base curves. As a solution, the presented techniques combine these (isolated) curve patches through the cross-evaluation of the proximities among the extreme elements of various offset sets. To avoid this post-processing stage, one might consider re-sequencing of the points in **S** globally with the utilization of grid search techniques discussed in Section 3.5. However, this aspect is left open for future studies.
- After developing different curve offset generation algorithms, it is realized that there is no need to generate boundary points that are to be removed due to the local boundary criteria. AMOBS is developed on this purpose. It only generates boundary points that do not interfere with the neighbor base points. Beside this advantage of the algorithm, it also eliminates the risk of not generating boundary points at directions of the base curve.
- The last approach titled PO is based on Boolean operations on polygons where they have been widely applied to various engineering fields from computer graphics to geosciences. This study adopts this popular technique to generate curve offsets and evaluates the applicability of the method. Despite its high time complexity, the method has a significant potential for improvement. Unlike MOBS, this method is not sensitive to the selection of N (i.e. number of vertices on **T**) owing to the fact that apart from elimination of vertices (i.e. boundary points), the technique automatically creates new vertices at the intersections (of **T** and **Q**). Hence, the problem of unpopulated boundary regions is intrinsically handled. Furthermore, if tool polygon **T** is restructured to have an elliptical shape, the technique could easily accommodate the effect of the SE sweeping through the base points.

- When all the approaches discussed in the chapter are considered, AMOBS, as is, appears to be the most viable technique for hardware implementations and could be employed as an integral part of a discrete-time command generator for CNC machine tools.
- Except for MOBI, the techniques yield unevenly distributed points on the boundaries. Hence, one needs to resample spatially the extracted sequences to obtain uniformly distributed points on the resultant curve offsets.
- The numerical complexity analysis revealed that the theoretical complexity analyses are in agreement with the experiments. One of the most important things about the linearity of the algorithms (IMOBS and AMOBS) is that it tends to be quadratic when the size of the hash table is decreased. 256 by 256 is a good choice for both of the methods.
- Geometric errors of the methods are also calculated in order to compare with performances of the other methods. It is observed that AMOBS performs better in terms of geometric errors in both test cases. This is due to the fact that the possible boundary points are located intentionally in this method. The angles of the possible boundary points at each base point are different depending on the interference with the neighbor points.
- Since the algorithms behind the commercial CAD/CAM software packages (such as AutoCAD, CorelDRAW, PowerMILL, SolidWorks, etc.), which are known to deal with large generator curves effectively, are not revealed [85], the performances of the algorithms discussed in this chapter could not be compared to those of these packages in terms of run-time and memory cost (under same test conditions). It is important to note that most commercial CAD packages do fail to yield acceptable results for self-intersecting cases (such as Doodle). Consequently, once can postulate that if implemented in C++, the new technique IMOBS may achieve similar/better overall performance than those of the commercial software packages.

CHAPTER 4

DIRECT COMMAND GENERATION FOR CNC MACHINERY BASED ON DATA COMPRESSION TECHNIQUES

This chapter of the thesis presents a direct command generation technique for digital motion control systems. In this paradigm, higher-order differences of a given trajectory (i.e. position) are calculated and the resulting sequence is compacted via data compression techniques. The overall method is capable of generating trajectory data at variable rates in forward- and reverse directions with the utilization of a linear interpolator. As a part of the command generation scheme, the chapter also proposes a new data compression technique titled as ΔY10. Apart from this new method, the performances of the proposed generator employing different compression algorithms (such as Huffman coding, Arithmetic coding, LZW, and run length encoding) are also evaluated through three test cases. The chapter illustrates that the ΔY10 technique, which is suitable for real-time hardware implementation, exhibits satisfactory performance in terms of data compaction achieved in the test cases considered.

4.1 Introduction

Command generators (CGs) are indispensable components of digital motion control systems such as computer numerical control (CNC) units, industrial motion control cards, motor control modules, advanced servo-motor drivers, etc. Their main function is to calculate/feed the reference positions (a.k.a. “set points”) on a prescribed trajectory to the discrete-time axis-motion controllers at the beginning of each sampling (i.e. servo-update) period.

For most production machinery such as CNC machine tools, robotic manipulators, coordinate measuring machines, and rapid prototyping machines, the motion of the end-effector/tool along with the accompanying machine functions is defined by control languages with different features and abstraction levels that comply with various standards such as ISO 6983 (EIA 274D) [86], BCL (EIA 494C) [87], STEP-NC (ISO 10303-238) [88], and DMIS [89]. For instance, the numerical control (NC) language as described by ISO 6983 (a.k.a. “G-codes”) is widely adapted in manufacturing industry and defines the trajectory in terms of line-, arc-, and complex curve segments (parabola, helix, spline, etc.). In a conventional CNC system, the task of the CG is typically to parse a NC code employing these primitives as well as the complementary data on fixtures and tools. During the interpretation phase, the CG needs to take into account several factors such as the geometry of the selected tool, interference along the impending path, default feedrate, acceleration/deceleration limits of the machine axes, etc. Once the parameters of particular section on the trajectory are extracted, a real-time (RT) interpolator is invoked to produce *feasible* reference/command signals such as position, velocity, and/or acceleration as required by the axis controllers.

A simple solution to the disadvantages described in the introductory chapter is to generate the reference positions in an offline fashion by sampling the tool trajectory at the servo-update rate for a given set of tools/conditions. Once the resulting data are stored in the main memory of the controller unit, the tool reference positions could be directly retrieved by the axis controllers. This scheme, in turn, not only simplifies the control architecture but also increases the

efficiency of the CG. Despite the fact that this direct approach normally yields a huge amount of data to be stored, the data size becomes hardly a problem owing to the fact that versatile memory devices (SDRAM, SRAM, EEPROM, SD cards, etc) with large capacity (> 1 GB) are nowadays widely available in the market at low costs. Furthermore, the motion command data in manufacturing/industrial applications are highly redundant: **i)** the tool trajectory is mostly symmetrical due to the symmetry of the workpiece; **ii)** the simple offsets among the successive tool paths frequently appear in manufacturing (machining, cutting, grinding, welding, 3D printing, assembly, etc) operations; **iii)** repetitious operations are prevalent in such tasks. Consequently, the size of raw data could be reduced via data compression techniques.

Another critical feature of this approach is that it effectively eliminates the NC programs, which serve as command transmission media residing at an intermediate level. Hence, the directly coded (and compressed) trajectory data can be regarded as universal/portable and do not require any modification from one machine tool to another. Consequently, the same code (i.e. file) could be run on all equivalent CNC machine tools. Note that the closest analogy to the presented approach is the audio MPEG Layer 3 (MP3) encoding/decoding format. The coded audio file could be played on almost all playback devices produced by different manufacturers. However, the MP3 decoders on these devices do generate the digital data (in two audio channels) in the same way.

For some CNC applications (which do not require any dynamic manipulation on the trajectory during operations); this approach may eliminate the need for preparing/interpreting NC programs which serve as intermediate command-transmission media. Hence, the directly coded trajectory data *as an entity/object* could be regarded as portable at the *controller level* and may be directly utilized by the machines with equivalent configurations.

Consequently, there is a potential for devising simple yet effective CGs for industrial motion control systems by fully taking advantage of the current state-of-the-art. Hence, the main motivation of this chapter is to look deeper into this

aspect that has not been fully explored in the technical literature and to develop a relevant CG paradigm for a wide spectrum of computer controlled machinery.

The remainder of the chapter is structured as follows: Section 2 introduces the proposed command generation method. Section 3 discusses the compression performance of the method and compares with the commonly used compression algorithms. Section 4 focuses on the linear interpolator utilized in the command generator. Section 5 discusses the applicability of Markov Chains for modeling the command trajectory data and the chapter is concluded with Section 6.

4.2 Proposed Method

The method adopted in this chapter is a natural extension of the ones presented by [28]-[29]. The technique specifically relies on the lossless compression (and decompression) of the higher-order differenced data. Figure 4-1 illustrates the overall block diagram of the proposed CG scheme where the computer aided manufacturing (CAM) software along with a special post-processor are utilized to generate the commanded trajectory (i.e. position) directly. Note that for most CNC machine systems employing digital position sensors (such as linear/rotary optical position encoders); the reference trajectories, which must satisfy C^n ($n \in Z^+$) continuity, can be conveniently represented as signed-integer (i.e. position encoder count) sequences. The method in this chapter is developed to exploit such temporal sequences to produce reference trajectories efficiently even on a control system with modest resources in real-time. The details of the paradigm follow.

4.2.1 Differencing

Since the servo-loop update rates for industrial motion controllers are continuously on the rise (>10 kHz), the direct storage of the lengthy command sequences becomes an unviable feat even with the large (>1 GB) memory devices (SRAM, SDRAM, EEPROM, SD Cards, etc) used in the current state-of-the-art.

For a five-axis CNC machine tool with a servo-update (sampling) rate of 16 kHz, the memory required to store a 32-bit position sequence (per hour) becomes 16000 [samples/s] × 5 axes × 4 [bytes] × 3600 [s/hour] ÷ 1024^3 [bytes/GB] ≅ 1 [GB/hour]. Furthermore, transferring such bulky sequences to the memory devices on the controllers (or motor drivers) through a standard serial communication interface (such as RS-422/485, Ethernet/UDP, CAN, SERCOS, etc.) could also take considerable time (minutes).

Relative data encoding methods [27], which involve higher-order differences of discrete-time integer sequences, can help decrease the memory space for storage as the magnitudes in the differenced data set tend to drop off substantially in typical industrial applications. For instance, the higher-order differences of a command sequence {y(k)} can be expressed as

$$\begin{aligned} \nabla y &= y(k) - y(k-1) = (1-q^{-1})y(k) \\ \nabla^2 y &= \nabla y(k) - \nabla y(k-1) = (1-q^{-1})^2 y(k) \\ &\vdots \\ \nabla^n y &= \nabla^{n-1} y(k) - \nabla^{n-1} y(k-1) = (1-q^{-1})^n y(k) \end{aligned} \tag{4-1}$$

where k is the time index; $\nabla^n y$ is the n^{th} order difference while q^{-1} denotes the backward-time shift operator. How the range of differentiated data changes as a function of the order (n) certainly depends on the context of the application (i.e. nature of the information source). For instance, [28] investigate the effect of order by considering various motion command sequences for a 6 degree-of-freedom (DOF) robotic manipulator. As illustrated in Figure 4-2, [28] shows that the memory usage (i.e. magnitude/range of the differentiated data) tends to increase due to the fact that the sign of each data point frequently changes in an alternating fashion after the fourth order difference. The best solution (in terms of memory requirement) is usually achieved when the order is 3 or 4 for such applications.

Once the differentiated data along with the initial values are provided; the original data can be conveniently extracted using a number of cascaded accumulators as

shown in Figure 4-1. In this scheme, lower-order differences, which are commonly utilized in the feedforward controllers of the advanced motion controller topologies [90], can be also computed as by-products. In fact, without the computation of the intermediate differences, the original sequence can be directly generated by using the following finite difference equation:

$$y(k) = (1 - q^{-1})^n x(k) \tag{4-2}$$

where $x(k) \equiv \nabla^n y(k)$. However, if zero initial conditions [i.e. y(-1) = y(-2) = ... = y(-n) = 0] are considered, the start-off values of x(i) for $i \in \{0, 1, ..., n-1\}$ must be calculated by (4-1) accordingly. It is interesting to note that one of the most successful approaches in the NC technology was to generate the tool trajectory in an incremental fashion with the utilization of digital integrators (see [1]). Hence, the proposed scheme can be regarded as a revisit to this former technique.

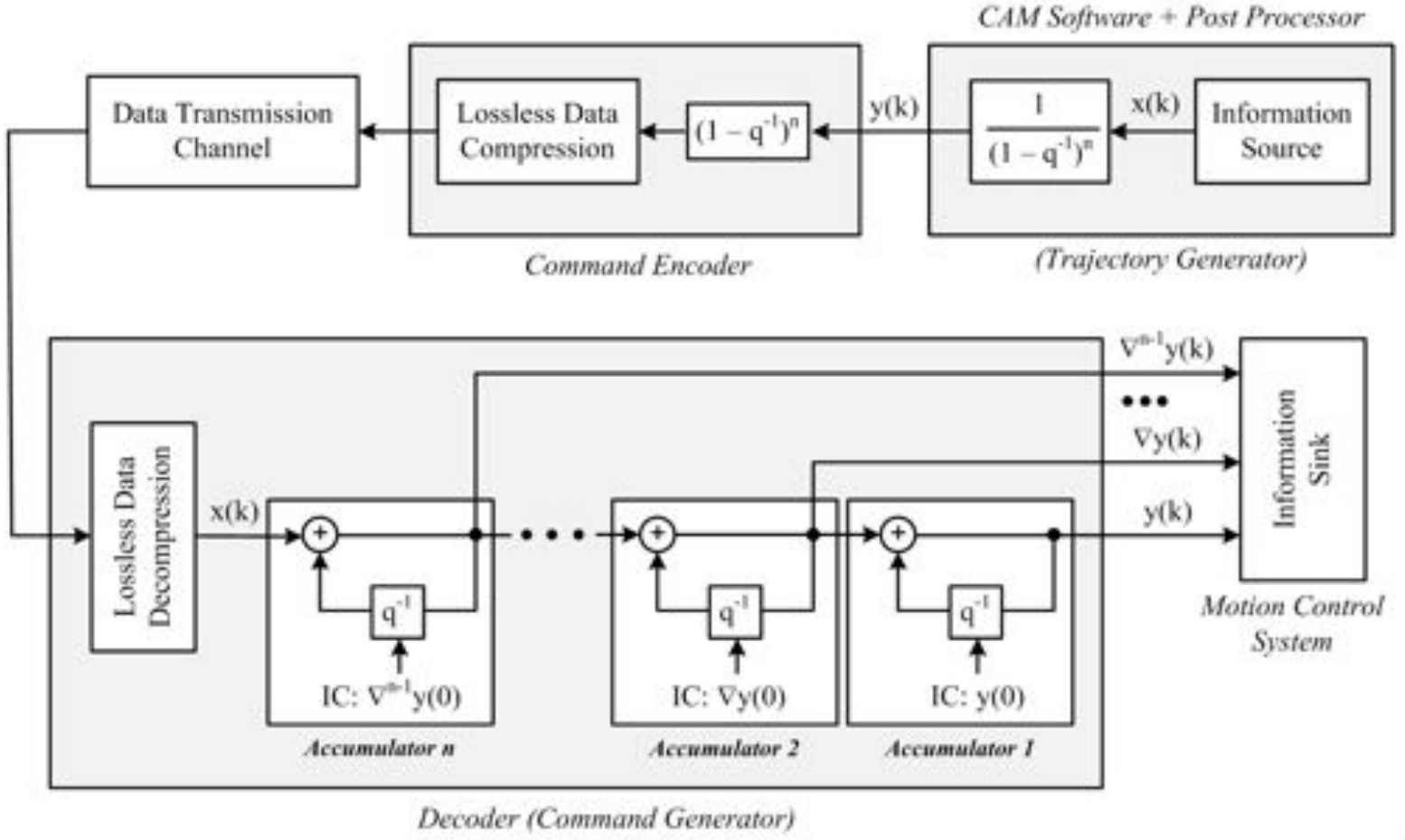


Figure 4-1 Simplified Block Diagram for the Command Generation Scheme

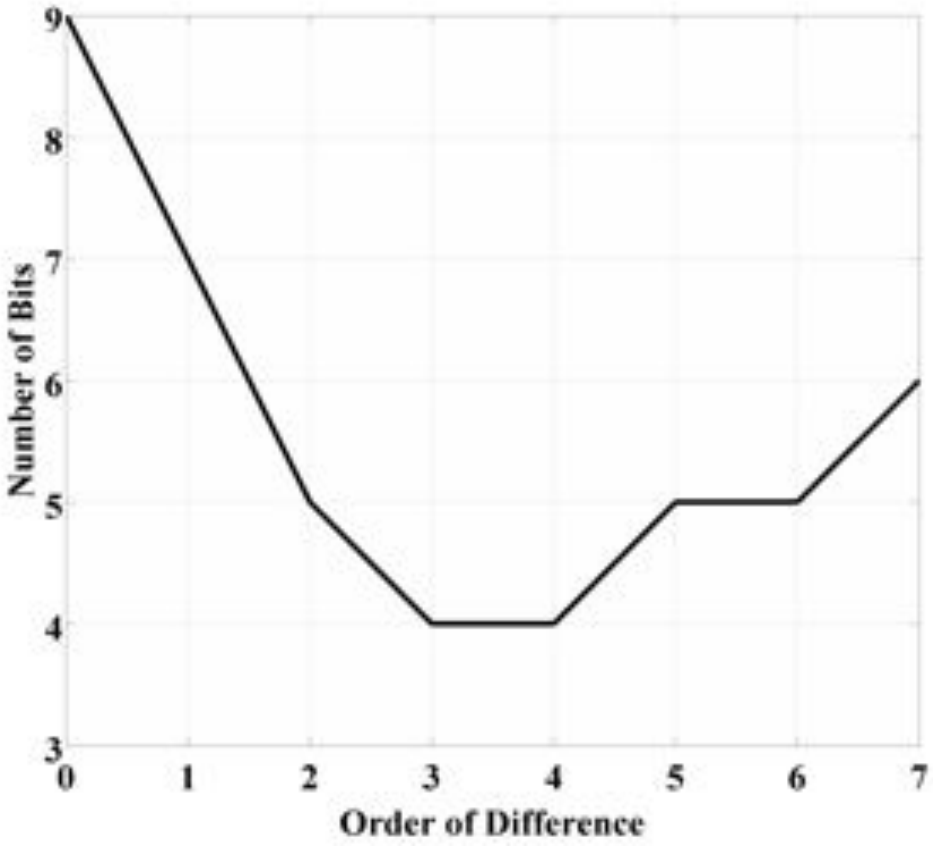


Figure 4-2 The Effect of Order on the Range of Differentiated Sequence [28]

4.2.2 Data compression/decompression via ΔY10 technique

Despite the fact that differencing can reduce the size of the original command sequence, the processed data, which have considerably redundancy, can further be compacted through universal (lossless) compression algorithms such as HC [91], Arithmetic Coding (AC) [92], and Lempel-Ziv-Welch (LZW) [93]. As an alternative, a simple (variable-length) compression technique (called **ΔY**) is adapted in this chapter. This relative encoding technique, which was originally

proposed by [28], has continued to evolve in time [29]. The one proposed in this chapter (called ΔY10**)** is an enhanced version that encodes the repeated (zero) patterns found in the target sequence (i.e. carries out run length encoding of zeros). The basic idea behind this paradigm is that when the higher-order differences of a reference trajectory (i.e. position sequence) is obtained; the (integer) values in the resulting sets do decrease considerably. Since most motion control applications require constant velocity along the traced trajectory, the majority of the differentiated data is likely to be zero while the rest is composed of small integers in which the probability of occurrence is inversely correlated with their magnitudes. Considering that the representation of a small integer number would require fewer bits, the difference data would take up significantly less memory if compared to the original data set.

Unlike entropy-coding based (general) compression techniques like HC, one can directly encode the difference data in this technique without calculating the probability of occurrence for the processed data and/or creating a corresponding dictionary (i.e. a binary tree) owing to the fact that the special requirements associated with the motion control applications (due to operational concerns) tightly dictate the statistical attributes of the data beforehand.

4.2.2.1 Encoding Process

The compressed code in ΔY10 technique is divided into five fields: **i)** Header; **ii)** Amplitude; **iii)** Sign; **iv)** Length; **v)** Zero. In this technique, the Amplitude Field (AF) encodes the absolute values of the data (i.e. unsigned integers) sequentially as Variable-Length Binary Numbers (VLBNs) whereas the Sign Field (SF) encodes sign bits of the sequence: 0 and 1 refer to positive- and negative numbers respectively. If the magnitude of a sample is zero, no sign bit is assigned for this case. Since the resulting binary sequence in the AF constitute VLBNs in order, another field called Length Field (LF), which yields the bit-length of each value in the AF, needs to be formed to extract the data. This field simply contains

sequences of 1's and 0's in an alternating manner. The bit-length of a particular number in the AF can be detected by simply counting the bits in between two consecutive transitions (0-to-1 or 1-to-0) in the LF. Note that unlike ΔY, a fourth field called Zero Field (ZF) is utilized to represent the number of consecutive zeros (i.e. so-called natural elements in delta encoding) encountered in the series. That is, the number of zeros is coded as a VLBN inside this field. The bit-length of each VLBN in this field is simply indicated by the number of zeros appended to the AF. Figure 4-3 illustrates a typical encoding process. In this technique, the memory (M) needed to represent a zero sequence (in bits) becomes

$$M = 3 \cdot ceil\left\{\log_2(N_z + 1)\right\} \tag{4-3}$$

where ceil{·} is the ceiling function and N_z (>1) denotes the number of zero elements in the sequence. Considering that the original ΔY technique requires $2N_z$ bits to describe the same sequence, the reduction in memory space could be quite significant for even small zero sequences. Lastly, the order of differencing, the initial conditions of the accumulators, along with the length of each individual field are stored as fixed length binary numbers in a special field called header. The pseudo-code given in Table 4-1 describes the encoding process of the elaborated technique.

4.2.2.2Decoding Process

The decoding process of ΔY10 method is even simpler than encoding. Since the data residing in the AF are coded as VLBNs, the length of each number must be decoded first by counting the bits in between two consecutive bit transitions encountered in the LF. After the absolute value of a particular number is obtained from the AF, the corresponding sign is simply fetched from the SF to produce the corresponding number as a signed integer. Note that if the value of a number is

found to be zero while its length (as indicated by the LF) is greater than 1, a number of consecutive zero elements are generated based on the number (of repetitions) decoded from the ZF. In the meantime, the Initial Conditions (ICs) are transferred to the accumulators so that the original sequence (i.e. position) can be generated by accumulating the extracted data in order. The pseudo-code shown in Table 4-2 elaborates this decoding process.

Sequence = { 12, -3, 1, -1, 0, 0, 0, -2, 1, 0, -5, ...}

Represented Number(s):	12	-3	1	-1	0,0,0	-2	1	0	-5	...
Amplitude:	1 1 0 0	1 1	1	1	0 0	1 0	1	0	1 0 1	...
Length:	1 1 1 1	0 0	1	0	1 1	0 0	1	0	1 1 1	...
Sign:	0	1	0	1		1	0		1	...
Zero Seq.:					1 1					...

Figure 4-3 Encoding of a Sample Sequence via ΔY10 Technique

Table 4-1 Pseudo-code for Encoding Process of DY10 Technique

```
Initialize empty (binary number) strings: AF,LF,ZF,SF; Let Q := 1₂; i := 1; j:=1;
D := (N)th order diff. of array Y (signed integer);
Calculate initial values:
  IC[1] := Y[1]; IC[2 := Y[2]-Y[1]; …
Find zero sequences in D and create two arrays: Z, ZI;
/* Z is an array storing the lengths of zero sequences while
array ZI stores starting addresses */
while (i ≤ length(D)) {
  if (D[i] ≠ 0) {
    amp := dec_to_bin(abs(D[i]));
    AF := AF + amp; m := length(amp);
    len := string with m number of Q;
    if (D[i] > 0)
       {SF := SF + 0₂;}
    else
       {SF := SF + 1₂;}
  }
  else {
    if (i = ZI[j]) {
      k := ceil(log(Z[j]+1)/log(2));
      amp := string with k number of 0₂; AF :=  AF + amp;
      len := string with k number of Q;
      ZF := ZF + dec_to_bin(k);
      i := i + Z[j] - 1; j := j + 1;}
    else {
      AF := AF + 0₂; len := Q;}
  }
  i := i + 1; LF := LF + len; Q := not(Q);
}
Convert AF, LF, RF, SF to integer number arrays;
```

Table 4-2 Pseudo-code for Decoding Process of ΔY10 Technique

```
Convert int arrays to binary number strings: AF,LF,ZF,SF;
LF := LF + not(LF[end]);
Initialize D and Y arrays;
Let Q := 12; i := 0; j := 1; k := 1; cs := 1; cz := 1;
while (j ≤ length(LF)) {
  i := j;
  while (LF[j] = Q) {j := j + 1;}
  Q := not(Q);
  seq := portion of AF string lying between i and (j-1);
  L := j – i;
  amp := bin_to_dec(seq);
  if (amp > 0) {
    if (SF[cs] = 02) {
       D[k]:=amp;}
    else {
       D[k]:= -amp;}
    cs := cs + 1;  k := k + 1;}
  else {
    if (i = (j-1)) {
      D[k]:= 0; k := k + 1;}
    else {
      seq := portion of ZF string lying between cz and (cz+L-1);
      len := bin_to_dec(seq); cz := cz + L;
      for i := 1 to len {
        D[k] := 0; k := k + 1;}
    }
  }
}
Using IC and D arrays, compute Y by successive accumulations;
```

4.2.3 Linear Interpolation

In CNC applications, the speed (i.e. feedrate) through the course of motion is generally modified by external input (like feedrate override). Under some extreme cases (such as the control scheme of an electro-discharge machine), it might be desirable to reverse the direction of motion as dictated by an external device. Therefore, the proposed CG method is to be augmented to accommodate a variable feedrate input. Figure 4-4 depicts the block diagram of the proposed CG as a system-on-chip (SOC) application.

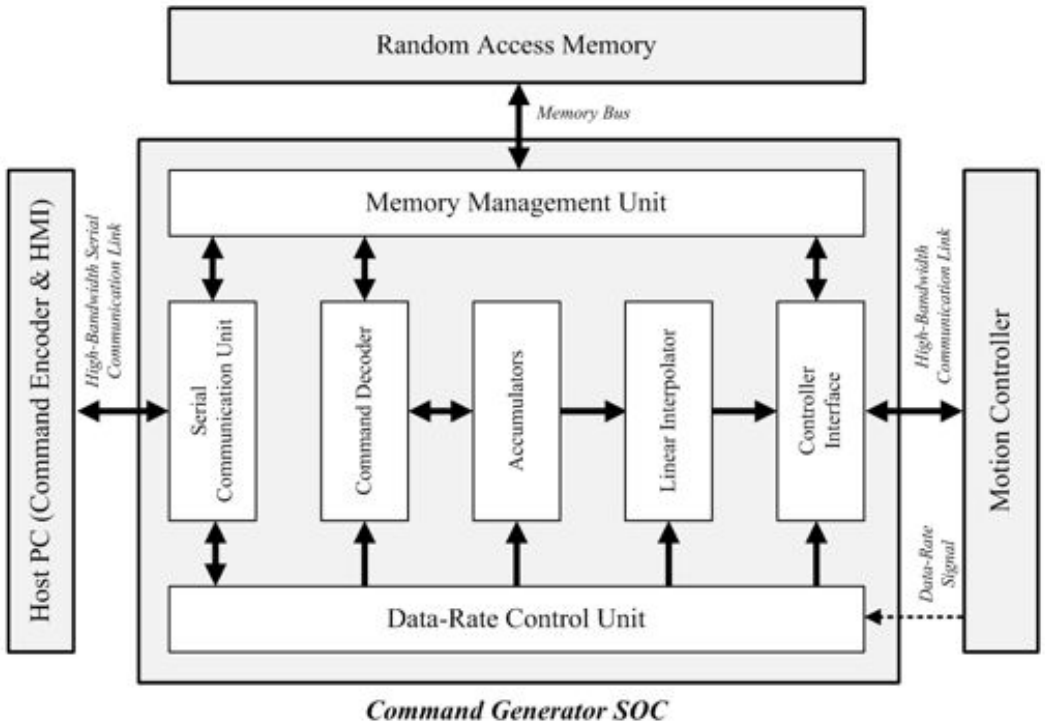


Figure 4-4 SOC Implementation of the Proposed Command Generation Paradigm

With this property, the users will be able to change the rate of command generation in both (forward and reverse) directions. During generation phase, since there is a need for the intermediate command values, a linear interpolator should be incorporated to the design. That is, this unit is to interpolate between the two decoded command values based on the following expressions:

$$a_k = a_{k-1} + f_k \left(\operatorname{mod} f_{\max} \right) \tag{4-4a}$$

$$m := \begin{cases} m-1, & a_{k-1} + f_k < 0 \\ m+1, & a_{k-1} + f_k > f_{\max} \end{cases} \tag{4-4b}$$

$$u_k = u_{m-1} + \frac{\left(u_m - u_{m-1} \right) a_k}{f_{\max}} \tag{4-5}$$

where u represents the decoded commands at the interval $m \in \{0, 1, \ldots, N\}$; k is the time index. Similarly, $f_k \in \{-f_{max}, \ldots -1, 0, 1, \ldots f_{max}\}$ indicates the current value of the feedrate input to the system while $f_{max} \in Z^+$ denotes the maximum feedrate at which commands could be generated. Note that the variable (a_k) in (4-5) essentially serves as a time scaling factor.

4.3 Compression Performance

To study the feasibility of the proposed CG, three sample applications are taken into account:

1. Stencil cutting of a roundabout road sign via 6-DOF robotic manipulator (i.e. Unimation PUMA 560).
2. Stencil cutting of a roundabout road sign via a three-axis CNC router.
3. Finishing of a plastic injection mold for a shampoo bottle using a high performance CNC vertical machining center.

The trajectories generated for the above-mentioned applications are illustrated in Figure 4-5 while their important attributes are summarized in Table 4-3. As can be seen from the table, the differenced data set cannot be represented less than 11 bits (or 21 bits depending on the case). Trajectories shown in Figure 4-5 are represented as integer number sequences in which the tool positions are essentially registered as encoder counts.

Table 4-3 Attributes of the Test Cases Considered

Test Case:	**1**	**2**	**3**
Number of Axes	6 (revolute)	3 (prismatic)	3 (prismatic) + PLC
Position Resolution	40000 cts/rev	1000 cts/mm	4000 cts/mm
Samples / Axis	1575	1575	904294
Sampling Period [s]	0.05	0.05	0.001
Command Duration [s]	78.75	78.75	904.294
Range of Data [Byte]	3	4	3 (4 bytes for PLC)
Range of Data (∇) [bit]	15	21	11
Range of Data (∇^2) [bit]	16	21	11
Total Size of Data [kB]	27.686	18.457	11480

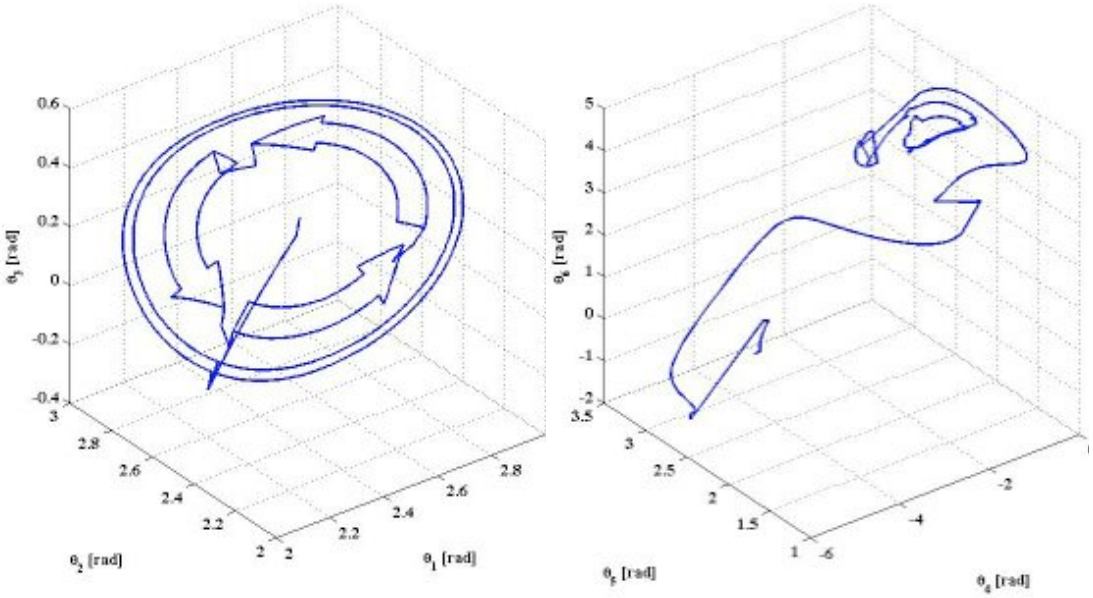
For the Case 3, (apart from axis-drives), the auxiliary units of the machine (such as the spindle drive, the coolant control unit, the automatic tool changer, the tool clamp, etc.) must be commanded during the machining operation. To that end, 32-bit data channel, which is referred to as Programmable Logic Controller (PLC) channel, is added as the fourth axis. In this channel,

16-bit signed integer represents the current spindle speed with one revolution-per-minute (rpm) increments;

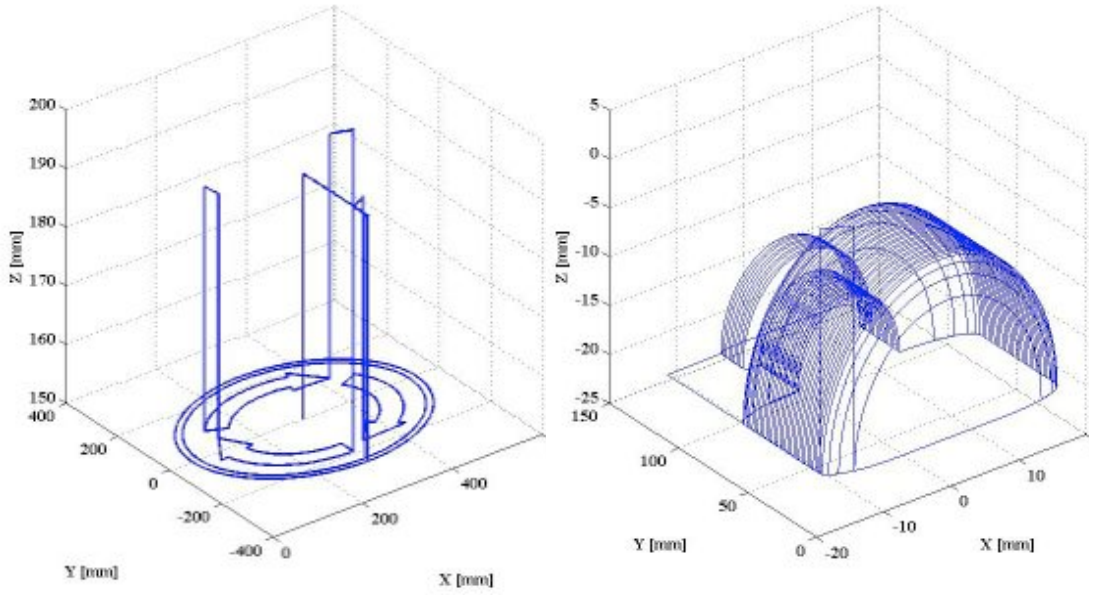
- 8-bit unsigned integer stands for the current position of the tool magazine (or current tool in use);
- 8-bit unsigned integer is allocated for various auxiliary functions such as coolant on/off, tool clamp on/off, etc.

Apart from ΔY, ΔY10, HC, and AC, two popular compression techniques are also considered to access their performances on the test cases:

- LZW, which is categorized as a dictionary-based compression technique, is implemented using the original algorithm proposed by [93]. To apply the technique, the differentiated data are first mapped (or re-sequenced) as unsigned (8-bit) integers. The encoding process simply utilizes a 16-bit (static) dictionary where the first 256 locations are initially occupied by the distinct elements of the resulting sequence. As the dictionary (table) is filled up with new entries, 16-bit pointers (or indices) are sent to the output stream to form the compressed code. Note that since the capacity of this 16-bit dictionary has never been exhausted for all the test cases (especially, the Case 3), well-known paradigms to manipulate the dictionary dynamically (such as resetting, removal/replacement of entries, allocation of new resources, implementation of a cyclic buffer, etc.) are not taken into consideration in this study.
- A Modified Huffman Coding (MHC) technique, which utilizes Run-Length Encoding of Zeros (RLEZ) followed by Huffman encoding, is adapted owing to the fact that the probability of occurrence for zero is relatively high when the sampling frequency is increased. In this technique, the data is preprocessed to encode the number of zeros encountered in the sequence [27]. After computing the first-order probability distributions, the resulting sequence is encoded via the HC technique.



(a) Case 1



(b) Case 2 **(c)** Case 3

Figure 4-5 Command Trajectories for the Studied Cases

In this study, the discrete information source shown in Figure 4-1 is assumed to be a (memoryless) stochastic process that outputs a symbol (i.e. integer) $x(k) \in \aleph$ based on some probability distribution at equal time intervals (T). The source alphabet $\aleph$ usually shrinks to {-1, 0, 1} [count/s^n] at very high sampling rates. Hence, the first-order probability distributions of the sequences (to be compacted) are utilized by the HC and AC techniques.

Note that all compression algorithms are implemented in MATLAB 2010a where special toolbox functions for HC and AC are available. For the considered cases, Table 4-4 through Table 4-6 demonstrate the compression ratios achieved by these methods as a function of differencing order (n). Note that the compression ratio (r) in the tables is defined as

$$\% \, r(n) = 100\left[\frac{N_c(n) + N_s(n)}{N_0}\right] \tag{4-6}$$

where N_c is the size of compressed data [Byte]; N_s denotes the size of supplementary data [Byte] needed to decode the original sequence [i.e. code dictionary (or any relevant data to interpret compressed code), initial conditions of the accumulators, etc]; N_0 is the size of the original command sequence (position) [Byte]. In other words, $N_c + N_s$ refers to the size of the complete (minimal) data set to extract the initial command sequence without any loss.

Table 4-4 Compression Ratios (%) of Various Techniques for Case 1

Each cell inside the 3-by-2 tables is allocated to a particular technique as shown on the right.

HC	**AC**
LZW	**MHC**
ΔY	**ΔY10**

Joint Axis	1		2		3		4		5		6	
6	74	**72**	32	30	36	34	48	44	62	56	79	71
	79	73	39	33	71	38	51	49	62	62	75	78
	72	**72**	**24**	25	**27**	28	**35**	**35**	**45**	**45**	**56**	**56**
5	59	**58**	23	22	32	30	44	41	58	52	73	66
	76	**58**	44	26	78	33	55	45	69	59	82	73
	69	69	**21**	**21**	**25**	26	**34**	**34**	**43**	**43**	**54**	**54**
4	72	**70**	32	30	36	33	49	45	61	56	78	70
	78	71	39	36	70	39	51	49	62	62	75	77
	72	72	**25**	**25**	**27**	**27**	**35**	**35**	**45**	**45**	**56**	**56**
3	58	**56**	19	**18**	27	25	38	35	50	45	64	57
	77	58	43	21	76	29	54	38	68	51	81	64
	65	65	20	21	**24**	25	**33**	**33**	**41**	**41**	**53**	**53**
2	53	**52**	18	**17**	26	25	36	34	49	44	62	56
	73	**52**	43	22	77	29	54	37	67	49	80	62
	68	68	19	20	**24**	**24**	**32**	**32**	**41**	**41**	**52**	**52**
1	**53**	**53**	19	**18**	26	25	37	35	48	44	62	56
	38	**53**	51	23	89	29	54	37	68	49	81	61
	67	67	20	20	**24**	25	**32**	33	**41**	**41**	**53**	**53**

Order of Difference

Table 4-5 Compression Ratios (%) of Various Techniques for Case 2

Each cell inside the 3-by-2 tables is allocated to a particular technique as shown on the right.

HC	AC
LZW	MHC
ΔY	ΔY10

Axis		1		2		3		4		5		6	
	Z	5	2	4	**1**	5	**2**	5	3	6	3	6	4
		4	**1**	5	**1**	5	**1**	6	**2**	9	**2**	11	**2**
		16	9	12	4	15	8	18	11	21	14	24	18
	Y	10	**93**	82	76	39	37	18	**17**	25	**23**	30	**27**
		11	10	97	80	35	38	75	29	68	25	44	29
		10	10	**58**	**58**	**33**	**33**	19	19	24	**23**	30	29
	X	84	**79**	96	87	39	37	18	**17**	23	**22**	30	**28**
		99	81	10	94	53	38	18	19	40	24	54	30
		99	99	58	**57**	33	**32**	18	18	23	**22**	29	**28**

Order of Difference

Table 4-6 Compression Ratios (%) of Various Techniques for Case 3

Each cell inside the 3-by-2 tables is allocated to a particular technique as shown on the right.

HC	AC
LZW	MHC
ΔY	ΔY10

Axis		1		2		3		4		5		6	
	PLC	4	.001	4	.002	4	.003	4	.003	4	.004	4	.005
		.09	**.0007**	.09	**.0009**	.09	**.001**	.09	**.002**	.09	**.002**	.09	**.002**
		6	.001	6	.002	6	.002	6	.003	6	.003	6	.004
	Z	20	19	7	6	9	9	10	10	14	14	14	13
		6	17	**3**	15	**3**	12	**4**	17	**4**	13	**4**	19
		29	28	10	8	13	11	15	13	21	19	25	23
	Y	6	4	5	2	5	3	5	2	6	4	5	3
		.6	2	**.4**	3	**.4**	1	**.4**	4	**.5**	2	**.5**	4
		12	5	9	1	9	2	9	2	10	3	11	4
	X	19	19	7	6	10	9	11	10	15	14	14	14
		6	18	**3**	14	**4**	12	**4**	16	**4**	14	**4**	19
		30	29	10	9	13	12	15	14	21	19	25	24

Order of Difference

As can be seen from Table 4-4, the best compression performance for the first case is generally achieved when n = 2 owing to the fact that the increments of encoder counts from one sampling step to another (i.e. angular velocity) are still quite high as the robot performs a jerky motion throughout this particular trajectory where only C^0 continuity is maintained. The performances of ΔY10 & ΔY methods are generally better than the other methods. For n = 2, the command sequence can be compressed to about one-fifth of its original size. After the second order, there is an increasing trend in the compression ratios associated with almost every method due to the reasons outlined in 4.2.1.

For the second case summarized in Table 4-5, the best compression ratios are usually attained at the X- and Y axes when n = 4 while n = 2 suits best for the Z axis. As can be seen from Table 4-5, the AC, which is known to produce near-optimal rates for long sequences, exhibits the best performance while HC and MHC happen to be the closest contenders for this case. On the other hand, the LZW technique yields rather poor compression ratios in the first two cases where the command sequences are relatively short. This is due to the fact that the compression for the LZW method does commence after the presentation of a few hundred samples to construct a representative dictionary [27].

As for the third test case outlined in Table 4-6, the LZW clearly outperforms the other techniques due to the identification and efficient representation of redundant data (i.e. repeated patterns). AC and HC seem to achieve comparable compression ratios (regardless of the order and the axis of motion). For this long command sequence, the compression ratios of the entropy-coding based algorithms (HC, AC, MHC, ΔY, ΔY10) generally tend to approach to each other. Another conclusion to be drawn from this table is that after the second-order of difference, there are no remarkable changes in the compression ratios.

MHC technique exhibits rather poor performance when dealing with short command sequences like the ones in Cases 1 & 2 due to the overhead costs imposed by the run-length encoding on fixed length data. If compared to HC alone, there are only minor improvements in compression performance for Case 3

which includes large number of natural elements. However, the method is not too successful since these encoded "zero" sequences (which are in large quantities) are apparently shorter in lengths so that the gain of RLEZ is not justified for the test cases considered.

For easy comparative evaluation, the results given in Table 4-4 through Table 4-6 are summarized in Figure 4-6. The table shows not only the average compression ratios for the best differencing order but also the corresponding bit rates (bits/sample or bps) for each case. That is, this average compression ratio is defined as

$$\bar{r}(c) = \min_{n\in\{1,2,\ldots,6\}} \left\{ \frac{1}{N_a(c)} \sum_{a=1}^{N_a(c)} r(a,n,c) \right\} \tag{4-7}$$

where r is the compression ratio defined by (4-6); c ($\in \{1, 2, 3\}$) is the case number while $N_a(c)$ refers the total number of axes associated with a particular case c. Similarly, Figure 4-6 illustrates the performance index of each technique where the index J $\in$ [0, 1] is defined as $J(c) = 1 - \bar{r}(c)$.

Table 4-7 Overall Data Compression Performance of Various Techniques for Three Test Cases

Case		H(ℵ)	HC	AC	LZW	MHC	DY	DY10
	% $\bar{r}$		23.83	22.50	43.17	26.83	21.50	22.00
1	bps	2.532	3.812	3.602	6.912	4.292	3.442	3.522
	n		2	2	2	2	2	2
	% $\bar{r}$		13.67	12.33	33.00	16.67	18.33	16.00
2	bps	2.304	3.284	2.964	7.924	4.004	4.404	3.844
	n		4	4	4	4	4	4
	% $\bar{r}$		5.75	3.50	1.80	6.25	8.75	4.50
3	bps	0.832	1.382	0.842	0.432	1.503	2.102	1.082
	n		2	2	2	3	2	2
Mean	**% $\bar{r}$**		14.42	12.78	25.99	16.58	16.19	14.17

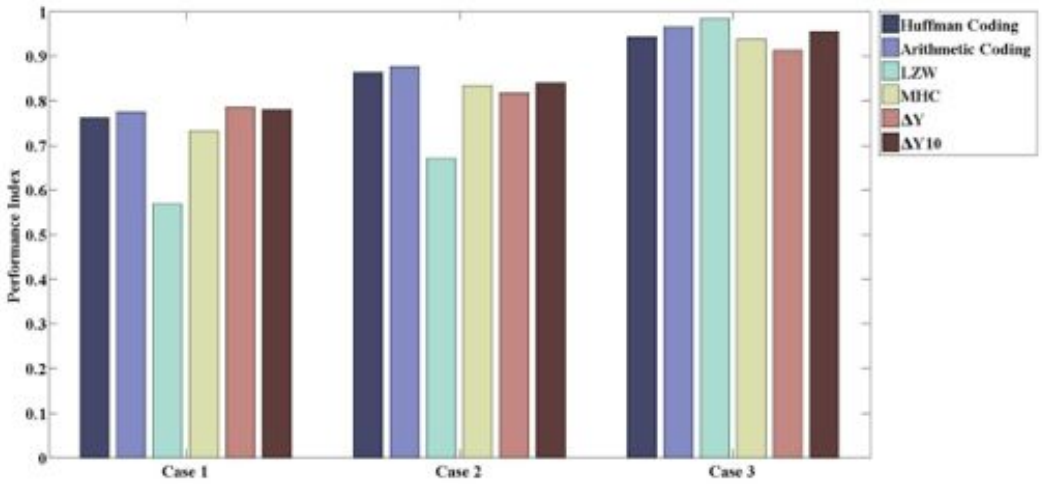


Figure 4-6 Performance Indices for Various Compression Techniques for Different Test Cases

Note that Table 4-7 also illustrates the (average) entropy rates for the differenced position sequences. In information technology, the entropy is employed as a measure for disorder/uncertainty (or average information content) associated with a random variable (or output of a stochastic process). The entropy (rate) of an information source naturally depends on its statistical properties and does impose a theoretical limit on achievable (lossless) compression performance (i.e. bit-length per sample). Based on the aforementioned assumptions, the entropy rate of the source (in bits/sample) can be estimated as

$$H(\aleph) = H(X) = -\sum_{x \in \aleph} p(x) \log_2 p(x) \tag{4-8}$$

where p refers to the probability mass function on X. It is critical to notice that the actual entropy rates of the sources are lower than the ones estimated by (4-8) in Table 4-7. As can be seen from this table, for Case 3, the bit rates associated with the LZW method, which successfully explores the redundancy in the given

data, does drop below the estimated entropy (of the source) owing to the fact that the assumptions on the information source may not be valid. That is, the (stationary) information source is likely to constitute memory and may be statistically modelled as an ergodic Markov process. Then, the information source can be presumed to generate a symbol x(k) whose probability of occurrence depends (or is conditional) on the previous symbol x(k-1). In that case, the entropy rate of the source could be expressed as

$$H(\aleph) = H(X_k \mid X_{k-1}) = -\sum_{x_{k-1}\in\aleph}\sum_{x_k\in\aleph} p(x_{k-1},x_k)\log_2 p(x_k \mid x_{k-1}) \tag{4-9}$$

where p denotes the joint probability distribution function [34]. However, to implement efficient data compression, developing suitable statistical models for the information source is left open for future studies.

Without the extensive experimentation and statistical analysis on the results; it is not possible to make definitive assessments on the "best" compression technique in this application domain. However, the ΔY10 method, which follows the AC technique in performance, yields acceptable compression ratios for all test cases considered. On average, the presented technique can compact long trajectory data with an overall compression ratio of 4.5% (Table 4-7, Case 3). If such a performance is attained, the memory for storing a typical (compacted) sequence that was elaborated in Section 4.2.1 turns out to be 1 [GB/hour] × 0.045 ≅ 46 [MB/hour]. The files (with such sizes) can be easily handled and managed in the today's technology and thus indicate the technical feasibility of the proposed method.

Despite the fact that (just like AC, HC) the time complexity of the ΔY method(s) is linear (i.e. O{N}), this simple technique inherently utilizes less hardware resources when implemented on a FPGA-based embedded system. To give readers a succinct idea about the implementation issues, the Table 4-8 summarizes

the number of 4-input Look-up Tables (LUTs) used to realize each technique [29], [94] - [96]. Although the presented technique in [29] includes extra modules such as accumulators and communication module, the ΔY method still expends the least amount of LUTs if compared to the other techniques. Note that the development/hardware implementation of ΔY10 based CG is currently underway. However, the resource utilization of the ΔY10 method is expected to be somewhat similar to that of its predecessor.

Table 4-8 FPGA Resource Utilization of Different Methods

Method	HC [94]	AC [95]	LZW [96]	ΔY [29]
No. of LUTs	3007	2714	1114	1105

4.4 Command Generation Performance with Variable Rate

As mentioned in Section 4.2, the proposed method, which incorporates a linear interpolator at its output stage, is capable of producing command sequences at variable rates as set by an external source. Hence, the performance of the method (utilizing ΔY10 as the data compression algorithm) on the third test case (discussed in previous section) is assessed for a normalized feedrate profile as shown in Figure 4-7. Note that the profile is formed such that all commands are first generated in the forward direction and then in the reverse direction with continuously changing scale factor.

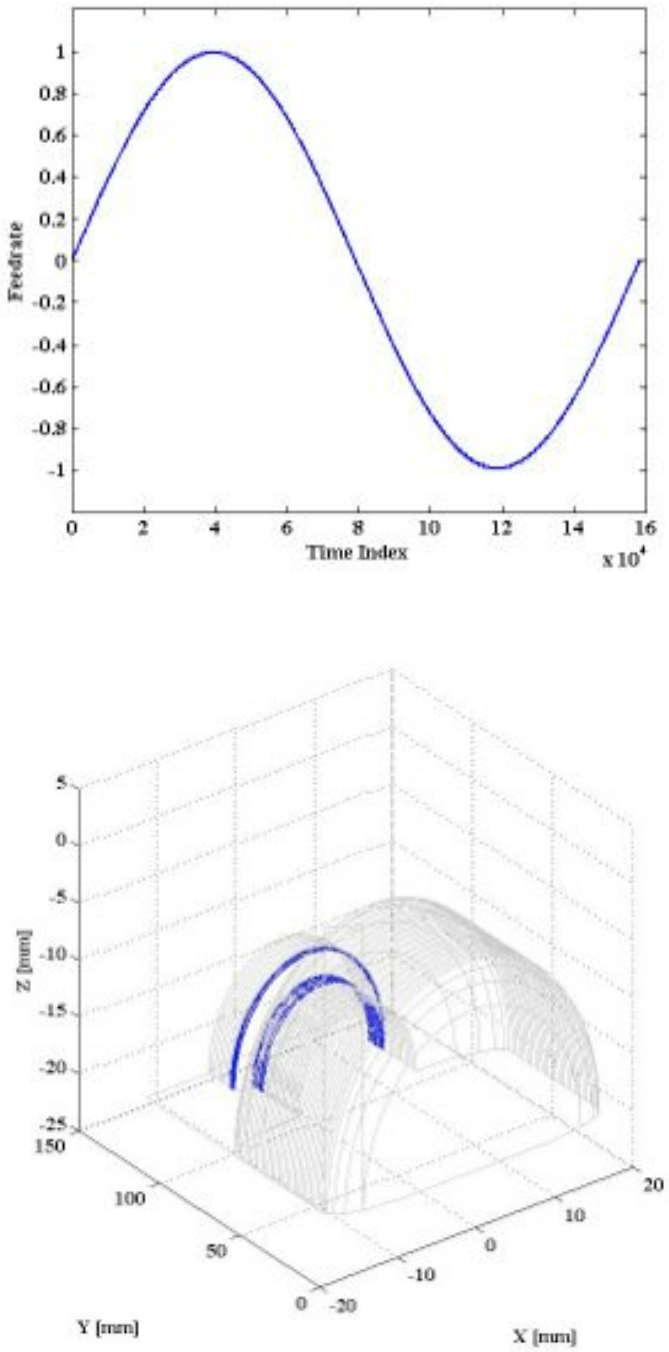


Figure 4-7 Normalized Feedrate Profile and Portion of Trajectory being Generated (Case 3)

Figure 4-8 illustrates the generated commands along the X, Y, and Z axes. Despite the fact that the linear interpolation yields acceptable results in between closely-spaced samples of time, the proposed interpolation scheme is known to create a significant representation error on the commanded trajectory known as *chord error*. Figure 4-9 demonstrates this error which is mainly caused by the data aliasing at the inflection points of the trajectory. Even though that magnitude of error is less than 15 encoder counts (0.4 microns) in all axes, a dynamic feedrate scheduling algorithm, which is the subject of an ongoing research, could be incorporated to the method to reduce (if not to eliminate) this error.

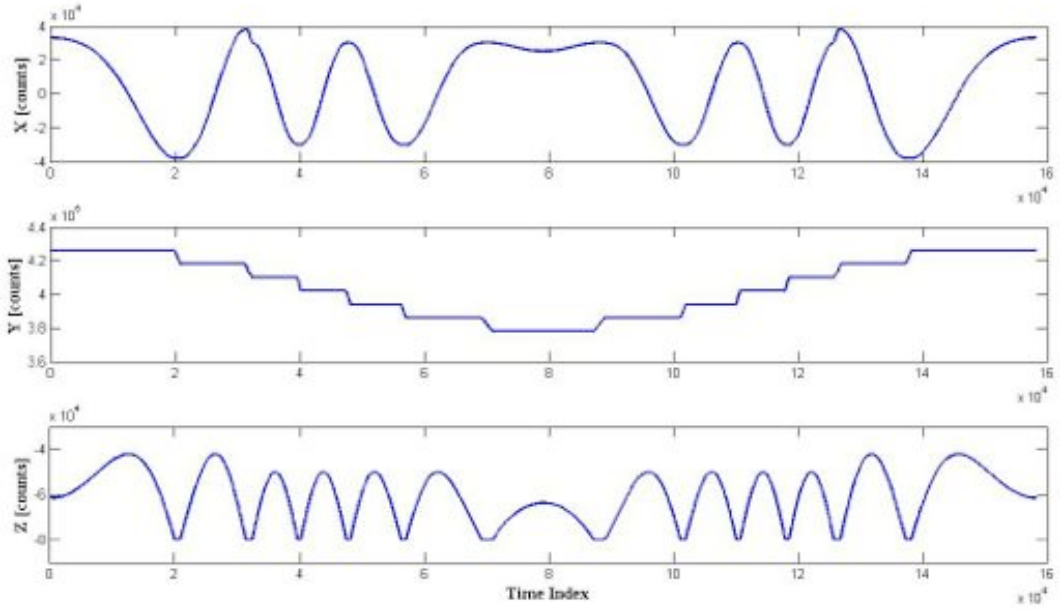


Figure 4-8 Interpolated Command Sequences

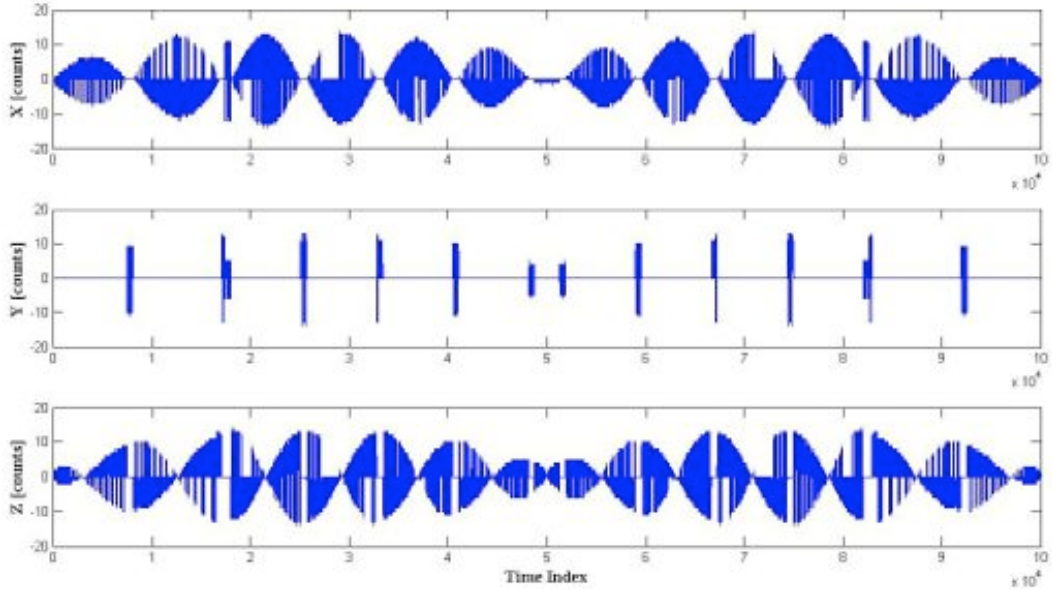


Figure 4-9 Chord Errors for Test Case 3

4.5 Modelling of Information Source via Markov Chains

In this section of the chapter, a different compression scheme is employed on the motion command trajectories composed of encoder counts. In this scheme, the higher order differences of motion commands are grouped with a predefined number of elements and then their occurrence probabilities are determined. As in most of the compression algorithms (HC, AC, MHC), the groups that have high probabilities are represented with less number of bits than the ones with low probabilities. A dictionary is also required to recover the command trajectory from the compressed dataset.

In the following subsection, the method is described in a detailed manner over a sample command trajectory. Then in the upcoming subsection, the method is employed on the same test cases utilized in this chapter of the dissertation and its performance is compared with the previously discussed algorithms. The effect of number of elements in the groups is also evaluated for the three test cases. Finally

the section is concluded with some remarks on the advantages and the disadvantages of the proposed scheme.

4.5.1 Proposed Approach

The idea of grouping different motion command values is similar to the construction of Markov matrices. In this scheme, the unique motion commands can be regarded as states of the Markov model. The probabilities of the groups are the state transition probabilities in the Markov model. If the number of elements in the group is selected as two, then the first order Markov chain model is to be constructed. As the number of elements in the groups is increased, the order of the Markov chain also increases. After the Markov chain is constructed, then the transitions with higher probabilities are represented with few bits.

The method is employed on the first 2000 commands of the x-axis of the Case 3 and is illustrated in Figure 4-10. As in the previous approaches, the higher order differences (3^{rd} for this case) of the given original motion command sequences are computed firstly. It is clearly seen that the magnitude of the commands do decrease tremendously. The initial values are stored to recover the original sequence after the compressed data is decompressed. By determining the unique commands and their number of occurrences in the differenced trajectory the first order Markov chain model is constructed. If each row of the constructed blue matrix is divided into the number of occurrences of the unique commands, one can get the Markov matrix. In the next step, a dictionary is formed to code the differenced motion command sequence. Since the number of elements in the group is determined to be two, groups whose probabilities are higher than zero are assigned to VLBNs. In these assignments, the same decimal values are assigned to different groups to improve the compression efficiency of the method. Since the lengths of the VLBNs corresponding to the same decimal value are different, there occurs no problem for the decompression stage. The compressed code constitutes of two fields. The couple field is the one where the VLBNs of the

groups of differenced motion commands reside. The second field is the same as the LF utilized in ΔY10 compression algorithm. The alternating manner of the LF helps the processor decode the Couple Field (CF) in a proper way. Given the compressed code and the corresponding dictionary, the differenced motion commands can be generated. After the utilization of integrators, the original command sequence is generated.

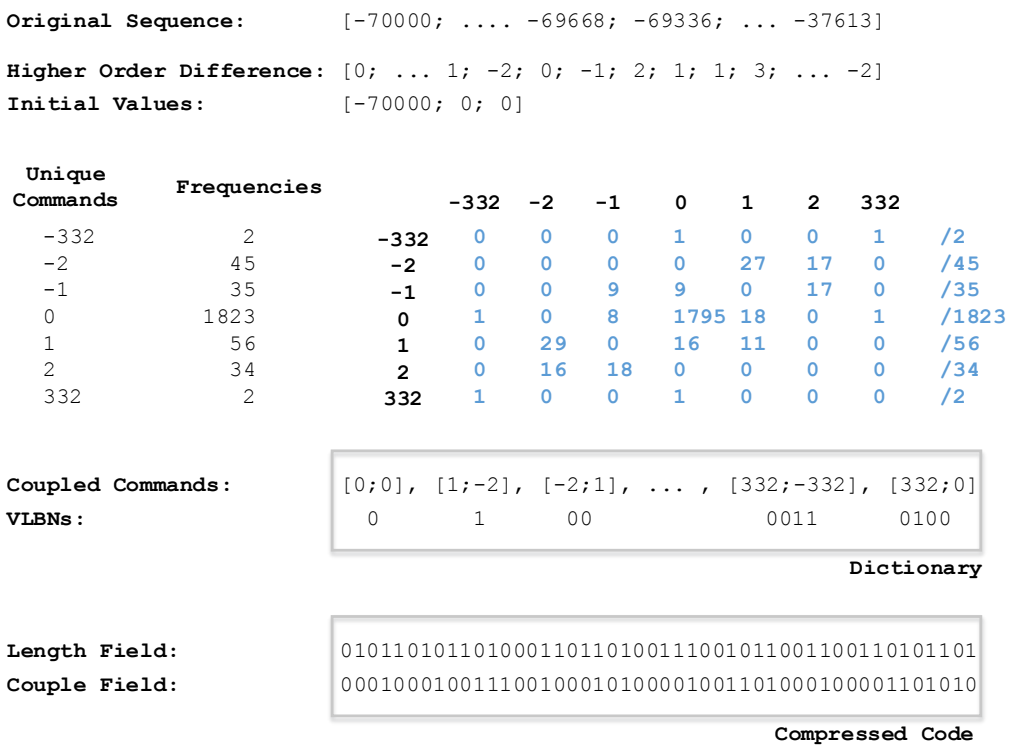


Figure 4-10 Information Source Modeling via Markov Chains (ISMMC)

4.5.2 Performance Evaluation

The three sample applications described in the Section 4.3 are used to determine the performance of the method proposed in this section. The compression ratios

are calculated using (4-6). The supplementary data in the equation constitutes of the initial values and the dictionary formed according to the frequencies of the groups. The results are summarized in Table 4-9 through Table 4-11 according to the order of difference. The color scale of the cells in the table represents the relative performance of the corresponding order of difference for the axis residing in that row. If a cell is dark, it means that its compression ratio is worse than the ones lighter than the cell.

When the first test case (Table 4-9) is considered, it is seen that the best performances are achieved for the second order of difference (except for the 6^{th} joint axis) as in the other compression schemes. For each of the joint axis, its performance is better than LZW and comparable with the other methods. Although the worst compression ratios are obtained for the first order of difference in four of the joint axes, the compression performance tends to decrease as the order of difference increases in general.

Considering the second test case (Table 4-10), the effect of the order of difference is the same as it is in the other compression methods. It is only better than LZW for the y-axis and better than ΔY for the z-axis. It is difficult to mention about the trend of the compression performance for this test case. It behaves differently for each axis.

For the last test case (Table 4-11), the effect of order of difference is again the same comparing with the previously discussed compression algorithms. The compression ratio remains constant for the PLC part of the original data, since there are few different values and taking higher order of difference makes no improvement.

Table 4-9 Compression Ratios (%) of ISMMC for Case 1

Joint Axis	1	2	3	4	5	6
6	111.2	36.5	35.3	51.2	71.9	96.6
5	97.6	28.2	33.1	49.4	69.8	94.3
4	114.2	37.2	34.9	50.9	73.4	96.5
3	81.2	24.0	29.5	45.0	64.7	85.2
2	79.9	23.5	28.9	44.5	65.4	86.7
1	90.2	24.6	29.2	44.9	65.3	88.0

Order of Difference

Table 4-10 Compression Ratios (%) of ISMMC for Case 2

Axis	1	2	3	4	5	6
Z	5.3	5.1	5.5	6.1	6.6	7.4
Y	112.6	111.0	77.1	30.9	35.3	54.3
X	111.8	110.2	75.8	29.4	34.6	55.3

Order of Difference

Table 4-11 Compression Ratios (%) of ISMMC for Case 3

Axis	1	2	3	4	5	6
PLC	3.1	3.1	3.1	3.1	3.1	3.1
Z	14.5	6.3	7.7	9.2	10.6	12.0
Y	5.0	4.6	4.7	4.8	4.8	4.9
X	14.9	6.4	7.9	9.4	10.9	12.3

Order of Difference

The performance of this method can be further increased via increasing the number of elements in the group. In the previous analysis, there were two elements in the groups corresponding to a first order Markov chain. The same three test cases are further analyzed for different orders of Markov chains and presented in Table 4-12 through Table 4-14. In these analyses, the order of differences are selected according to the best performances of the axes and kept constant for the same row.

When the first case is considered, it can be inferred from the table that the effect of the order of Markov chains is not linear. It tends to increase the compression ratio for the lower orders, but after 4th or 5th orders it decreases the compression ratio of the proposed method. The best compression performances are obtained for the first order model in general.

The trend in the second test case is similar to the first case, since they are corresponding to the same motion trajectory and their sampling frequencies are the same. The main difference is that the best ratios are achieved for the 8^{th} order of Markov chains and it tends to decrease as the order is increased.

The performance behavior of the method on the third case is totally different from the other two test cases. As can be seen from Table 4-14 that the compression ratios continuously tend to decrease as the order of Markov chains are increased regardless of the axis. This is due to the high sampling frequency (1 kHz) of the test case. The first two test cases were sampled at 20 Hz, which is very low when compared to 1 kHz. The performance of the method for this case is not better than the LZW method even for the 8th order of Markov chain, but with further increase in the order one may get similar results to LZW compression scheme.

Table 4-12 Compression Ratios (%) of ISMMC under Different Orders of Markov Chain for Case 3

Joint Axis	1	2	3	4	5	6	7	8
6	35.3	37.9	42.9	46.5	44.6	41.6	37.6	33.7
5	28.2	34.4	39.4	43.0	42.2	39.6	36.6	32.3
4	37.2	41.4	44.8	48.8	44.1	41.6	36.5	32.5
3	24.0	29.1	35.9	40.3	40.7	39.0	35.4	33.1
2	23.5	29.1	36.1	41.1	40.9	39.5	36.8	32.5
1	24.6	30.7	39.1	42.9	43.2	39.5	36.4	32.3

Order of Markov Chain

Table 4-13 Compression Ratios (%) of ISMMC under Different Orders of Markov Chain for Case 2

Axis	1	2	3	4	5	6	7	8
Z	5.1	3.9	3.6	3.6	3.0	3.3	3.5	2.8
Y	30.9	39.5	43.6	42.2	38.8	34.1	29.3	26.6
X	29.4	39.2	44.2	42.5	37.0	32.8	29.3	26.4

Order of Markov Chain

Table 4-14 Compression Ratios (%) of ISMMC under Different Orders of Markov Chain for Case 3

Axis	1	2	3	4	5	6	7	8
PLC	3.1	2.1	1.6	1.3	1.0	0.9	0.8	0.7
Z	6.3	5.2	4.6	4.3	4.0	3.8	3.6	3.5
Y	4.6	3.1	2.4	1.9	1.6	1.4	1.3	1.2
X	6.4	5.3	4.7	4.4	4.1	3.9	3.7	3.6

Order of Markov Chain

4.6 Conclusion

In this chapter, a direct command generation method based on differencing and data compression is introduced. Key points and contributions of the chapter can be summarized as follows:

- The lossless ΔY10 compression algorithm is specifically tailored for digital motion control applications where the command trajectories have distinct features: **i)** Since digital motion sensors (e.g. optical position encoders) are commonly utilized in CNC technology; the command sequences, which must be compatible with digital sensor readings, can be represented as integer series; **ii)** In most CNC machine tool applications, a constant speed along the traversed trajectory should be maintained to obtain the desired surface finish; **iii)** Certain restrictions on the acceleration (and jerk) profiles are imposed to avoid not only the saturation of servo-motor drivers but also the structural excitation of machine; **iv)** The tool trajectory is generally symmetrical; **v)** Due to nature of machining operations, repeated patterns in the sequence frequently emerge. Not surprisingly, this task-specific paradigm, which fully takes advantage of these special attributes, usually yields satisfactory compression performance on short command sequences if compared to the other (general purpose) techniques (HC, AC, LZW) frequently used in the technical literature. For long (stationary) sequences, the compression performance is comparable to the AC technique which is optimal in the sense that the rate achieved is close to the source entropy.
- The performance evaluation conducted in this chapter has revealed that a command trajectory for a CNC machine tool application may in some cases be compacted with a compression ratio as small as 4.5%. In that case, if the uncompressed data requires a gigabyte, the size of a compressed command trajectory would be less than a hundred MB. Hence, such command sequences can be easily stored in the (state-of-the-art) memory devices embedded inside the controller units. Furthermore, the

files could be easily handled and transferred to the devices within short time intervals using standard serial data-communication protocols like RS-485 where a maximum data transmission rate of 10 Mb/s is attainable.

- The compression performance of the presented method could be improved provided that better statistical models for the information source are utilized.
- Since the proposed CG technique extracts the original command sequences using a number of cascaded accumulators, lower-order differences (i.e. velocity-, acceleration-, jerk estimates), which are commonly utilized in the feedforward controllers of the advanced motion controller topologies (see [90]), are computed as by-products in this scheme.
- Since on most production machinery, the machine operator may modify the tool speed along the traversed trajectory (a.k.a. feedrate) on the fly; the proposed method incorporates a linear interpolator to generate the command signals at variable feedrates employing the original data.
- The introduced method lends itself to robust/reliable hardware implementation (a.k.a. system-on-a-chip). For instance, [28] and [29] realize a command generator (which makes good use of an earlier version of the ΔY algorithm) using an Altera Cyclone II (EP2C20F484C7) FPGA. Hence, the CG as a module can be incorporated in cost-effective and high-performance motion control systems developed for advanced electrical machinery, printing/textile equipment, robotic manipulators (for arc welding, painting, assembly, material handling) and manufacturing machines such as abrasive water jet cutters, laser beam machining centers, plasma cutters, rapid prototyping machines, wire EDM, etc.
- For CNC machining applications like turning and milling, the conditions changes may occur during the operation (e.g. deployment of different set of tools, tool geometry changes due to wear, etc.). In that case, the original ("pre-generated") data may no longer be valid and thus the new trajectory data, which essentially comprise the offsets of the initial tool path, must be regenerated off-line using a CAM software package. For the time-being,

this issue appears to be a limiting factor of the proposed method. On the other hand, efficient offset generation algorithms (such as the one presented by [97]) along with a metadata for tool geometry can be incorporated in the proposed paradigm as a natural extension. However, this prospect is to be explored in future studies.

CHAPTER 5

ADVANCED COMMAND GENERATION VIA CONTEXTUAL MODELING

An advanced command generation paradigm depending on contextual modeling is proposed in this chapter of the thesis. The paradigm incorporating vector operations of the given base curves is capable of generating different machining trajectories by simply modifying the inputs of the written program. In this paradigm, the machining trajectories are first defined with the proposed commands of the paradigm. After the compilation of the program, the generated machine code is transferred onto the hardware of the paradigm installed on the corresponding CNC machinery. Then the trajectories are generated by processing the program on the hardware. The developed motion command generation method is employed on different test cases and their results are presented in this chapter.

5.1 Introduction

CAD software platforms are continuously improving and evolving. With the new updates, various features are being included into the software. Thus, designing complex parts are becoming easier each day. On the other hand, it becomes difficult or sometimes impossible to write NC programs for the production of such complex parts manually. In these situations, CAM software is employed to generate conventional NC programs automatically according to the provided machine and tool specifications. These generated NC programs are usually very long and difficult to follow and modify manually. When the workpieces produced via CNC machinery are evaluated, it can be observed that most of them have symmetries (reflectional, rotational and translational), curve offsets, biased offsets, repetitive structures, etc. in general. Machining trajectories may be programmed utilizing these properties of the workpieces such that the machining program is now much smaller in size than the one formed with the conventional approach. In the recent study of Yaman and Dolen [66], they compressed the raw motion trajectories via their compression algorithm and achieved high compression ratios although they have not considered the above-mentioned properties of the workpieces directly. In order to further improve the compression performance and propose an alternative paradigm, a contextual modeling based command generation paradigm is proposed in this chapter of the thesis. The proposed paradigm makes use of the physical properties of the parts to be manufactured. The trajectories are defined by vectors and these vectors are later processed on the hardware of the machinery. Due to the utilization of the vectors, the paradigm is named as Vector Processor (VEPRO). As in the conventional approach, the motion trajectory is defined manually by writing the VEPRO program using the provided commands of the VEPRO. After it is compiled on the host PC, the output of the compilation (a.k.a. machine code) is transferred to the VEPRO hardware through different kinds of communication for processing. Then the hardware generates the motion commands with the help of its auxiliary units on the CNC machinery. With this proposed approach, the required time for

developing manual manufacturing programs is now reduced and the user is able to write the program on his/her own resulting in an increase in the quality of the program.

Cutter offset compensation can also be handled in the VEPRO paradigm without regenerating the tool trajectories. The user can modify the offset tables according to the wears on the tools. There is no need to deal with the special cutter offset compensation commands (such as G41 and G42 in NC programming). Another main advantage of the paradigm is that one can produce different workpieces with the same VEPRO program by just simply changing the base curves, offset table, etc. stored in the memory of the VEPRO hardware. These advantages of the VEPRO are illustrated in the following subsections of the chapter.

The proposed paradigm can be compared with the conventional approach in terms of Kolmogorov complexities [100]. The comparison should be made from designing the machining trajectory stage to the generation of the tool paths (Inputs and outputs of the paradigms are the same). Both paradigms can be divided into two main parts. In the first part of these methods, motion trajectories are defined according to the requirements of the CNC hardware. The complexities of the first parts can be considered as the same since the tasks done in this stage are alike. On the other, complexities do differ for the second part in which the hardware is generating the motion trajectories for CNC controllers. In the conventional approach, there exists an industrial computer responsible for the generation of motion commands utilizing the NC program installed. Considering the proposed paradigm, an average FPGA chip (Altera Cyclone V) is enough to embed the VEPRO circuitry having multiple kernels [98]. Thus, it can be concluded that Kolmogorov complexity of the conventional approach is higher than VEPRO.

The remainder of the chapter is formed as follows. The next section, the ancestor of the proposed paradigm is summarized and in the third section the novel command generation method is explained. Then in the fourth section, the commands necessary to describe the tool trajectories for the VEPRO are provided.

After discussing the performance of the VEPRO in the upcoming three sections through different test cases, the chapter is concluded with the last section.

5.2 Primitive Approach

In this section of the chapter, the basics of the primitive version of the advanced command generation paradigm proposed in this chapter are presented and it is evaluated by employing it onto a test case. The method itself is not finalized yet, but when the primitive evaluations are considered, it can be stated that the command sequences can be compressed to about one-thousandth of their original sizes by this approach.

5.2.1 The Aim of the Method and the Test Case

It is easier to explain the algorithm with a test case. A drawing of a Rabbit (Figure 5-1) is chosen as the test case. It is used in the study of Zhiwei et al. [85] and has 11 base sets, each of them are illustrated with a different color in the figure. The red one is the outer set of the rabbit. The remaining ones are called the “islands” in cases where they should not to be machined.

The original of the case was in millimeters. It is further processed to represent it with encoder counts. In this case, 1 μm corresponds to 1 count. This form of the test case and its offsets are shown in Figure 5-2. These offsets can be utilized to produce male of female versions of the Rabbit via 2.5D pocketing operations. The offsets are 1000 counts (1 mm) apart from each other. It is also shifted in x-y plane to make sure that the all trajectory has positive components.

The proposed command generation paradigm only needs the base sets of the case, which are required to generate offsets having different offset radii. Since the main goal of this approach is to represent the raw discrete trajectories with minimum required memory, the base sets are compressed with ΔY10 compression

algorithm, which is suitable for discrete trajectories composed of encoder pulses. After the base sets are compressed in the encoding stage of the command generation paradigm, the necessary part for the curve offset generation is formed by using a simple symbolic language. Thus, there is no need to compress the offsets of the base sets. The latest curve offset generation algorithm discussed in Chapter 3 can be utilized to generate them from the base sets. With this overall approach, the original sequence can be compressed to about one-thousandth of it. In the next subsection, the details of this paradigm are elaborated and a primitive memory structure is proposed.

Figure 5-1 Rabbit Composed of 11 Base Sets

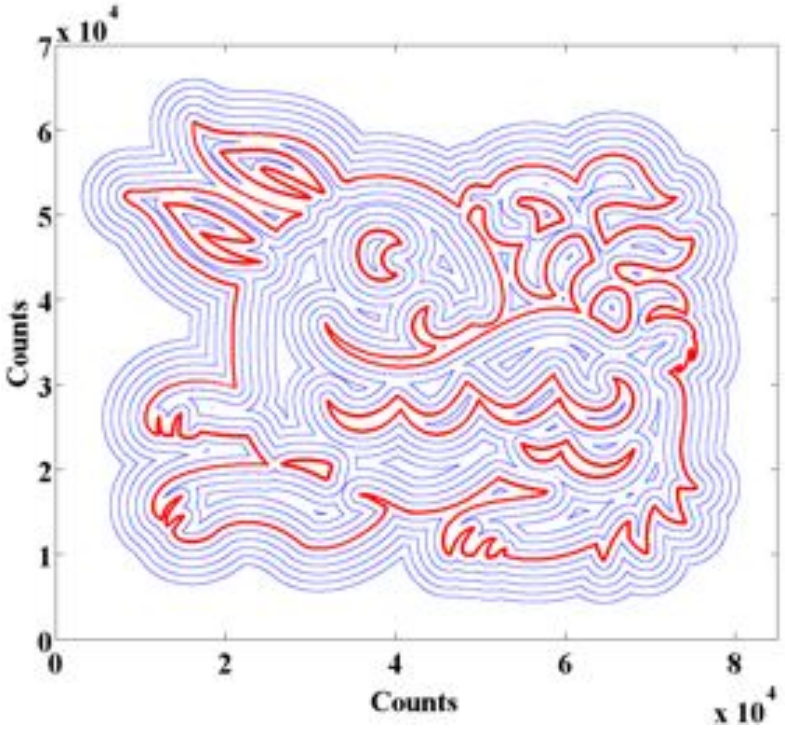


Figure 5-2 Rabbit with Offsets

5.2.2 Implementation Details

In order to use the memory efficiently and realize the command generation system easily, the compressed code is structured as shown in Figure 5-3 for a generic command sequence. The first four words and the next coupled words for each base set can be regarded as the header for this memory block. The initial 4 bits of the first word indicate the order of finite difference (where a maximum of 15th order for the differences can be represented) for the segments of the trajectory where any curve offsets are not to be generated. The rest of the first word (28 bits) are reserved for expressing the length of the command sequence. The second word of the header is used to specify the number of words reserved for the magnitude field, which indirectly determines the starting address of the sign field. The fourth word of the header gives the starting address of the zero field. In the remaining part of the header, there are coupled words for the description of the

curve offset generation. The first word in these couples gives the starting address of the corresponding compressed base set. The first two bits of the second word in the coupled set describe the type of the offsets. These bits are interpreted as follows:

- 00: Outer Offsets,
- 01: Inner and Outer Offsets,
- 11: Inner Offsets.

The next six bits are reserved for the number of offsets to be generated for the given offset radius. With the six bits, $2^6 = 64$ offsets can be generated in the maximum case. The last twenty-four bits ($2^{24} = 16777216$) in the second word of the couple represents the radius of the offsets in counts. After the header part, the first memory block corresponds to the compressed sequence for the segments where any curve offsets are not to be generated. Then comes the memory blocks for the compressed base sets. The structure of these memory sets is given in Figure 6-4.

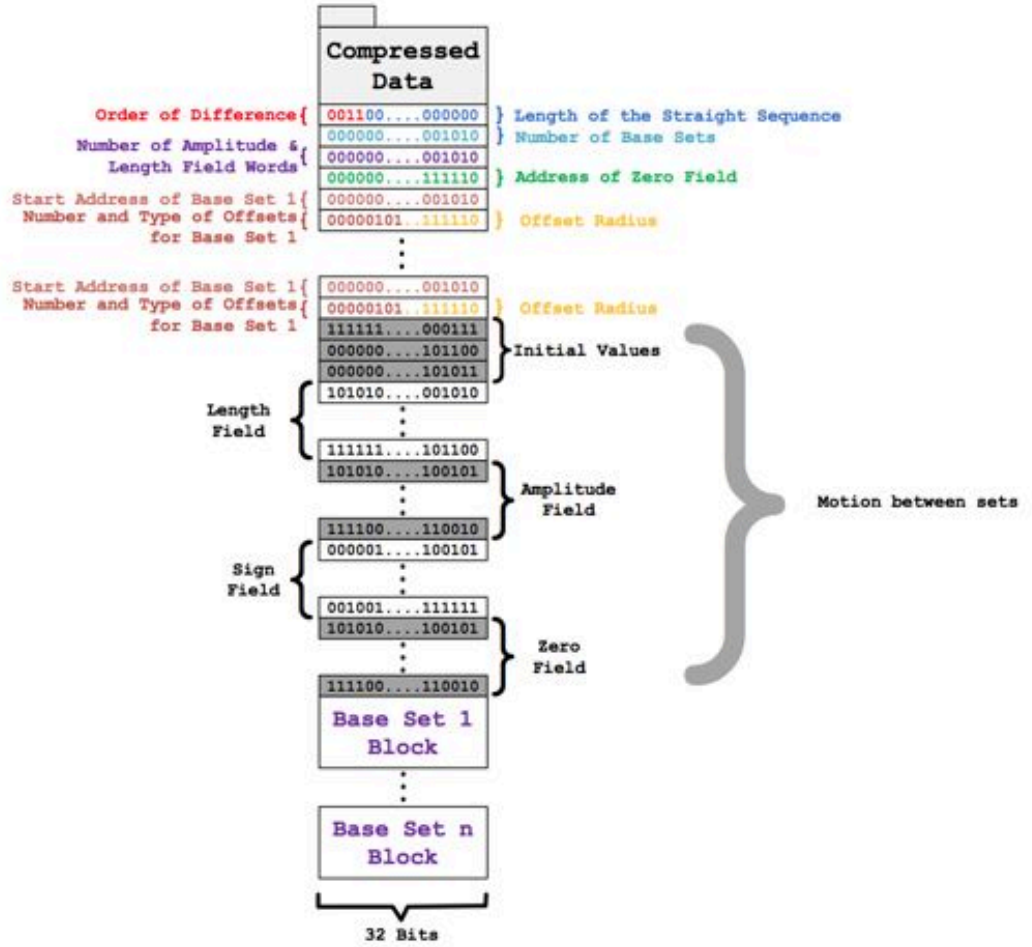


Figure 5-3 Memory Structure of the Primitive Method

During the decompression of the motion commands, the header of the overall memory is processed first. Then the commands are decompressed starting from the block for the non-offsetted trajectory. The ΔY10 compression algorithm is modified such that when there is a zero sequence of length two, there exists an offset sequence and decompression should continue by processing the coupled words for the base set and generate the defined curve offsets. When the generation of the corresponding offsets is finished, the decompression moves back to the non-offsetted part. This loop resumes until the non-offsetted part is finished.

5.2.3 Evaluation of the Primitive Method

The proposed command generation paradigm is elaborated by employing it onto the described test case in the previous subsections. As stated in these sections, the test case has 11 different base sets. In order to see the effect of curve offset generation algorithm on the memory size, the sets shown in Figure 5-1 are compressed with ΔY10 algorithm and the results are shown in Table 5-1. Each element in x and y axes can be represented by 3 and 2 bytes, respectively, when the range of the points are considered. Since there are 14111 points in the Rabbit case, the size of the x and y axes are 42333 and 28222, respectively. In the second part of the evaluation, the generated offsets are compressed with the same algorithm. The results are summarized in Table 5-2. The sizes of the original sequences are also given in the table. As can be inferred from the table that by just using the base sets of the case, one can achieve 0.3 % compression ratio that is too low when compared to the ratio of the case with compressed offsets.

Table 5-1 Bytes Required to Represent the Sequences After Compressing Them with ΔY10

Part	X	Y
0	2296	2404
1	319	312
2	301	304
3	292	289
4	300	332
5	300	332
6	295	300
7	417	550
8	294	291
9	292	288
10	563	568
Sum	5669	5970
Original	42333	28222

Table 5-2 Comparison of Sizes of Sequences Under Different Approaches

	X		Y	
	Size	**Ratio**	**Size**	**Ratio**
Compressed Base Sets	5669	0.0031	5970	0.0049
Compressed Offsets	345052	0.1884	317135	0.22597
Original	1831398		1220932	

In this section of the chapter, the idea of combining curve offset generation algorithms with data compression methods is proposed as a new command generation paradigm. After describing the test case suitable for the proposed scheme, the implementation details on the memory are discussed. The implementation of the paradigm is further improved and tested in the following sections.

5.3 Proposed Approach (VEPRO)

VEPRO is a motion command generator designed specifically for CNC machinery. It is established after trying to implement the command generation method proposed in the previous section. VEPRO can be regarded as the advanced version of that method.

As illustrated in Figure 5-4, the system-on-a-chip (SOC) solution employs a multi-kernel processor architecture that is equipped with specialized peripheral units. In this architecture, the memory management unit (MMU) plays a key role as it manages all the system registers (namely, C, R, S) shared by all kernels. The machine code (compilation of the user written program) and the related data are transferred to the memory chips available on the hardware over a serial communication protocol.

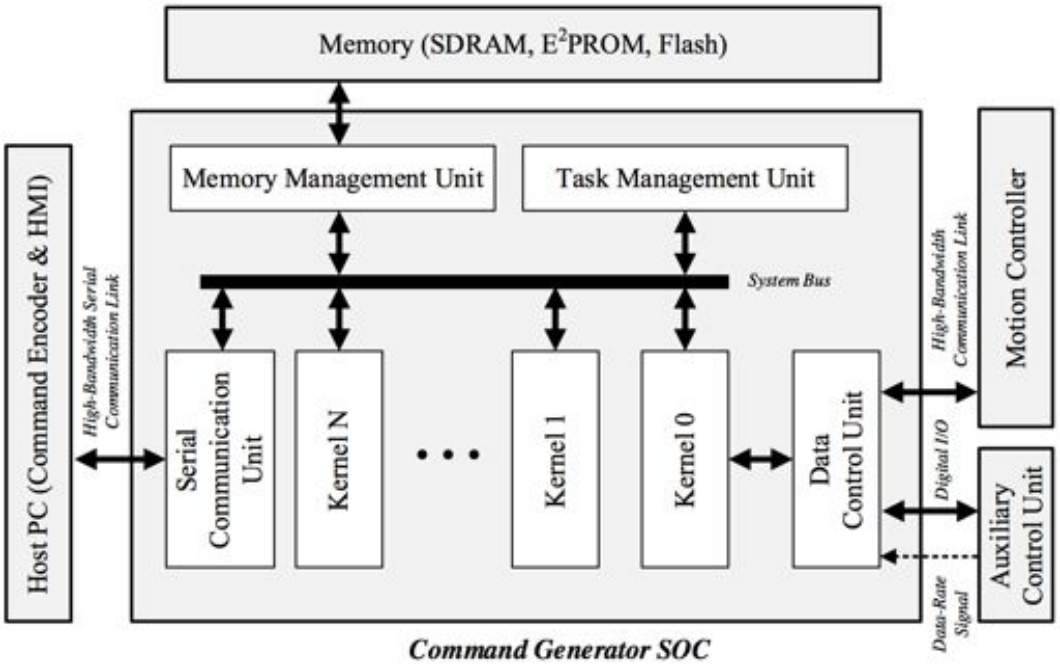


Figure 5-4 SOC Solution of VEPRO

On the other hand, each kernel does have its own set of local dynamic matrix registers C(0) & C(1) which are exclusively used as temporary data storage. Similarly, all kernels are to invoke functions (subroutines) defined globally. The timing events along with the synchronization among the kernel are performed by the task management unit (TMU). To avoid collision, only main kernel (Kernel 0) is allowed to interact with data-rate control unit (DCU), which outputs the motion command data at the specified rate set by the external logic. The PLC functions of the CNC machinery are handled by the communication between DCU and the auxiliary control unit.

The main advantage of this SOC is that it is very simple and can be realized by a low-cost FPGA having limited resources. The current hardware processing NC programs is comprised of computers having at least 1 GB of RAM. With the proposed command generation paradigm, the procedure is simplified and its hardware complexity is decreased.

5.4 VEPRO Commands

The language of the proposed motion command generation paradigm is categorized into eighth sub categories and for each of them the relevant commands and their explanations are provided in upcoming subsections.

5.4.1 Register Sets

There are four types of registers used in the VEPRO language. The first one is the dynamic registers represented with the letter C. They are mainly utilized to store base curves and the results of vector operations. The second type is the ones used for general purposes. This is represented with the letter R. The next one is the simplest register kind of all. It is stored in the last element of R and used for flags. The last register is reserved for PLC functions of the CNC machinery. The details of these registers are discussed below.

Register (C). It is named as dynamic matrix registers. There are 1024 dynamic elements in the register. The first element of this register is reserved for the intermediate results of vector operations such as the output of the curve offset generation algorithm. The rest of the elements can be utilized to store the base curves of the trajectory.

Register (R). It is named as general purpose registers. There are also 1024 elements in the register. The first element of the register is reserved for the intermediate arithmetic results. The rest are used for various purposes such as loop numbers, current and previous axis positions, offset values, etc.

Register (S). It is a read-only register and called as status register. The last element of the general purpose registers (R) is reserved for this register. The current length of the register is 8 bits, but it can be increased according to the needs new vector operations. The defined status registers are as follows.

- **EF:** Error

- **LT:** Less than (N)
- **ZF:** Zero (Z)
- **GT:** Greater than
- **OF:** Overflow (V)
- **UF:** Underflow (U)
- **LI:** Linear interpolation mode
- **OE:** Output enable

Register (P). It is named as PLC register. PLC functions of the CNC machinery are controlled via this register. It is designed to be 32-bit currently. The least significant 24 bits are used for setting the spindle speed. When it is needed to stop the spindle, the speed should simply be set to zero. The next two bits are utilized for the direction of the rotation and for enabling coolant. The following six bits are reserved for the definition of the tools to be used in the machining. If there are more PLC functions in the machinery, the word size of this register should be increased. The usage of this register is illustrated below.

Spindle Speed (24 bits):	20000 rpm: $000000000100111000100000_2$
Spindle Rotation (1 bit):	1 (CW) / 0 (CCW)
Coolant On/Off (1 bit):	1 (On) / 0 (Off)
Tool (6 bits):	Tool #5: 000101_2

These register adjustments are transferred to the VEPRO via PLC command. For the illustrated example one should use PLC (385895968).

5.4.2 Declarations

Declarations in the language are mainly used to label some command lines in the program and assign some parameters of the written program for the specified command trajectory. There are ten different types of declarations presently. These declarations are briefly explained below.

Declaration (label **EQU** address**).** It is used to assign labels to the specified memory locations.

Declaration (label **BGN** #**).** It is used to assign label to the beginning of the thread #.

Declaration (END #**).** It is used to define the end of the thread #.

Declaration (label **DEF R**(i)**).** It is used to assign a label to the specified element (i) of the general purpose register (**R**).

Declaration (DIM #**).** It is used to set the number of axis to #.

Declaration (DIO #**).** It is used to set the number of digital Input/Output channels to #.

Declaration (FPL #**).** It is used to set the location of the fixed-point numbers to #. It should be between 0 and 8×(Word Length − 1).

Declaration (NCR #**).** It is used to set the number of curve registers to #. Depending on the application, one may need more than one register for the base sets or for the results of vector operations.

Declaration (SIL #**).** It is used to set the maximum step size to # in linear interpolation (in counts).

Declaration (WOR #**).** It is used to set the word length to #.

5.4.3 Parallel Processor Commands

As discussed in the previous section, there are parallel processors embedded into the VEPRO. For the proper operation of these processors, additional commands are necessary. Two of the commands are used to enable and disable the regarding kernels and the other two are utilized for the synchronization of the auxiliary kernels with the master kernel. These commands are described below.

Start Processors STA (#): This command is used to enable the execution of the parallel processors. Bits of the specified # indicates the id number of the processors to be enabled. For instance, STA 15 (= 0001111_2) denotes that the kernels 1 to 4 will be started.

Terminate Processors TER (#): It is used to terminate the execution of the threads whose id numbers are specified by #.

Wait Processors WAI (#): The master kernel awaits the timing event for the multiple threads whose id numbers are specified by #.

Trigger Processors TIC: The auxiliary kernels send acknowledgement signals to the master kernel indicating that they have completed the assigned tasks to them by the master kernel. After the master kernel receives these trigger signals, it continues to its proper operations.

5.4.4 Compare, Test, and Branch Commands

These commands are mainly used to implement the loops in the algorithms. They are utilized to compare some values and take action according to the result of the comparisons. These commands are briefly explained below.

Compare CMP(R(j), R(i).#): The result of the difference between the two inputs of the command is written on the read-only status register (S).

Bit Assign BIT(R(i).#): The specified bit of R(i) is assigned to ZF bit of the status register (S). The bit number should be between 0 and $8\times$(Word Length $-$ 1).

Branch Equal BEQ(label): The algorithm branches to the given label if ZF bit of the status register (S) is equal to 1.

Branch Not Equal BNE(label): The algorithm branches to the given label if ZF bit of the status register (S) is equal to 0.

Branch Greater Than BGT(label): The algorithm branches to the given label if GT bit of the status register (S) is equal to 1.

Branch Greater Than or Equal BGE(label): The algorithm branches to the given label if ZF bit of the status register (S) is equal to 1 or GT bit is equal to 1.

Branch Less Than BLT(label): The algorithm branches to the given label if LT bit of the status register (S) is equal to 1.

Branch Less Than or Equal BLE(label): The algorithm branches to the given label if ZF bit of the status register (S) is equal to 1 or LT bit is equal to 1.

Branch Overflow BOF(label): The algorithm branches to the given label if OF bit of the status register (S) is equal to 1.

Branch Underflow BUF(label): The algorithm branches to the given label if UF bit of the status register (S) is equal to 1.

Branch Error BER(label): The algorithm branches to the given label if EF bit of the status register (S) is equal to 1.

Jump JMP(label): The algorithm jumps to the specified label with this command.

Jump Subroutine JSR(label): The algorithm jumps to the subroutine's starting address.

Return Subroutine RTS: The algorithm returns from the subroutine to the main program when this command is processed.

End Program END: The program terminates with this command.

5.4.5 Register and Memory Operations

There are only two commands for this category, but if there is a necessity of new commands it can be added. The first one is used for moving the data stored in some part of the memory to the given variable and the second one is utilized to empty the registers. These two commands are given below.

Move MOV(R(j), R(i)/@MEM/#): The operation is used to move the information residing in the register R(i) or in the specified memory location MEM or the specified bit of R(i) to the register R(j).

Clear CLC(R(i)): The given register to the command is assigned to zero.

5.4.6 Arithmetic and Logic Operations

There exist simple mathematical operations in this category. The commands are used only with the general purpose registers (R). The abbreviations and explanations of these commands are discussed below.

Addition ADD(R(j), R(i)/@MEM/#): In this simple operation, the value residing in the register R(i) or in the specified memory location MEM or the specified bit of R(i) is summed with R(j) and assigned to R(j).

Multiplication MUL(R(j), R(i)/@MEM/#): In this simple operation, the value residing in the register R(i) or in the specified memory location MEM or the specified bit of R(i) is multiplied with R(j) and assigned to R(j).

Sign SGN(R(i)): The only one that uses the flags is this operation. It checks the sign of the given input and then raises the LT flag if it is negative and raises ZF flag if the input is equal to zero.

Negate NEG(R(i)): In this operation, the given input is multiplied with -1 and the result is assigned to R(i).

Absolute ABS(R(i)): In this operation, the absolute value of the given input is assigned to R(i).

Decrease DEC(R(i)): The given input is decreased by one and assigned to the same register.

Decrease INC(R(i)): The given input is increased by one and assigned to the same register.

5.4.7 Flag Operations

Flag operations are the simplest and the most important commands of the VEPRO. Thus, they should be handled carefully. For these purpose, five different commands are proposed for the flag operations. If necessary, they may be increased in the future editions of the VEPRO. These operations are provided below.

Flag SLI. By this command, the linear interpolation flag (LI) is raised.

Flag CLI. The linear interpolation flag (LI) is lowered when this command is processes in the program.

Flag CEF. The command simply clears the error flag (EF = 0).

Flag SOE. The motion commands are generated if the output flag (OE) is enabled with this command.

Flag COE. The motion commands are suppressed if the output flag (OE) is disabled with this command.

5.4.8 Vector Operations

These vector operations presented below are the heart of the VEPRO command generation system. The first seven of the commands (MOV) are used to move the motion data residing in the given register to the other registers. CLR command

simply clears the given curve register and REV is used to reverse/flip the registers. The OUT command has two different versions as can be seen below. The first one simply interpolates the given data and assigns them to the temporary register, which is C(0). The other version of the OUT command is used when there is a need to perform interpolation in the third axis. For instance, if the curve offset generation operation needs to be done in three axis, this command is utilized to perform further operation to update the result of curve offset generation for three axis representation. The DDY command is used to decompress the motion data if ΔY compression algorithm is utilized in advance. In order to generate curve offsets of a given base curve, GCO command is used. According to the sign of the value, it can generate inner (negative) or outer (positive) offsets of the given curve segments. It is also capable of handling cases where there exists islands and self-intersecting lines. The upcoming commands GPC and GPB are responsible for the generation of equidistant points on a circle or on a base curve, respectively. HTM command transforms the base curve residing in the curve register C(3) and assigns the result to the same curve. The transformation is done according to the given angles and displacements. PLC command, as the name implies, modifies the PLC register of the VEPRO. When there is a need to raise a custom flag, EVE command is employed. It takes action before the commands are generated with OUT command. The last three of the commands are simple connection, addition and multiplication operations defined for curve segments.

Move MOV(C(j.n_j), @MEM.start.end): Data residing between the given start and end addresses of the memory is assigned to C(j.n_j) with this vector operation. Here n_i represents the n_i'th axis of the curve register.

Move MOV(C(j.n_j), C(i.n_i).start.end): Specified segment of C(i.n_i) is transferred to C(j.n_j). If start address is higher than the end address, the vector is reversed firstly and then moved to C(j.n_j). From now on the shaded portions of the commands refer to the optional parameters. When start and end addresses are not provided, all elements of the register will be moved.

Move MOV(R(j), C(i.n_i).index): One of the elements of C(i.n_i) numbered as index is assigned to the register R(j).

Move MOV(R(j), C(j.n_j).start.end): The specified portion of C(j.n_j) is assigned to R(j). In this command, the start and end addresses refers to the bit addresses. For example, **MOV** (R(0), C(0).145.153) means that R(0)<0:7> := C(0.1)[4]<17:24> (It is assumed that the word length is 4 bytes).

Move MOV(C(j.n_j).start.end, R(i)): This command is the reverse of the previous vector operation. Here, R(i) is stored to the specified region the curve register C(j.n_j). For example, **MOV** (C(0).145.153, R(0)) means that C(0.1)[4]<17:24> := R(0)<0:7>.

Move MOV(C(j.n_j).index, R(i)): R(i) is assigned to the specified index of C(j.n_j). If the axis of the register is not defined, it is taken as 1.

Move MOV(C(j.n_j), bit.address, 0/1): In this vector operation, 0 or 1 is assigned to the stated bit of the curve register C(j.n_j). For instance, **MOV** (C(0), 1781,1) means that C(0.1)[55]<21> := 1 (The word length is again assumed to be 4 bytes).

Clear CLC(C(j.n_j)): C(j.n_j) is emptied.

Reverse REV(C(i.n_i).start.end): C(i.n_i) is reversed/flipped and assigned to C(0.n_i). This operation can also be accomplished via **MOV** command by using a bigger number in the start address. This command is added to increase the readability of the VEPRO programs

Output OUT(C(i).start.end): Motion command data in C(i) is linearly interpolated and assigned to the curve register C(0).

Output OUT(C(i).start.end, C(j)): The third axis of C(i) is linearly interpolated according to the uncommon axis of C(j) and the results of the interpolation is written to the curve register C(0).

Decompress ΔY DDY(C(i.n_i), @MEM.start.end): The data at MEM[start.end] are decompressed using ΔY and assigned to C(i.n_i) by this vector operation.

Generate Curve Offsets GCO(C(i).start.end, offset): This command computes 2D offset of C(i)[start.end] and assigns it to C(0): if offset > 0, the outer offset is generated. If there are multiple offset curves, the overflow flag is set; C(1) holds the starting addresses associated with the offset curves stored in C(0).

Generate Points on a Circle GPC(C(j), radius, number, angle): The command generates a "number" of equidistant points on a circle (in XY plane) with specified "radius" (counts) beginning from a certain "angle" (degree) and assigns the result to C(j).

Generate Points on a Base Curve GPB(C(j), C(i), number, start): The command generates a "number" of equidistant points on a base curve stored in C(i) beginning from a specified starting position (in counts) and assigns the result to C(j).

Homogeneous Transformation Matrix HTM(C(j), C(i).n): In this vector operation, homogeneous transformation (HT) of C(j) is accomplished via the elements of C(i.n) and the result is stored in the curve register C(0). The required angles and displacements are provided in C(i).n. The transformation is only defined for 3D dynamic matrices whose any two vectors [i.e. C(j.1), C(j.2), C(j.3)] must be (at least) non-empty. Three different operations can be completed according to the types of the given inputs.

- If the dimension of C(j) is 2, 2D HT is performed where the undefined (or absent) axis assumed to be a zero vector. Here, C(i.n)[1] is the offset of the first axis; C(i.n)[2] is the offset of the proceeding axis; C(i.n)[3] is the angle (fixed-point representation in degrees) defined around the remaining axis.
- If the dimension of C(j) is 3 (then the length of C(i.n) must be 6 for this case), 3D HT is performed. Here, C(i.n)[1:3] refers the offsets for each axis while C(i.n)[4:6] denotes the corresponding Euler angles.

- If the dimension of C(j) is 3 but the length of C(i.n) is 3 then 2D HT is performed in XY plane. Here, C(i.n)[1:2] are the offsets of the X and Y axes; C(i.n)[3] is the angle defined around the Z-axis.

Programmable Logic Controller PLC(R(j)): The PLC register described in the first subsection is adjusted with this command. The modifications are processed just before the generation of motion commands.

Concatenate CON(C($j.n_j$), C($i.n_i$)): The command concatenates C($j.n_j$) and C($i.n_i$) and assigns the new curve register to C(0).

Addition ADD(C($i.n_i$), bias): The bias is added to C($i.n_i$) (signed integer operation) and the result is assigned to the curve register C(0. n_i).

Multiplication MUL(C($i.n_i$), scale): C($i.n_i$) is multiplied with the scale (fixed-point operation) and the result is assigned to curve register C(0. n_i).

5.4.9 Vector Queries

The last category of the VEPRO is the vector queries. There are three of them as explained below. These commands are utilized to gather some specific information (length, dimension, and axis configuration) of a curve segment and assign them to the register R(0).

Length Query QLC(C(i)): The length of the input curve register is assigned to first element of the general purpose register (R(0)) with this query command.

Dimension Query QDC(C(i)): In this query operation, the number of axis whose command trajectories are residing in the register is assigned to the intermediate result register (R(0)).

Axis Configuration Query QAC(C(i)): When the axis configuration of a curve register is required, this command is used. The result will again be written to R(0) in terms of 3 bits (XYZ). For instance, if the value in R(0) = 5 = 101_2, then we can say that X and Z axes are present in the curve register. There is no data available for Y-axis.

5.5 MATLAB Emulations of the VEPRO Hardware

In this section of the chapter, a generic VEPRO program is developed for the finishing operations of the plastic injection molds of

- Hemisphere
- Shampoo bottle
- Handset

Rendered SolidWorks parts of these cases are provided from Figure 5-5 to Figure 5-7. The radius of the hemisphere is $50\,mm$ and the number of base curves is determined to be 100 for the finishing operation (having $1\,mm$ offsets along the Y-axis). The shampoo bottle provided in Figure 5-6 is $100\,mm$ in height and the radius of the main part is $20\,mm$. As in the case of the hemisphere, 100 base curves are generated in VEPRO. The third test case is totally different from the first two ones. Its maximum dimensions are $86mm \times 240mm \times 36mm$ and there are 50 different base curves.

By only changing the offset table and the base curves, different machining operations are obtained with the same VEPRO program. For the cases of the Bottle and the Hemisphere, there is no need to redefine the base curves. Each segment of these two test cases can be generated from a semi-circle. On the other hand, all of the segments of the Handset are different from each other due to the complexity of the Handset. Thus, they should be stored in the appropriate memory field and be fed to the VEPRO program when necessary. The offset table of this case simply consists of zeros indicating that the original base curves are to be utilized. The VEPRO program capable of generating motion trajectories for the three cases is provided in Table 5-3. According to the written program, the base curves are assigned to the appropriate curve register from the specified memory locations. For the third test case, an appropriate memory index table should be embedded into the VEPRO program.

These test cases are emulated in MATLAB environment by writing the corresponding MATLAB functions of the utilized VEPRO commands. The results of these emulations are represented from Figure 5-8 to Figure 5-10. Top, isometric and front views of the test cases are presented in these figures. The lines in red color represent the motions where machining is done and the blue lines are for the fast movements of the machine tool. In each fast movement, the tool first goes up to the defined Z-axis absolute position and then moves to the final X and Y points via linear interpolation. Different Z-axis absolute positions are used in the test cases. As can be observed from the VEPRO program and the output figures that the tool moves to the next curve segment from the nearest side after the generation of one curve segment is completed. This task is accomplished in the VEPRO program by flipping the base curve via **MOV** command.

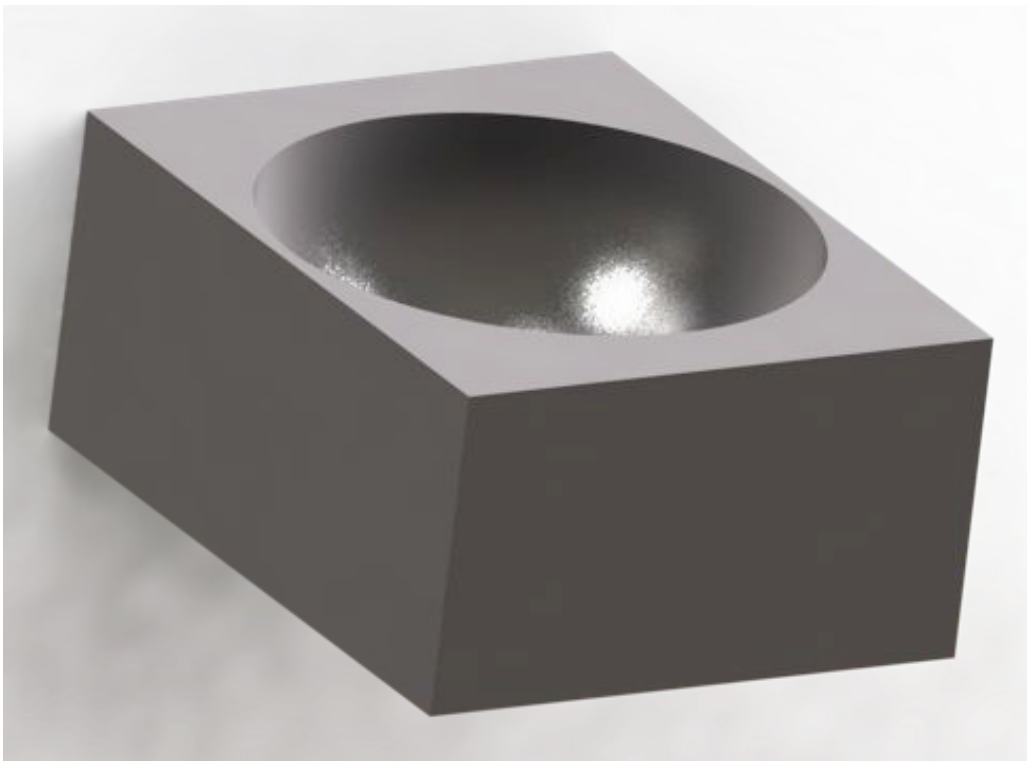


Figure 5-5 Rendered SolidWorks Part of the Test Case Sphere

Figure 5-6 Rendered SolidWorks Part of the Test Case Bottle

Figure 5-7 Rendered SolidWorks Part of the Test Case Handset

Table 5-3 Generic VEPRO Program for the Three Test Cases

```
; VEPRO program for the "Bottle","Sphere", and "Handset" cases
;
               DIM     3       ; 3-axis (X,Y,Z)
               WOR     4       ; use 4-byte word (32-bit)
               FPL     0       ; no fixed-point multiplication
               NCR     3       ; 3 dynamic arrays including C(0) are needed!
;
; Define addresses in memory
;
X0             EQU     0x0     ; initial x location (1 word)
Y0             EQU     0x4     ; initial y location (1 word)
Z0             EQU     0x8     ; initial z location (1 word)
DeltaY         EQU     0xC     ; increment in Y (1 word)
Rtab           EQU     0x10    ; starting address of offset table (50 words)
Bcur           EQU     0xD8    ; starting address of base curve (2*1000 words)
;
; Assign labels to registers (or constants)
;
X              DEF     R(1)    ; position register for axis-1: X
Y              DEF     R(2)    ; position register for axis-2: Y
Z              DEF     R(3)    ; position register for axis-3: Z
i              DEF     R(4)    ; R(4) is to be used for indexing
r              DEF     R(5)    ; R(5) stores the temporary offset value
Zup            DEF     R(6)    ; Absolute z position of the tool in fast movements
;
; Program starts here
;
               ORG     0x2018  ; starting address(following 2054*4 words of data)
               SLI             ; set linear interpolation mode
               SOE             ; enable output
               CLR             ; clear all curve registers
               MOV     Zup,5   ; Zup is assigned to be 5 mm.
               MOV     X,X0    ; load initial tool locations to pos. registers
               MOV     Y,Y0
               MOV     Z,Z0
               MOV     C(1.1),Rtab,0,199      ; load offset values to C(1) (1D)
               MOV     C(2.1),Bcur,0,3999     ; X-axis components of base curve
               MOV     C(2.3),Bcur,4000,7999  ; Z-axis components of base curve
               MOV     i,0                    ; initial index value ("for loop")
Loop:          MOV     r,C(1.1),i             ; r := C(1.1)[i]
               GCO     C(2),r                 ; generate 2D offsets of C(2) with r
               OUT     C(0)                   ; output C(0) holding the offset
               MOV     C(2),C(2),3999,0       ; make sure to flip C(2) for the next
iteration
               ADD     Y,DeltaY               ; Y coordinate increases by DeltaY
               INC     i                      ; increase loop index i
               CMP     i,100                  ; if i is not equal to 50,
               BNE     Loop                   ;  then  continue  with  the  loop  (100
iterations!)
               CLR     C(0)                   ; clear C(0)
               MOV     X,X0                   ;  load  the  initial  pos.  to  the
resisters
               MOV     Y,Y0
               MOV     Z,Z0
               OUT     C(0)                   ; perform linear int. to the initial
position

               END
```

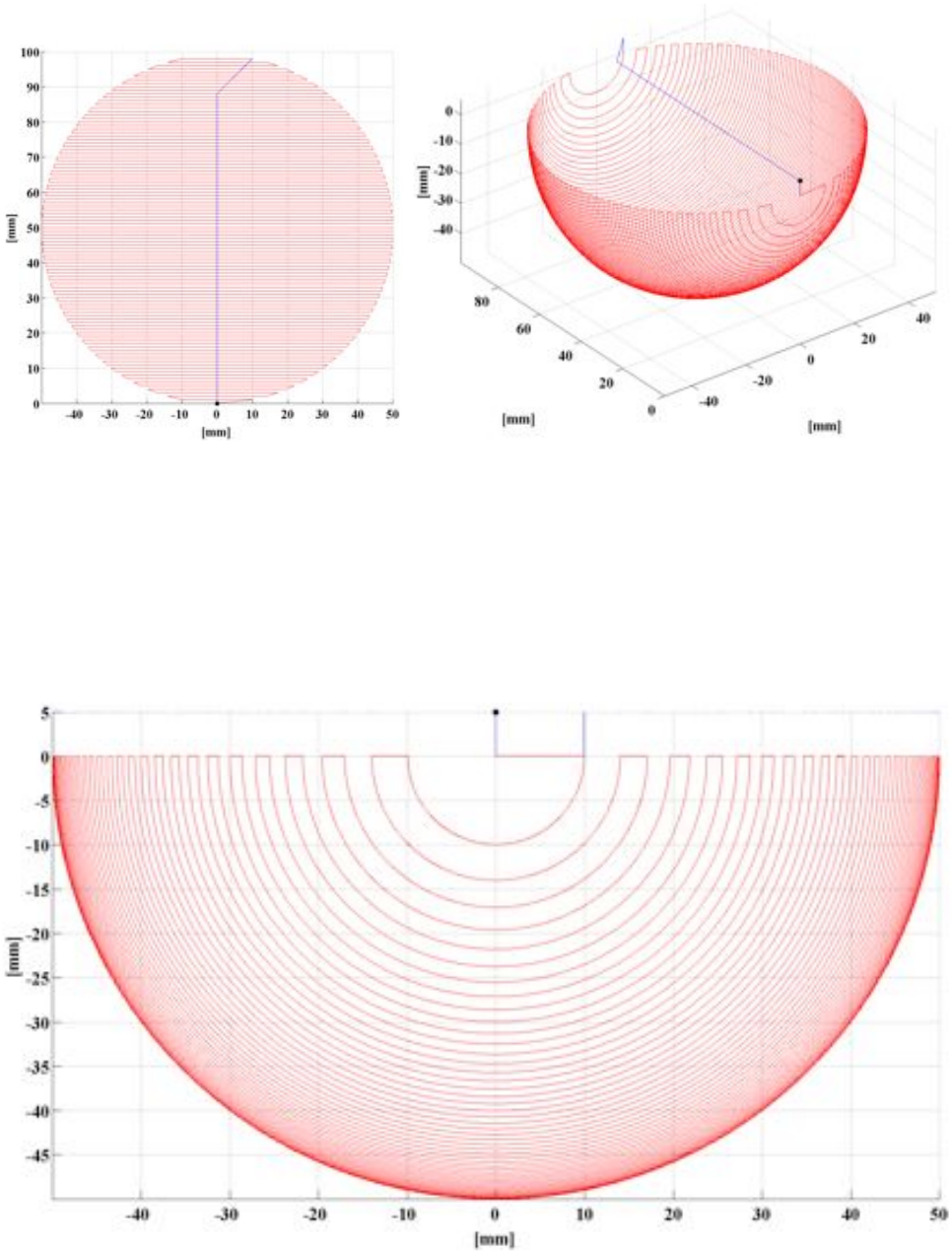


Figure 5-8 Plots of MATLAB Emulation of the Test Case Sphere

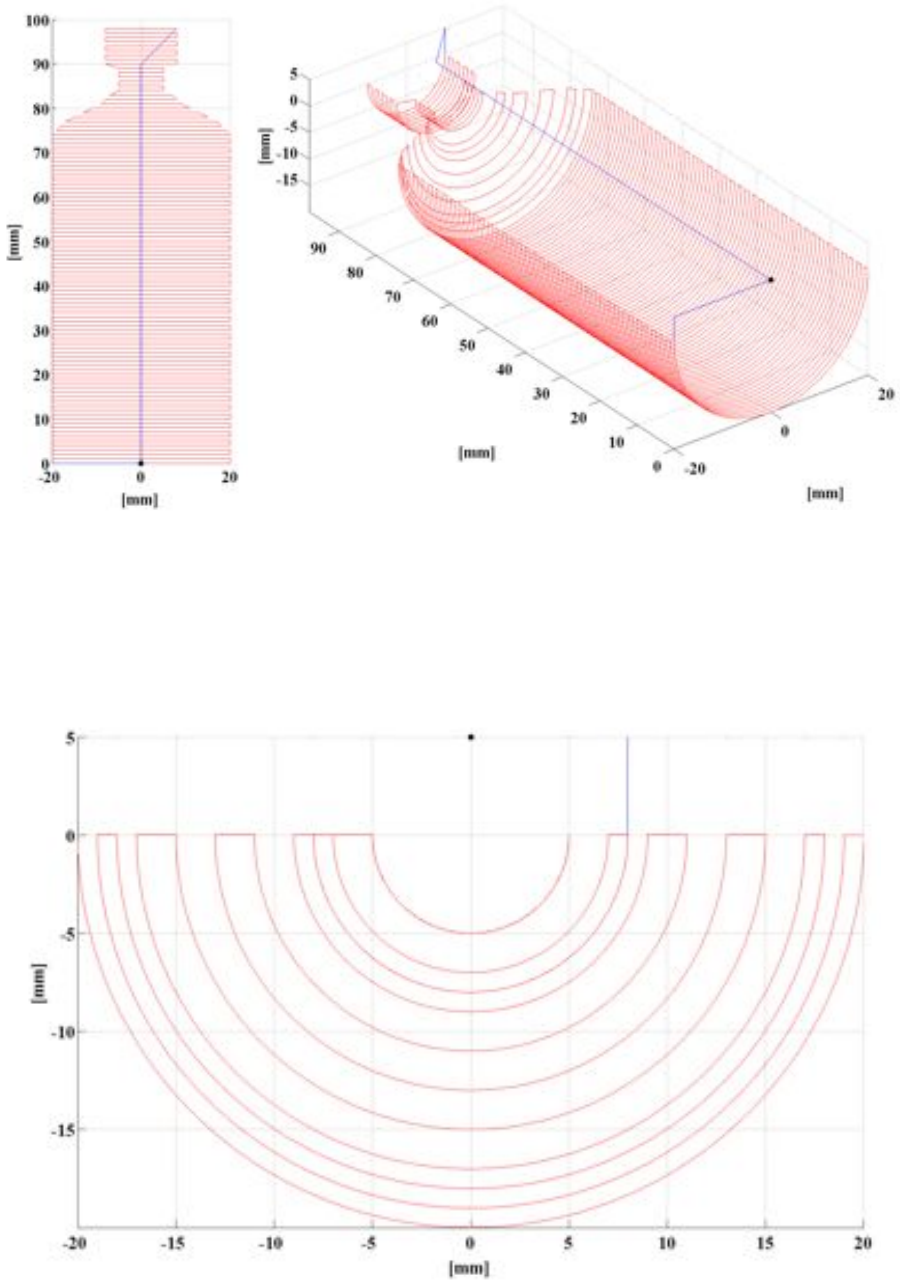


Figure 5-9 Plots of MATLAB Emulation of the Test Case Bottle

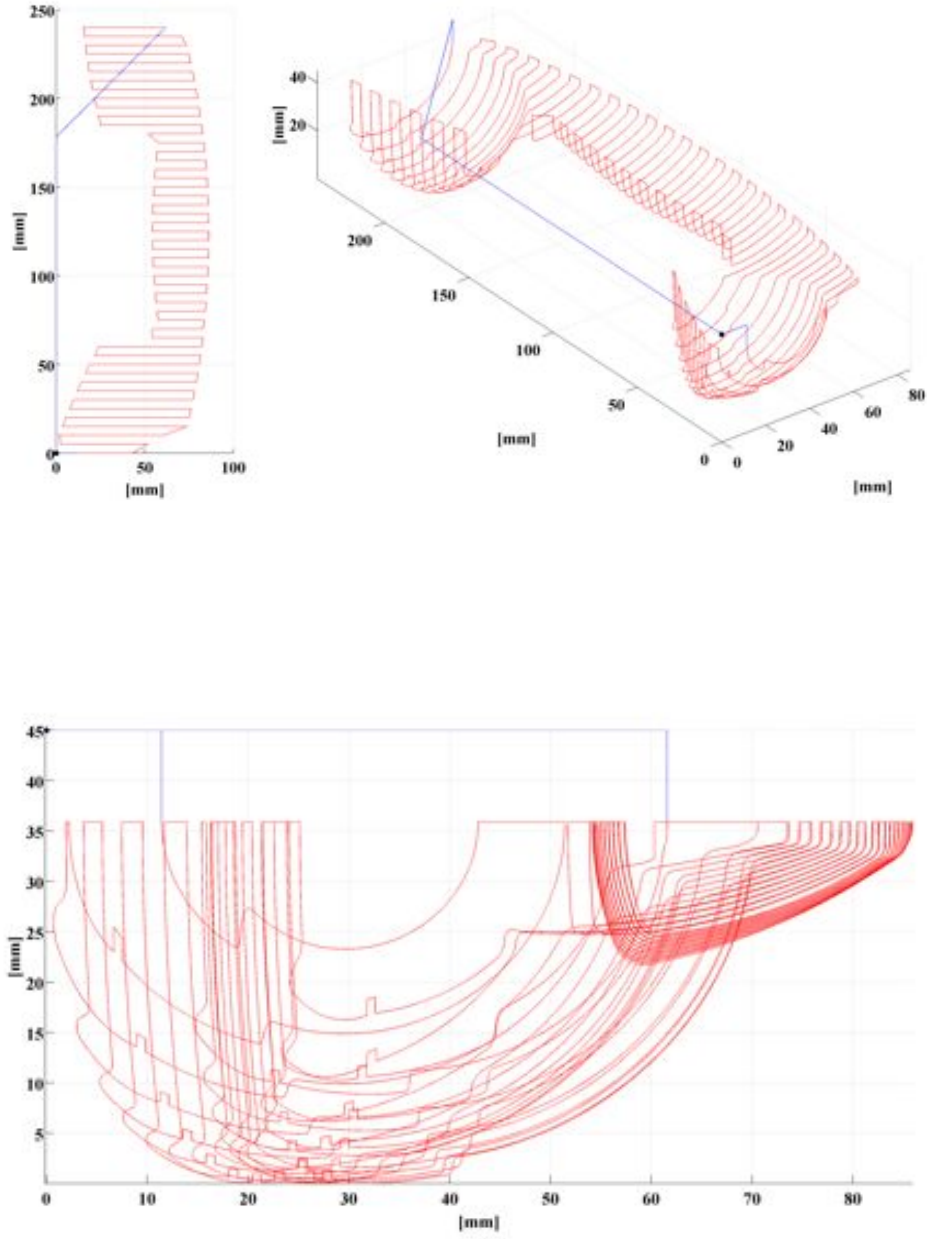


Figure 5-10 Plots of MATLAB Emulation of the Test Case Handset

In order present the capabilities of the VEPRO paradigm, simple modifications are employed on the third test case (Handset). The average x-axis values of each base curve of the handset are stored into the M_X array and the original base curves are shifted such that they are now centered along y-axis. If one needs to generate the original trajectories of the finishing operation, the corresponding element in the M_X array and the x-axis values of the modified base curves should be summed before the generating the commands. The modifications of the VEPRO program are provided in Table 5-6. The first command is used to get the M_X array from the memory and the second command is placed in the loop for the summation of the x-axis.

Table 5-4 Modifications on the Generic VEPRO Program of the Three Test Cases

```
MOV     MX,Bcur,8000,8049      ; MX is formed
....    ................
ADD     C(2.1),MX(i)           ; X coordinate of base curves updated
```

With the adjustments in the M_X array, different command trajectories can be obtained. When all of the elements in this array are zero, the trajectory in Figure 5-11 is obtained. In this configuration, the base curves are centered along the y-axis. If one wants to obtain a mirrored image of the original trajectory, the initially obtained M_X array should simply be multiplied by -1. The result of this configuration is presented in Figure 5-12. In another application, the elements of the M_X array are generated randomly (from 0 to 100) in MATLAB and the result is shown in Figure 5-13. In the last configuration of the M_X array approach, the initially obtained M_X array is multiplied by 0.5 and presented in Figure 5-14. The same approach may be employed on the other axes and different configurations can be obtained by simply modifying the initially given VEPRO program.

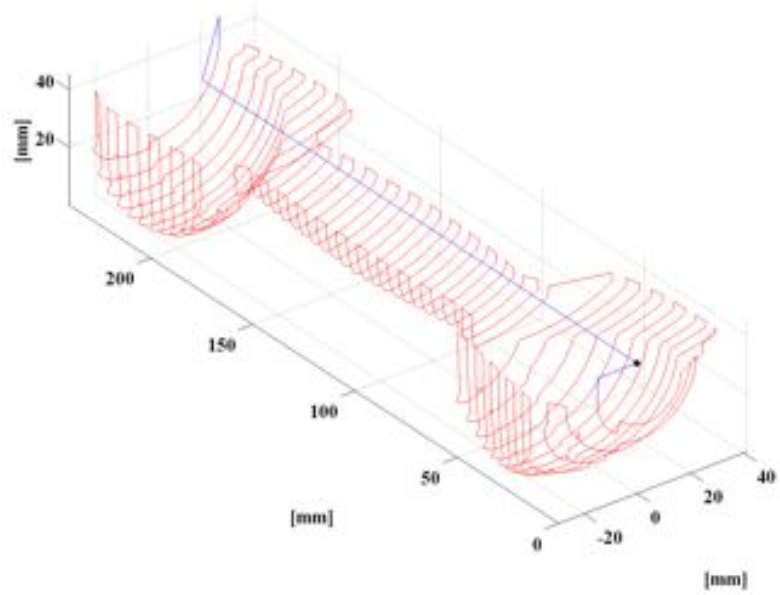


Figure 5-11 Centered Base Curves of the Handset

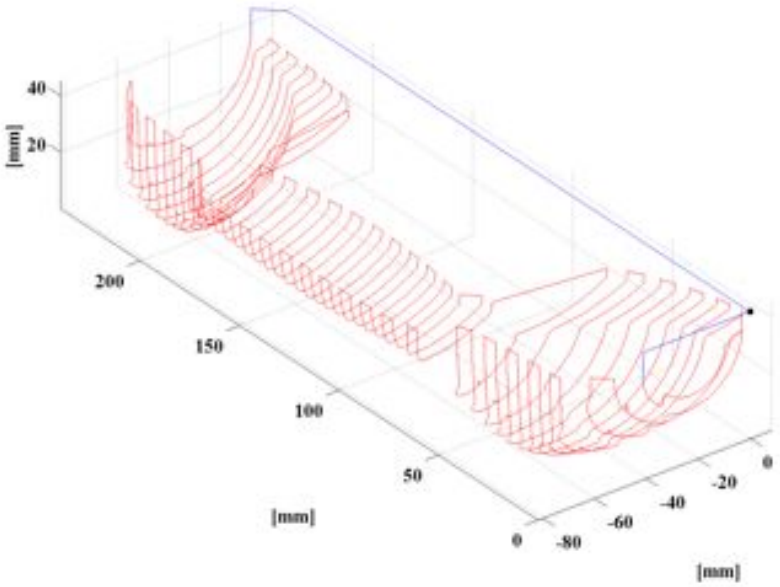


Figure 5-12 Mirrored Base Curves of the Handset along Y-axis

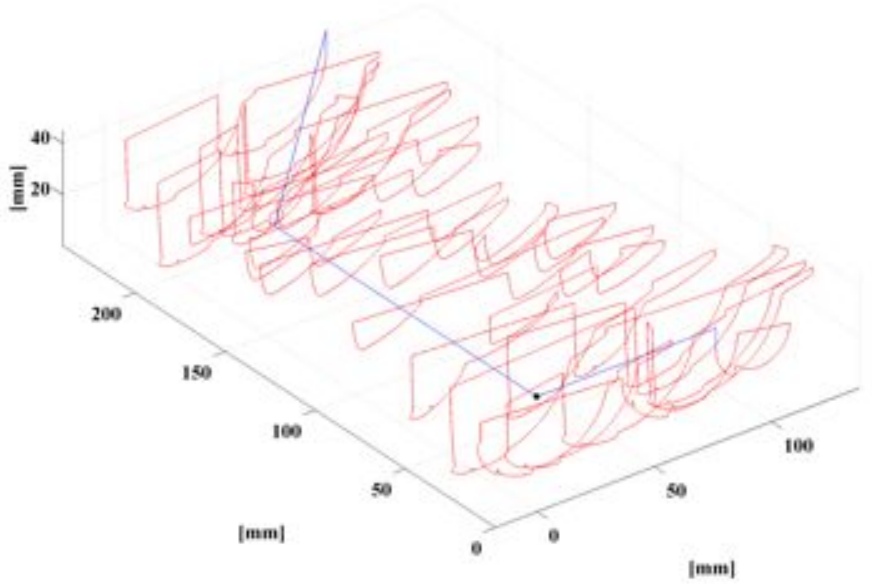


Figure 5-13 Random Distribution of the Base Curves of the Handset over X-axis

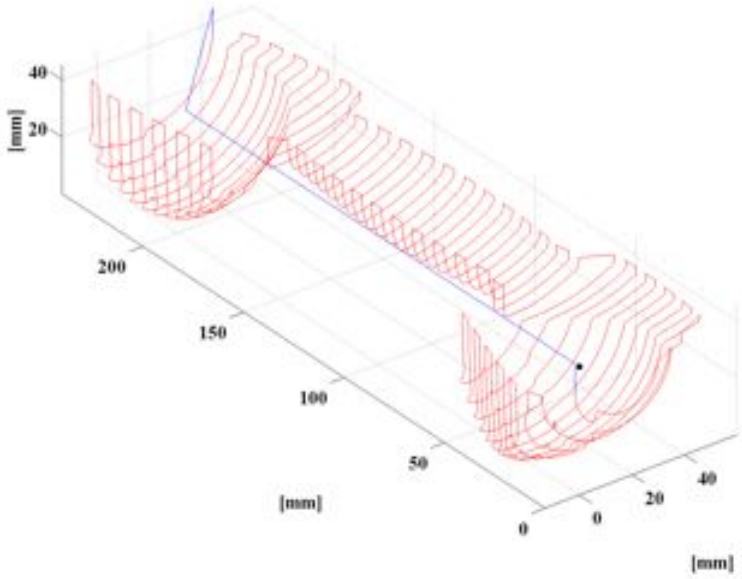


Figure 5-14 Base Curves of the Handset Placed at Half of Their Exact Distances

5.6 Parallel Processors

As described in the third section of this chapter, there are multiple kernels available on the VEPRO hardware. Four commands are introduced to utilize these kernels efficiently in the upcoming section.

As an example, the generic VEPRO program written for the discussed test cases is modified to use four auxiliary kernels available on the hardware. The new program is provided in Table 5-5. In this application, the tasks of curve offset generation are distributed over the auxiliary kernels via using appropriate loop indexes and offset radii. The internal VEPRO programs of the kernels are provided at the end of the table. As can be observed from the table that the tasks accomplished in the kernels are similar. After the kernels are enabled with the **STA** command of the VEPRO, the tasks assigned to the kernels are processed and the results are conveyed to the master kernel with an acknowledgement signal (**TIC**).

Table 5-5 Utilization of the Parallel Processors in the Generic VEPRO Program for the Three Test Cases

```
; VEPRO program for the "Bottle","Sphere", and "Handset" cases
;
               DIM     3       ; 3-axis (X,Y,Z)
               WOR     4       ; use 4-byte word (32-bit)
               FPL     0       ; no fixed-point multiplication
               NCR     3       ; 3 dynamic arrays including C(0) are needed!
;
; Define addresses in memory
;
X0             EQU     0x0     ; initial x location (1 word)
Y0             EQU     0x4     ; initial y location (1 word)
Z0             EQU     0x8     ; initial z location (1 word)
DeltaY         EQU     0xC     ; increment in Y (1 word)
Rtab           EQU     0x10    ; starting address of offset table (50 words)
Bcur           EQU     0xD8    ; starting address of base curve (2*1000 words)
;
; Assign labels to registers (or constants)
;
X              DEF     R(1)    ; position register for axis-1: X
Y              DEF     R(2)    ; position register for axis-2: Y
Z              DEF     R(3)    ; position register for axis-3: Z
i              DEF     R(4)    ; R(4) is to be used for indexing
r              DEF     R(5)    ; R(5) stores the temporary offset value
Zup            DEF     R(6)    ; Absolute z position of the tool in fast movements
;
```

Table 5-5 (continued)

```
; Program starts here
;
                ORG     0x2018  ; starting address(following 2054*4 words of data)
                SLI             ; set linear interpolation mode
                SOE             ; enable output
                CLR             ; clear all curve registers
                MOV     Zup,5   ; Zup is assigned to be 5 mm.
                MOV     X,X0    ; load initial tool locations to pos. registers
                MOV     Y,Y0
                MOV     Z,Z0
                MOV     C(1.1),Rtab,0,199       ; load offset values to C(1) (1D)
                MOV     C(2.1),Bcur,0,3999      ; X-axis components of base curve
                MOV     C(2.3),Bcur,4000,7999   ; Z-axis components of base curve
                MOV     i,5                     ; initial index value (“for loop”)
                STA     b0001111                ; start the auxiliary kernels
Loop:           MOV     r,C(1.1),i-4            ; r := C(1.1)[i-4]
                GCO     C(2),r                  ; generate 2D offsets of C(2) with r
                WAI                             ; wait for the other kernels
                OUT     C(0)                    ; output C(0) holding the offset
                OUT     K1.C(0)                 ; output C(0) of the first kernel
                OUT     K2.C(0)                 ; output C(0) of the second kernel
                OUT     K3.C(0)                 ; output C(0) of the third kernel
                OUT     K4.C(0)                 ; output C(0) of the fourth kernel
                MOV     C(2),C(2),3999,0        ; make sure to flip C(2)
                ADD     Y,DeltaY*5              ; Y coordinate increases by DeltaY*5
                ADD     i,5                     ; increase loop index i by 5
                CMP     i,105                   ; if i is not equal to 105,
                BNE     Loop                    ; loop continues
                CLR     C(0)                    ; clear C(0)
                MOV     X,X0                    ; load the initial position
                MOV     Y,Y0
                MOV     Z,Z0
                OUT     C(0)                    ; perform linear interpolation
                END

                BGN     1
                MOV     C(2),C(2),3999,0        ; make sure to flip C(2)
                ADD     Y,DeltaY*1              ; Y coordinate increases by DeltaY*1
                MOV     r,C(1.1),i-3            ; r := C(1.1)[i-3]
                GCO     C(2),r                  ; generate offsets of C(2) with r
                TIC     1                       ; send the acknowledgement signal
                END     1

                BGN     2
                ADD     Y,DeltaY*2              ; Y coordinate increases by DeltaY*2
                MOV     r,C(1.1),i-2            ; r := C(1.1)[i-2]
                GCO     C(2),r                  ; generate offsets of C(2) with r
                TIC     2                       ; send the acknowledgement signal
                END     2

                BGN     3
                MOV     C(2),C(2),3999,0        ; make sure to flip C(2)
                ADD     Y,DeltaY*3              ; Y coordinate increases by DeltaY*3
                MOV     r,C(1.1),i-1            ; r := C(1.1)[i-1]
                GCO     C(2),r                  ; generate offsets of C(2) with r
                TIC     3                       ; send the acknowledgement signal
                END     3

                BGN     4
                ADD     Y,DeltaY*4              ; Y coordinate increases by DeltaY*4
                MOV     r,C(1.1),i              ; r := C(1.1)[i]
                GCO     C(2),r                  ; generate offsets of C(2) with r
                TIC     4                       ; send the acknowledgement
                END     4
```

5.7 Comparison with the Conventional Approach

In this section of the chapter, another application area of the VEPRO is evaluated. 2.5D pocketing operations are employed on two different test cases (Flower and Rabbit, which are modified version of test cases presented in [99]) via writing an appropriate VEPRO program.

Rendered SolidWorks parts of these two cases are shown in Figure 5-15 and Figure 5-16. The flower and rabbit figures of [99] are modified such that they now fit into $200mm \times 300mm$. The SolidWorks parts of the test cases are formed from the discrete points of the figures. Due to the memory problems, the number sof points in Flower and Rabbit are limited to 44022 and 33011, respectively. The depth of the pocketing operation is $3.5\ mm$ in each of the cases.

As opposed to the previous section, these cases are realized with a CNC machining center available at the machine shop of the department. The parameters of the machine are given in Table 5-6. The CNC machining center has 3 axes and the resolution of the encoders for each axis is $10000\ pulse/rev$. The CNC controller of the machine is Sinumerik 802D. Thus, the written NC programs should be compatible with Sinumerik. Due to the complexities of the test cases, it is difficult to write NC programs manually. Instead, a freeware CAM software (HSM Express) is utilized to generate required NC programs for the pocketing operations. The properties of the obtained NC files and the workpieces are summarized in Table 5-7. As can be inferred from the first row of the table, the generated NC files are very long (117123 lines for the Rabbit and 191626 lines for the Flower). The reason of these lengthy files is that in automatic generation the points on the figures are simply connected with the NC commands G1, G2, and G3. The numbers of these commands are also provided in the table. Three different sizes of milling tools are used in both of the cases. While pocketing is done in two levels for the Rabbit, it is completed in three levels for the Flower case. The sequences of the machining levels and the accompanying tools are given in Table 5-8. The Rabbit is manufactured in 4 sequences, but it took 6

sequences for the Flower to be produced from Aluminum 7075. Due to the additional sequences, the manufacturing of the Flower took 100 minutes longer than the Rabbit. During the manufacturing of the test cases, Boron Oil & Water is used as the coolant.

The images of the produced parts are provided in Figure 5-17 and Figure 5-18. The scratches of the tools can clearly be seen on the machined surfaces. The two test cases are machined on the same aluminum block. Since the size of the Flower is smaller than the Rabbit, there is extra space left on the Flower side of the aluminum.

Figure 5-15 Rendered SolidWorks Part of the Test Case Flower

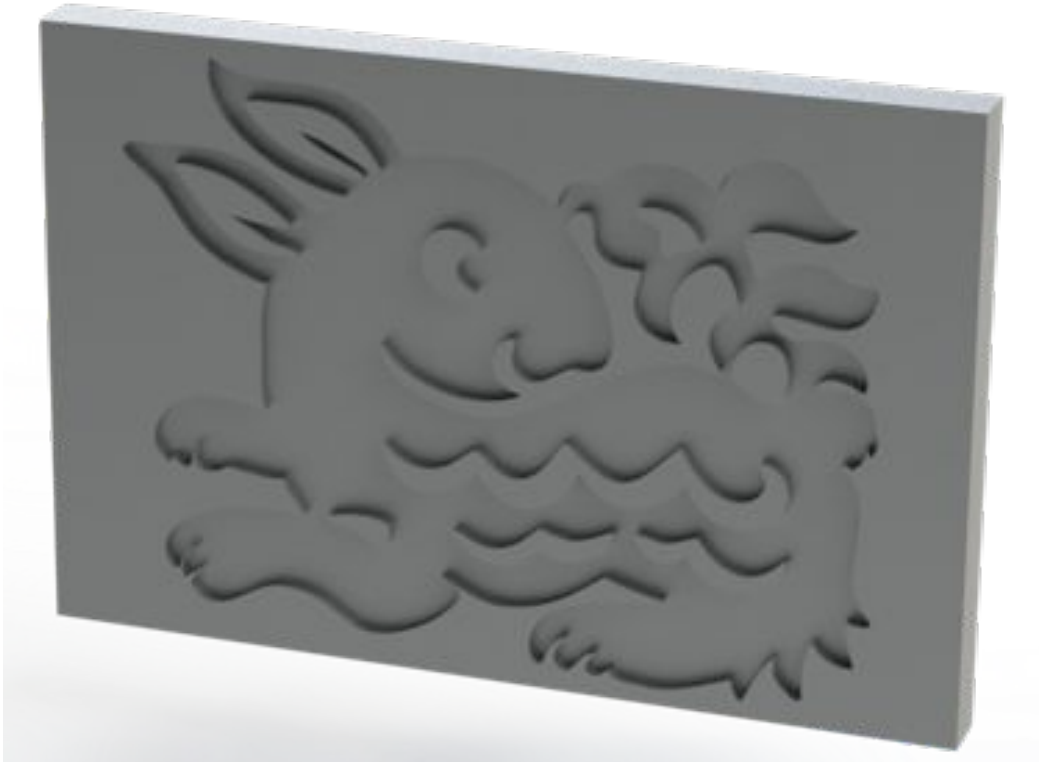


Figure 5-16 Rendered SolidWorks Part of the Test Case Rabbit

Table 5-6 Parameters of the CNC Machining Center

Parameter	Symbol	Unit	*x-axis*	*y-axis*	*z-axis*
Mass	m	kg	130	331.97	260
Dry friction force	F_f	N	200	200	200
Moment of inertia	J	kgm^2	7.9941×10^{-3}	16.4838×10^{-3}	19.7446×10^{-3}
Dry friction torque	T_f	N	1.1	1.5	2.1
Viscous friction coefficient	b	Nms/rad	0.0005	0.0005	0.0005
Equivalent moment of inertia	J_{eq}	kgm^2	0.00834	0.01737	0.02044
Equivalent dry friction	$T_{f,eq}$	Nm	1.435	1.835	2.435
Ball screw lead	h_s	m	0.010	0.010	0.010
Ball screw efficiency	η_s	-	0.95	0.95	0.95
Rated torque	T_r	Nm	12	22	30
Rated speed	ω_r	rad/s	209.44	209.44	209.44
Rated power	P_r	W	2,094.4	3,769.9	4,398.2
Torque-speed slope	m_T	Nms/rad	-0.00955	-0.01910	-0.04297
Encoder resolution	-	pulses/rev	10,000	10,000	10,000

Table 5-7 Properties of the NC Files and the Workpieces

Property	Rabbit	Flower
# of Command Lines	117123	191626
# of G0 Commands	125	324
# of G1 Commands	15472	32382
# of G2 Commands	16958	33976
# of G3 Commands	3338	3102
# of Tool Changes	3	5
Duration of Machining	320’	420’
Tool Diameters (mm)	2, 4, 6	2, 4, 6
Coolant	Boron Oil (10%) & Water (90%)	Boron Oil (10%) & Water (90%)
Feedrate (mm/min)	350	350
Spindle (rpm)	2000	2000
Axial Depth (mm)	3.5	3.5
Levels of Pocketing (mm)	2.5 and 3.5	1, 2, 2.5, and 3.5
Material	Aluminum 7075	Aluminum 7075
Workpiece Size (mm)	300 × 200 × 20	300 × 200 × 20
Feature Size (mm)	244.1 × 187.2	189.4 × 182.1
Number of Points in Base Curve	33011	44022
NC File Size (MB)	2.53	4.15
Zipped File Size (MB)	0.6	0.64

Table 5-8 Machining Sequences of the Test Cases

Sequence	Rabbit	Flower
1	ϕ6 & Z2.5	ϕ6 & Z2
2	ϕ4 & Z2.5	ϕ4 & Z2
3	ϕ2 & Z2.5	ϕ2 & Z1
4	ϕ2 & Z3.5	ϕ2 & Z2.5
5	-	ϕ4 & Z3.5
6	-	ϕ2 & Z3.5

Figure 5-17 Manufactured Test Case Flower

Figure 5-18 Manufactured Test Case Rabbit

After the manufacturing of the test cases, a generic VEPRO program is written for the comparison purposes with the conventional approach. The program is provided in Table 5-9. The main difference of this program from the previous one is that it utilizes **GCO** command of the VEPRO to generate curve offsets of the base curves for pocketing operations. By only changing the base curves, offset tables, number of sequences, and the height of the levels one can obtain trajectories for 2.5D pocketing operations. The written VEPRO program is also emulated in MATLAB and the results of these emulations are given in Figure 5-19 and Figure 5-20. As in the previous section, the red lines represents machining and the blue lines represents fast movements of the tools utilized. The properties of the generated tool trajectories are summarized in Table 5-10. As opposed to the previous section, the number of points on the base curves are increased in this section for a better quality of the curve offsets.

Table 5-9 Generic VEPRO Program for the Test Cases Flower and Rabbit

```
;
; VEPRO program for the "Rabbit" and "Flower" cases
;
               DIM     3       ; 3-axis (X,Y,Z)
               WOR     4       ; use 4-byte word (32-bit)
               FPL     0       ; no fixed-point multiplication
               NCR     3       ; 3 dynamic arrays including C(0) are needed!
;
; Define addresses in memory
;
X0             EQU     0x0     ; initial x location (1 word)
Y0             EQU     0x4     ; initial y location (1 word)
Z0             EQU     0x8     ; initial z location (1 word)
Rtab           EQU     0x9     ; starting address of offset table (32/40 words)
Levels         EQU     0xA9    ;  starting  address  of  levels  table  (2/4  words)
OffsetIndex    EQU     0xB9    ; starting address of offset index table (2/4 words)
Bcur           EQU     0xC9    ; starting address of base curve (2*220022 words)
;
; Assign labels to registers (or constants)
;
X              DEF     R(1)    ; position register for axis-1: X
Y              DEF     R(2)    ; position register for axis-2: Y
Z              DEF     R(3)    ; position register for axis-3: Z
i              DEF     R(4)    ; R(4) is to be used for indexing
r              DEF     R(5)    ; R(5) stores the temporary offset value
j              DEF     R(6)    ; R(6) is to be used for indexing
k              DEF     R(7)    ; R(7) is to be used for indexing
Zup            DEF     R(8)    ; Absolute z position of the tool in fast movements
NumLevels      DEF     R(9)    ; Number of levels in the machining
;
; Program starts here
;
               ORG     0x0001ADC69    ; starting address
```

Table 5-9 (continued)

```
             SLI                    ; set linear interpolation mode
             SOE                                ; enable output
             CLR                                ; clear all curve registers
             MOV    i,0                         ; initial index value
             MOV    j,0                         ; initial index value
             MOV    Zup,5                       ; Zup is assigned to be 5 mm.
             MOV    X,X0                        ; load initial tool locations
             MOV    Y,Y0                        ; to position registers
             MOV    C(1.1),Rtab,0,159           ; load offsets to C(1) (1D)
             MOV    C(2.1),Bcur,0,880088        ; define X of base curve
             MOV    C(2.2),Bcur,880089,1760176  ; define Y of base curve
             MOV    C(3.1), Levels,0,15         ; load levels to C(3) (1D)
             MOV    C(4.1), OffsetIndex,0,15    ; load indexes to C(4) (1D)
             MOV    Z,C(3.1),j                  ; Initial level
Loop:        MOV    r,C(1.1),i                  ; r := C(1.1)[i]
             GCO    C(2),r  ; generate 2D curve offset of C(2) with r
             OUT    C(0)    ; output C(0) that holds the current curve offset
             INC    i               ; increase loop index i
             CMP    i,C(4.1),j      ; if i not equal to next level starting index
             BNE    Loop            ; then continue with the loop
             INC    j               ; increase loop index j
             MOV    Z,C(3.1),j      ; Level changes
             CLR    ZF              ; Clear the flag ZF
             CMP    j,NumLevels     ; if j is not equal to Numlevels,
             BNE    Loop            ; then continue with the loop
             CLR    C(0)            ; clear C(0)
             MOV    X,X0            ; load the initial pos. to the resisters
             MOV    Y,Y0
             MOV    Z,Z0
             OUT    C(0)            ; perform linear int. to the initial position
             END
```

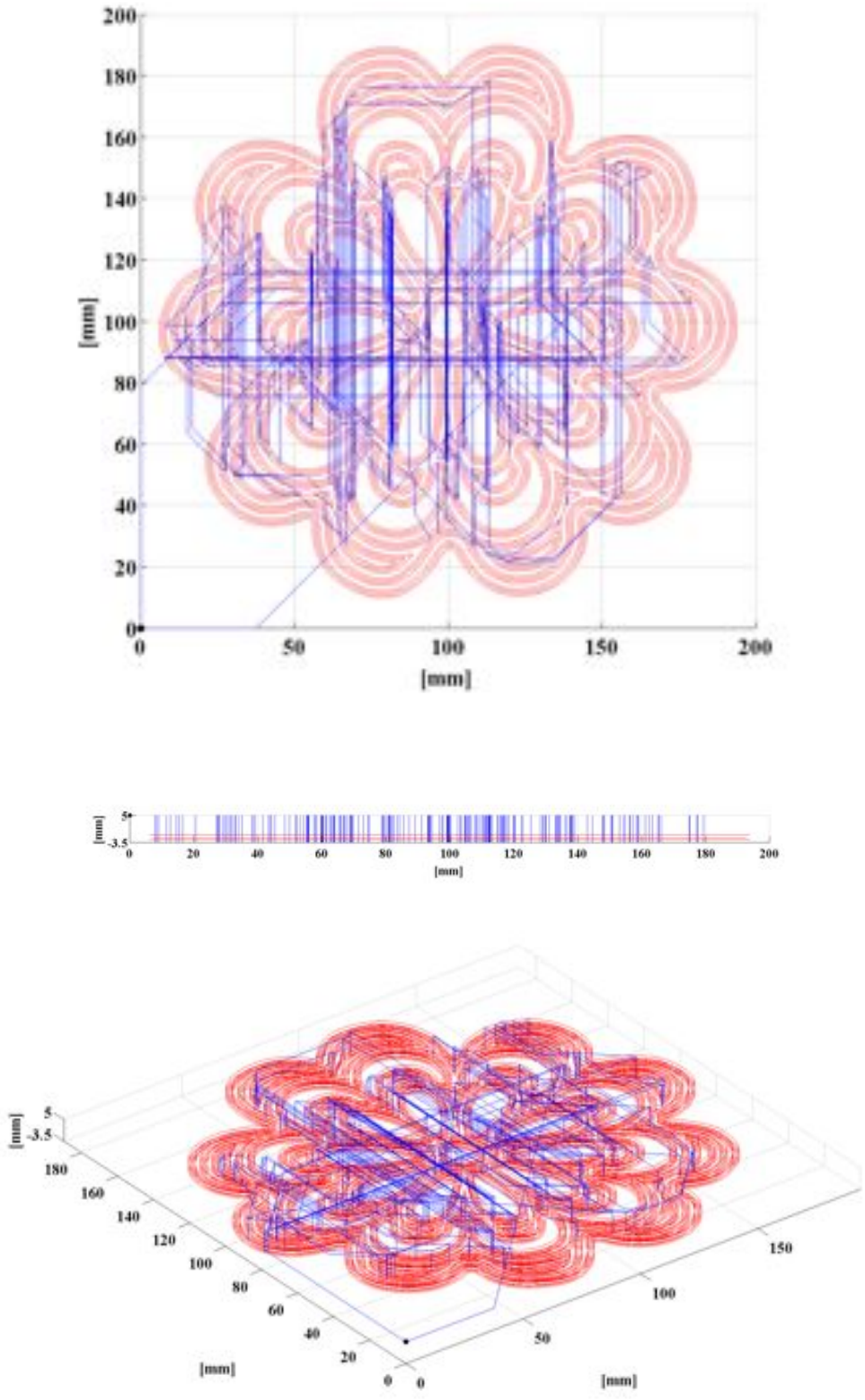


Figure 5-19 Plots of MATLAB Emulation of the Test Case Flower

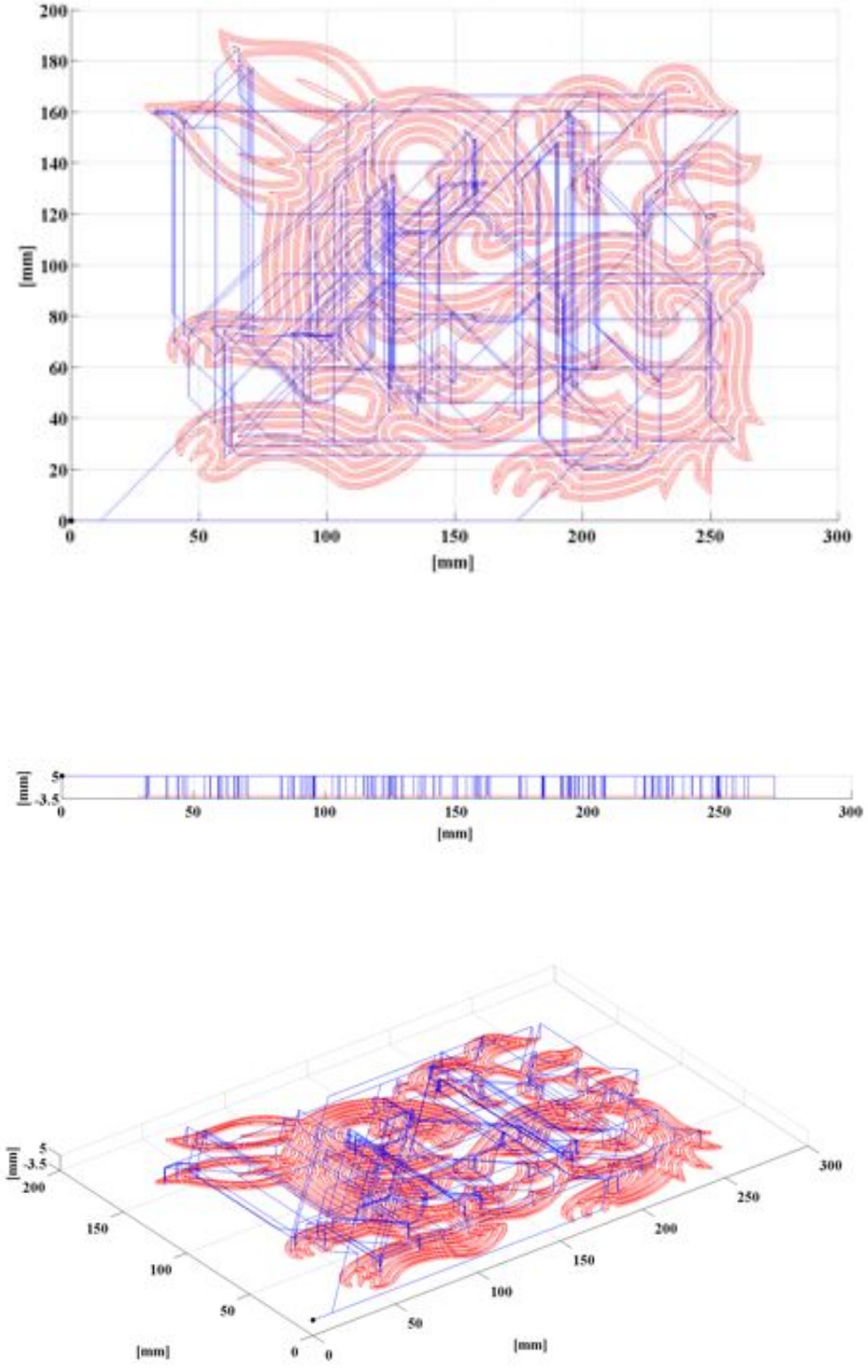


Figure 5-20 Plots of MATLAB Emulation of the Test Case Rabbit

Table 5-10 Data Attributes of the Test Cases Rabbit and Flower

Description	Rabbit	Flower
Number of Commands in Base Curve	220,011	220,022
Size of Original Base Curve (Bytes)	1,320,066	1,320,132
Size of Compressed (ΔY10) Base Curve (Bytes)	133,433	128,762
Number of Commands in Tool Trajectory	3,907,907	4,789,165
Size of Original Tool Trajectory (Bytes)	31,263,256	38,313,320
Size of Compressed (ΔY10) Tool Trajectory (Bytes)	3,852,223	3,680,187

It can be inferred from Table 5-10 that by compressing the base curves via ΔY10 compression algorithm and then utilizing the VEPRO commands do outperform the conventional approach in terms of memory. The original tool trajectories can be represented with 1% of the initial memory sizes with the VEPRO. For instance, for the test case Rabbit the compression ratio is calculated as

$$r = \frac{133{,}433}{31{,}263{,}256} = 0.4\% \tag{5-1}$$

When this low compression ratio and the primitive hardware to employ VEPRO are considered together, one can conclude that Kolmogorov complexity of the conventional approach is higher than the proposed paradigm.

5.8 Conclusion

A novel motion command generation paradigm is proposed in the chapter. As in the conventional approach, the motion trajectory is defined manually by writing the VEPRO program using the given commands of VEPRO language. After it is compiled on the host PC, the output of the compilation (a.k.a. machine code) is transferred to the VEPRO hardware through different kinds of communication for processing. Then the hardware generates the motion command with the help of its auxiliary units on the hardware.

The chapter starts with the review of the primitive version of this command generation paradigm. Then the VEPRO is introduced by describing the hardware layout of a possible SOC implementation. In the fourth section of the chapter, the commands of the VEPRO language are presented along with their explanations. Different types of test cases are evaluated in the fifth and sixth sections. When the results of MATLAB emulation of these cases are considered, it can be noted that with the same program one can machine different shapes by just changing the offset tables and base curve sets.

Apart from the given test cases in the previous sections of the chapter, VEPRO can also be utilized to generate various machining trajectories for different types of CNC machinery and robotics applications. The proposed paradigm is still under development. According to the new application fields, different types of commands can be embedded into the VEPRO in the near future.

CHAPTER 6

FPGA IMPLEMENTATION OF COMMAND GENERATOR

This chapter of the thesis focuses on the implementation of a novel motion command generator for servo-motor drives using FPGAs. The underlying method, which incorporates a new data compression algorithm (ΔY10), is capable of generating trajectory data at variable rates in forward and reverse directions. In this paradigm, higher-order differences of a given trajectory (i.e. position) are first computed and the resulting data are compacted via the proposed compression technique. After the compressed data is transferred onto the memory chip of the FPGA development board, the generation of the commands is carried out according to the feedrate (i.e. the speed along the trajectory) arranged by the external logic dynamically. The FPGA implementation is realized in two different approaches (hardwired and softcore). Then the performance of this system is evaluated in terms of the hardware resources used in different aspects by employing it on two different test cases.

6.1 Introduction

The current state of the art motion drive systems utilize on-board controllers with digital signal processors and micro-controllers to control the position and the velocity of the servo-motors. They are also used to regulate precisely the phase currents of the motors to control the electromagnetic torque. Despite these capabilities, in most of the multi-axis motion control applications, a central motion control unit is utilized commonly to generate velocity and/or position trajectories. The generated motion signals are then conveyed to each motor driver over a serial communication protocol. As an alternative to this conventional approach, a direct command generation system with variable feedrate for servo motor drives is aimed in the study. The proposed architecture can also be utilized to generate commands for machines having more than one axis. In these cases, an additional unit embedded into the hardware is necessary to synchronize the single axis motion command generators. The proposed novel approach produces the commands directly in the drive system from the encoded (with ΔY10 encoding method) data set. There is no need for the data transfer from the central control unit of the overall system. This proposed scheme is implemented on an FPGA development board within the scope of the study. With the proposed hardware implementation, the novel command generation paradigm can easily be realized as an embedded part of the motor drive systems.

When the related literature discussed in Chapter 2 is considered, it can be concluded that there are studies on the FPGA implementations of the data compression algorithms and digital control of CNC based applications, but there are not any studies on the command generation for CNC machinery and robotics utilizing FPGA and compressed motion data. The main contribution of the chapter is that it improves the command generation scheme proposed by Yaman and Dolen [66] and realizes the method on an FPGA development board by employing it on some test cases.

The rest of the chapter is organized as follows: Next section discusses the details of the novel command generation method, and the following section presents the hardware architecture of FPGA implementation of the method. After the test cases are described and the performance of the method is evaluated in the fourth section, the chapter is concluded with the closure section.

6.2 Proposed Technique

The system basically depends on the compression of differentiated motion command data. Computer aided manufacturing or robotics software generates the motion command trajectory data according to the predefined sampling frequency. After the higher order differences of the trajectory are taken, the differentiated sequence is compressed with the proposed compression method. The transferred data is decoded on the command generator and conveyed to the motion control system. With the proposed scheme, the utilization of an intermediate programming file (such as NC files) describing the path of the tool is eliminated. The motion controller directly uses the output of the decoder embedded on the controller board as the reference trajectory generator. Although the details of the command generation paradigm are discussed in Chapter 4, the summary of the method is provided in the following parts for the integrity of this chapter.

6.2.1 Encoding of ΔY10

The encoding part of the scheme is composed of relative encoding and compression processes. The details of these steps are discussed in the following subsections.

6.2.1.1 Relative Encoding

In relative encoding, the higher order differences of sequences of the trajectories are computed. This approach decreases the memory requirement of the original trajectory sequence since the magnitudes of the values do decrease extensively. The order of decrease in the memory cost depends on the type of the application. Considerable decrease in the memory is mostly achieved when the order is 3 or 4 for CNC machining applications, but when the sampling frequency is about 1 kHz or higher the best order of difference may be 2 [66]. The decoding can easily be completed, provided that the initial values are given, by the utilization of certain number of accumulators. Due to the employment of relative encoding, the decoder is capable of generating position, velocity and acceleration trajectories at the same time. This is one of the main advantages of the proposed scheme.

6.2.1.2 Compression Process

The developed command generation encoding technique ΔY10 is employed on a simple test case in order to describe the phases of the compression part of it clearly. This is illustrated in Figure 6-1. In the first phase of the command generation encoding method, higher order differences of the command sequence are taken. When the example in the figure is considered, after the first order of difference, the magnitude of the command values do decrease considerably. At this stage, the first initial value should be stored to recover the original command sequence. Since the higher order difference of this example is determined to be two, one more discrete differentiation is carried out on the resulting sequence. The second initial value is also stored. After the relative encoding part is completed, the construction of the AF is started. In the first stage of this construction, the commands in the sequence are eliminated from their signs. Then in the second stage, the commands are represented as Variable Length Binary Numbers. If there are any zero sequences in the first stage, their binary representations are modified

in the third stage by new zero sequences having lengths equal to the lengths of the original zero sequences in binary form. In order to recover the magnitudes of the differentiated commands, another field called LF is constructed. It simply constitutes of sequences of ones and zeros. The length of these sequences determines the length of the commands in the AF. The third field of the encoding method stores the signs of the differentiated commands. *1* is used for negative values and *0* is reserved for positive ones. No bits are used for zero values. The last field of the compression part is the ZF. Here, the lengths of the sequences are stored. Their lengths are determined from the corresponding part of the LF. After the construction of the fields is done, the resulting codes (initial values and fields) are coupled with necessary descriptions (header) and stored into the memory.

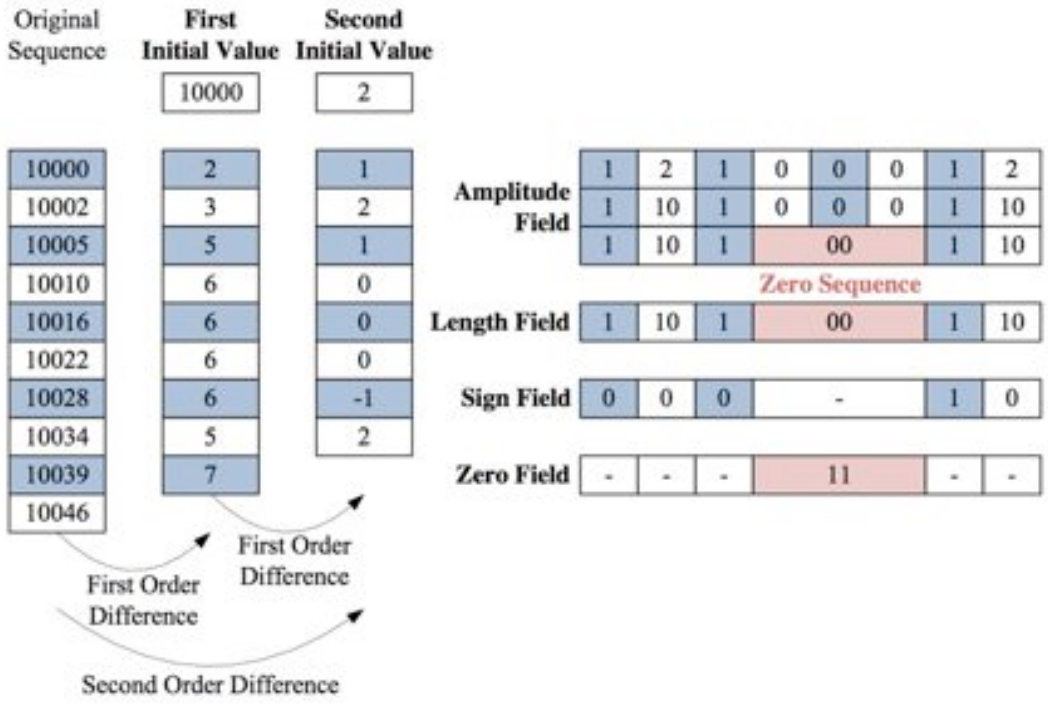


Figure 6-1 Encoding of a Sample Sequence via ΔY10 Technique

6.2.2 Decoding of ΔY10

The decoding part of the command generation scheme has also two main steps: decompression and linear interpolation.

6.2.2.1 Decompression Process

The decompression procedure of the proposed command generation method is easier than its encoding process. It starts with determining the length of each command from the LF by counting the bits in between the successive bit transitions. By knowing the length of the commands, the absolute values of them are obtained from the AF. The signs of these commands are fetched from the SF in order to obtain the signed integer versions of commands. During decoding if it is found that the magnitude of the command is zero and its length is greater than one, then it is concluded that there is a sequence of zero. The number of zeros in this sequence is determined from the ZF. During the decompression of the differentiated data, initial conditions are conveyed to the accumulators from the memory. Finally, the original commands are generated by the last accumulator used in the system.

6.2.2.2 Linear Interpolation

In manufacturing operations with CNC machinery, the operator usually modifies the speed (i.e. feedrate) of tool motion on the workpiece with an external input in order to increase the quality of the product and to eliminate the risks of crack formation on the tool. During the manufacturing processes it may be required to modify the feedrate dynamically through the course of motion by an external input (like feedrate override). Furthermore, under some extreme cases (such as the control scheme of an electro-discharge machine), it is desirable to reverse the direction of motion as dictated by an auxiliary input (e.g. electrode gap voltage).

A variable feedrate input available in forward and reverse directions is integrated into the proposed CG paradigm in order to overcome the aforementioned disadvantages in the manufacturing operations. With the modified feedrate of command generation, extra command values other than the ones in the original sequence should be generated. For this purpose, a linear interpolation is carried out after the decompression of the original values. This is accomplished by the Interpolator Unit in the hardware design. It simply interpolates between the two decoded command values and interpolates according to the current value of the feedrate input.

6.3 FPGA Implementation

Altera FPGA DE1 Development Board is utilized in the study in order to implement the proposed command generation scheme. There are different kinds of memory devices (SRAM, SDRAM, SD Card, Flash) on the board. Among these chips SDRAM is selected due to its ease of control and memory capacity (8 MB). Two different approaches are used to implement the command generator. In the first approach, the command generation method is directly written in VHSIC Hardware Description Language (VHDL) and it is called as hardwired approach. A softcore processor IP (NIOS II) serving as an embedded microcontroller is utilized in the second approach and named as the softcore approach.

The hardware architecture of the proposed command generation scheme for the hardwired approach is given in Figure 6-2. It is an improved and more stable architecture of the one proposed by Yaman et al. [29] for a different compression algorithm. There are eight different modules, a bus line and inputs in the architecture. All of the inputs (Start/Pause, Reset, Clock, Speed, Direction) are connected to the bus line. Except the Clock input, the rest are supplied to the system from the buttons and switches located on the FPGA development board. The Clock input is directly connected to the system from the software. The Speed

input (7 bit) is simulated with seven switches forming an unsigned integer. All of the data lines are 32 bit and the address line is 13 bit.

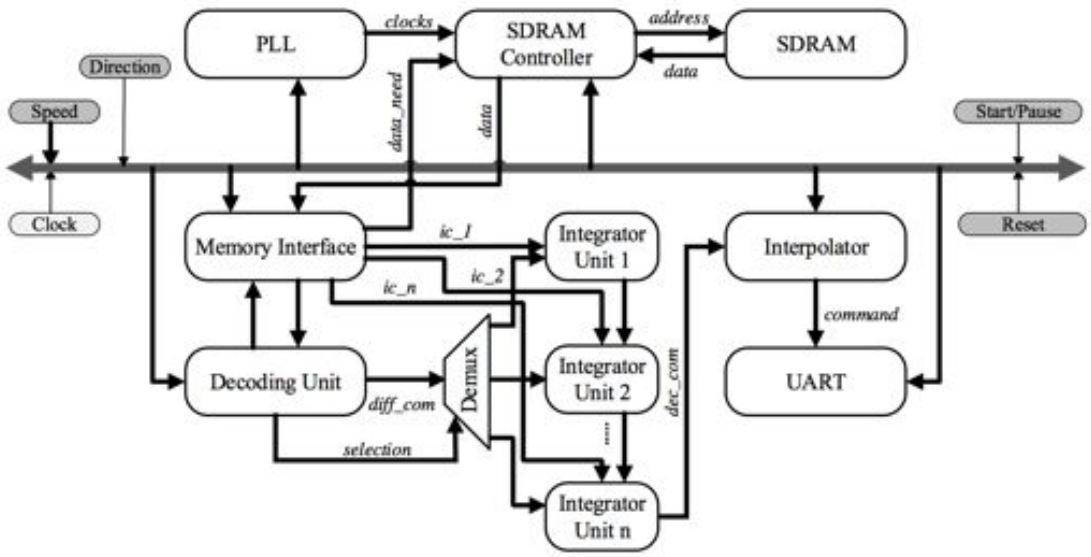


Figure 6-2 Decompression Architecture

On the upper part of the bus line, blocks necessary for the memory operations are located. Phase Locked Loop (PLL) block generates necessary timing signals for the SDRAM Controller and has no other tasks. SDRAM is only connected to the SDRAM Controller. Thus, it is the only block that can read and write to SDRAM. The modules to the below of the bus line communicates with the SDRAM Controller through Memory Interface. The Decoding Unit is the core of the architecture. It gets the required data from the Memory Interface and conveys the decoded commands to Integrator Units. There are *n* number of Integrator Units in the design. According to the order of the difference defined in the header part of the compressed file, the Decoding Unit selects the starting integrator via a demultiplexer and conveys the decoded differentiated commands to this integrator. Initial conditions for each integrator unit are transferred to them at the start of the system from the Memory Interface. The circuit schematics of an

Integrator Unit is illustrated in Figure 6-3. There are two adder block and a register storing the latest command value. If it is not the first Integrator Unit in the system, the differentiated command is not supplied to the circuit. Integrated commands coming from the previous unit is only fed to the corresponding unit. After the integration is done with *n* number of units, the Interpolator Unit transfers the commands to the UART Module for monitoring. When a controller is present in the system, the Interpolator Unit should also send the trajectory commands to the controller.

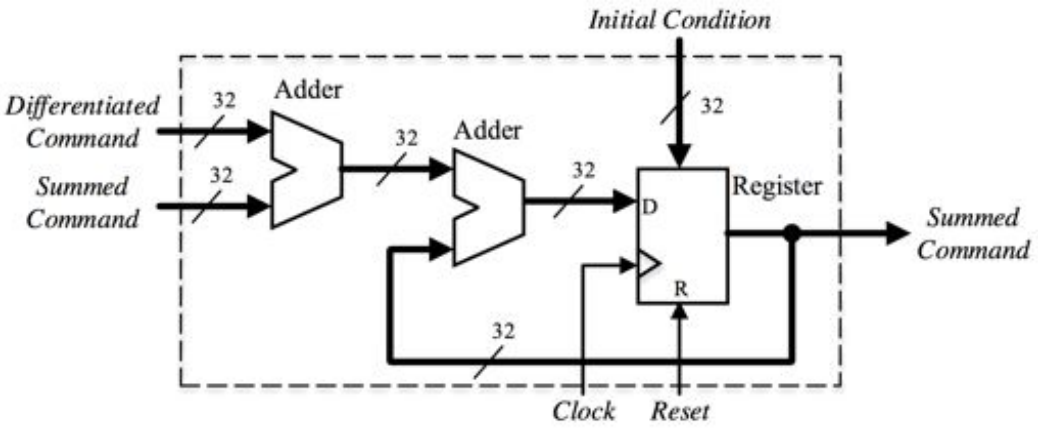


Figure 6-3 Integrator Unit

To decrease the complexity of the coding in VHDL, the schematic design property of Quartus II 11.1 Web Edition is employed. The hardware architecture (Figure 6-2) given is constructed within the software. The following sections elaborate the design of the units by explaining the State Transition Diagrams (STDs) utilized in each unit.

6.3.1 Hardwired Approach

In this approach of the FPGA implementation of the command generation scheme, the hardware architecture illustrated in Figure 6-2 is implemented via designing the state diagrams of each module in Altera Quartus software. The details of each module are described in the following subsections.

6.3.1.1 SDRAM Controller

SDRAM Controller in the architecture is used to read and write to SDRAM located on the board over the Memory Interface in order not to deal with the details of the reading and writing procedures. The Memory Interface sends the related address information of the data required and the controller conveys the data to the Memory Interface. The controller utilized in the implementation is developed by Altera. The two outputs of the controller are connected to the Memory Interface to receive the specified words and the rest of the outputs are attached to the SDRAM chip on the board.

In order to use the memory efficiently, the compressed code is structured as shown in Figure 6-4 for a generic command sequence. The first three words of the compressed data can be regarded as the header. Initial 4 bits of the first word indicate the order of finite difference (where a maximum of 15th order for the differences can be represented). The rest of the first word (28 bits) is reserved for expressing the length of the command sequence. The second word of the header is used to specify the number of words reserved for the AF and LF, which indirectly determines the starting address of the SF. Finally, the last word of the header gives the starting address of the ZF. After the header part, the initial values section is located. They are stored in the form of signed binary integers. The number of initial values necessary for integration is set by the order of finite difference which is represented with the first 4 bits of the data. After the information about

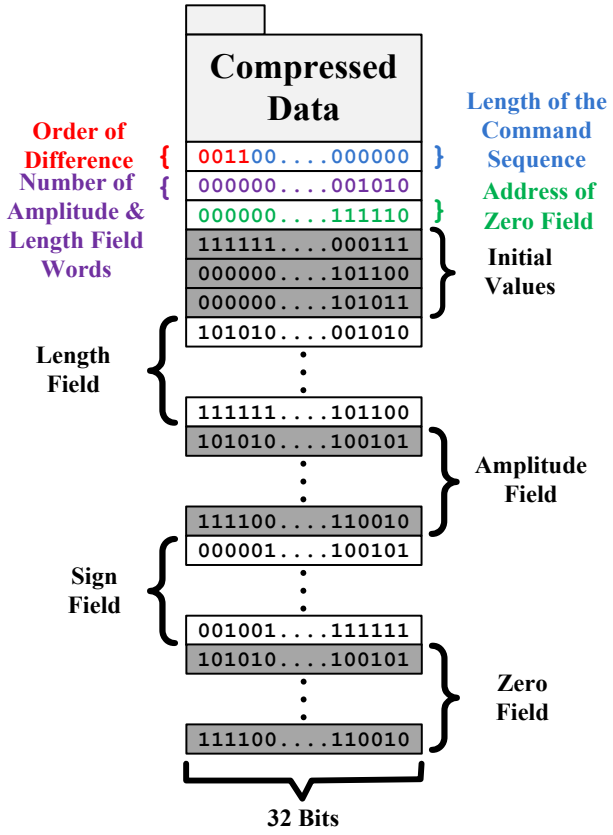


Figure 6-4 Compressed File Format

the compressed data and initial values are given, the LF is then stored in the proceeding words. Since the length of the header part and the number of initial values are known, the starting address of the LF is easily determined during decoding. Note that the amplitude and sign fields are located after the LF. The starting addresses of these two fields are calculated via the number of amplitude and length field words stored in the second word of the header. The last section of the compressed file consists of ZF words, whose starting address is provided in the third word of the header. With the described data format, the compressed sequences are generated without any error.

6.3.1.2 Memory Interface

The second module to be introduced is the memory interface. Its main task is to establish and maintain the communication between the SDRAM on the development board and the other modules present in the structure. The STD developed for memory interface is given in Figure 6-5. There are 6 states in the diagram. The first state is the *Idle* state. After it is initiated with the *Start* or *Reset* inputs, the flow is switched to the next state that is *Send Header & Initial Conditions*. In this state, the header is fetched from the memory and sent to the decoding unit. After the header is sent, *Send Amplitude and Length Words* state is activated by setting the *data_need* signal to "*01"*. In the first operation of this state, first two words of the amplitude and length fields are sent to the decoding unit directly (Afterwards only one word of the fields are transferred). Then the operation of the memory interface is controlled by the decoding unit. It sends required signals to the memory interface and fetches words from different fields of the compressed code. At this stage of operation, the state continuously changes between three states *Send Amplitude and Length Words*, *Send Sign Word*, and *Send Zero Word* according to the coming signal (*data_need*) from the decoding unit. When the decoding operation is completed, the final state *Done* is activated.

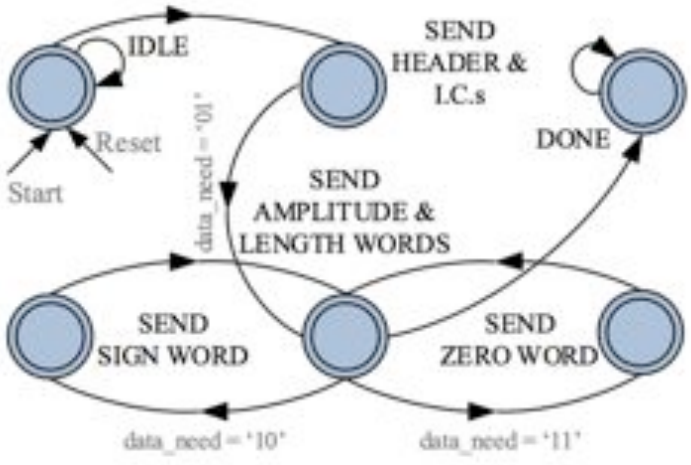


Figure 6-5 STD of the Memory Interface

6.3.1.3 Decoding Unit

This unit can be regarded as the core of the hardware architecture, because ΔY10 decompression algorithm is employed here. There are only two modules (Memory Interface and the first Integrator Unit) communicating with the Decoding Unit. Of these two units, only the Memory Interface gives inputs to the unit. After a differentiated command is decoded in the unit, it is fed to the first integrator in signed integer format along with an acknowledgement clock. STD of the Decoding Unit is given in Figure 6-6. There are fifteen different states in the diagram. Decompression starts with the reset input and firstly the header information is fetched from the SDRAM via Memory Interface. Afterwards, the necessary information for decoding is acquired by processing the header words. Later in the third state of the STD, initial amplitude and length words are fetched from SDRAM. The next state is the *Decode Length Word* state in which the length word is analyzed with bit operations and the length of the command is determined. Then amplitude of the command is determined in the *Decode Amplitude Word* state. When a command has components in two different words, the *Detect Pair* state is activated by setting the signal *pair_flag* to *"1"*. Another

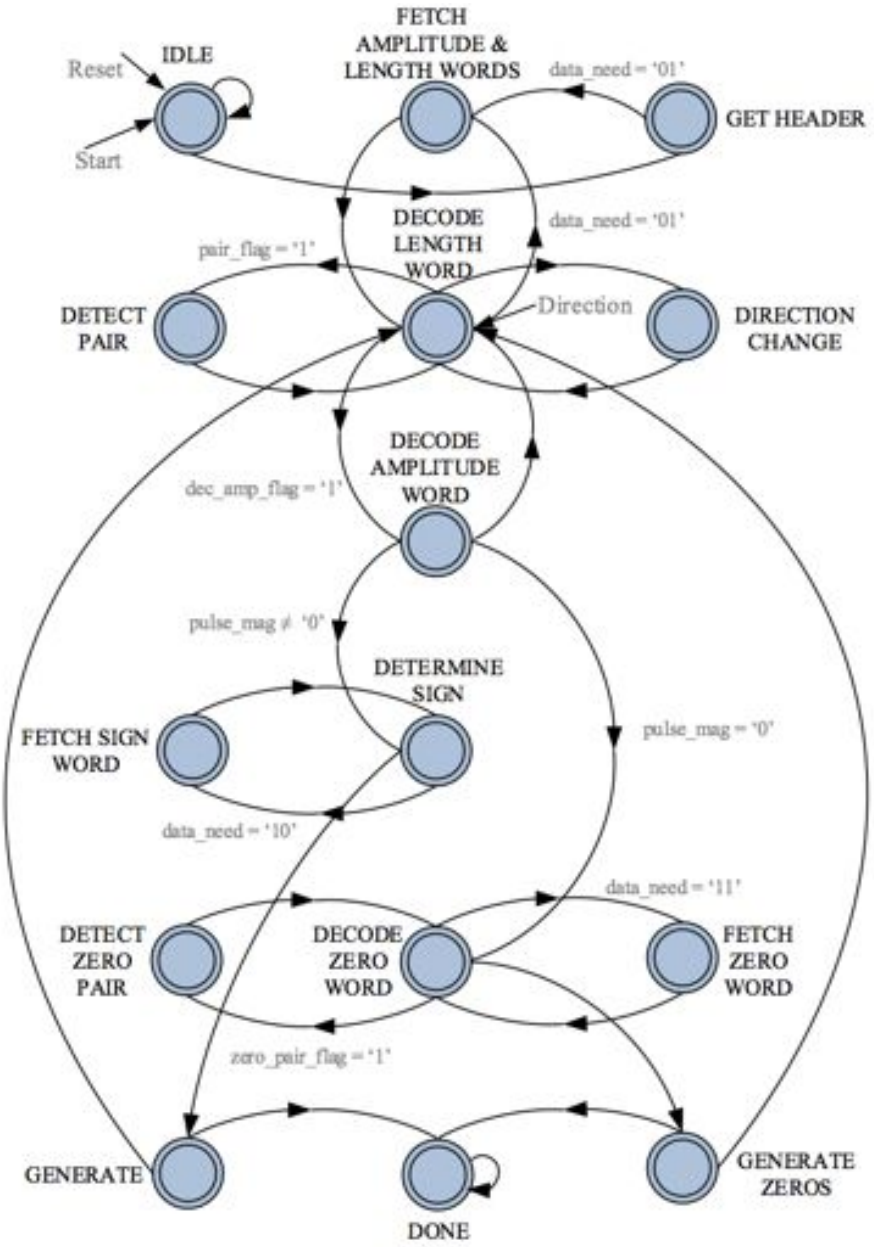


Figure 6-6 STD of the Decoding Unit

state connected to the *Decode Length Word* state is the *Direction Change* state. If the user changes the direction of the decoding, necessary modifications on the variables and the signals are done in this state. When there is a lack of data during decoding, the finite-state machine moves on to *Fetch Amplitude and Length Words* state to obtain the required data. After the amplitude of the command is

determined, it moves to *Determine Sign* state where the sign of the state is assigned. If there is a need for new sign word, the machine moves to *Fetch Sign Word* state. During the determination of the amplitude, if it is turned out that the amplitude is zero and the length is greater than one, it means that there exists a sequence of zeros. This is decoded in *Decode Zero Word* state. This state is in communication with two other states related to the zero field decoding. When the data in the zero field lies in two different words, then the *Detect Zero Pair* state is activated. If there is a lack of zero word, the flow continues to *Fetch Zero Word* state in order to receive the new zero word from the memory interface. After the command is generated in the related states, the machine goes back to the *Decode Length Word* state and continues decoding. When the decoding operation of all commands is accomplished, the last state *Done* is activated.

6.3.1.4 Integrator Unit

The circuit schematic of the Integrator Unit is supplied in Figure 6-3, but this circuit is not capable of handling the reverse directions. Therefore, a simple STD (Figure 6-7) is developed to generate commands in both directions. There are five states in the diagram. As in the other diagrams, the first state is the *Idle* state. After the *Idle* state is initiated, initial conditions are generated in the next state. Then in the third state, commands are generated according to the incoming decoded data from the decoding unit at the rising edge of *data_clk* signal. If the direction of the command generation is changed, the state *Direction Change* performs necessary tasks and continues its operation. When all the commands available in the memory are generated, the last state *Done* is activated.

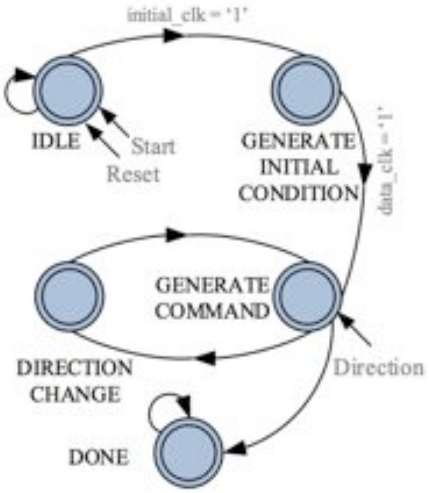


Figure 6-7 STD of the Integrator Unit

6.3.1.5 Interpolator Unit

The interpolator unit performs necessary computations in order to interpolate between the consecutive commands according to the input (*speed*) given by the user. The STD of the interpolator unit is formed with seven states as can be seen in Figure 6-8. The first and the last states are the same with the previous STDs. After the command is supplied to the *Get Data* state, there are two possible next states which are *Interpolate* and *Generate Original Commands*. If the user does not change the speed of command generation (Speed is at its maximum level), original commands are generated according to the frequency specified at the encoding stage. When an interpolation is necessary, the *Interpolate* state is activated. Direction change is detected in this state and necessary computations are carried out in the state *Direction Change*. When the user wants the system to generate constant commands for a period of time (Speed is set to its minimum), the related state performs the task. As usual when the command generation is completed, the last state becomes active.

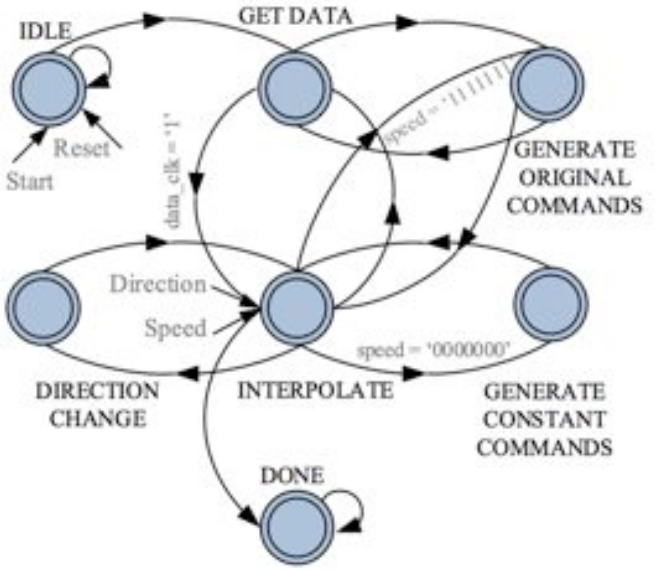


Figure 6-8 STD of the Interpolator Unit

6.3.2 Softcore Approach

In the second approach of the FPGA implementations, rather than building the integrated circuit for the command generation scheme a softcore processor is utilized to generate the commands. For the construction of the processor, the System on a Programmable Chip (SOPC) Builder tool of Quartus is used. After the system is designed, the program required to run the command generation algorithm is written in NIOS II Embedded Development Environment in C language and then downloaded onto the FPGA development board for operation. The details of this implementation approach are given in the following subsections.

6.3.2.1 Construction of the Softcore

The softcore processor and its peripheral units are designed via SOPC Builder. As shown in Figure 6-9, there are ten different modules in the design. The first

module is the Central Processing Unit (CPU) of the design. The most primitive core of NIOS II is selected as the CPU which utilizes about 600 to 700 logic elements of the FPGA chip. The second module is the communication module. The compiled algorithm is transferred to the core through this module that employs serial communication. The third one is the on-chip memory module. Its data width is selected as 32 and total memory size is fixed to 36000 bytes. The preceding module flash controller is added in order to read and write onto the flash memory. The fifth module is utilized to output some of the signals for debugging purposes. Currently there are 8 outputs and they are connected to the LEDs on the FPGA development board. The next module is the controller for the SDRAM. The compressed motion command data are stored into the SDRAM and read via this module in the design. The following module is auxiliary module for the SDRAM controller. It generates necessary clock signals for the controller like PLL module in the hardwired approach. The System ID Peripheral is used to assign a unique ID when the system is generated. The next module Interval Timer is included in the design in order to measure the average command generation time. The last one helps the core get inputs from the user. These inputs are the speed (feedrate) and the direction of the generation along with the Start/Pause switch. After the core is designed, as in the hardwired approach the schematic property of Quartus is used to connect the inputs and outputs to the core (Figure 6-10).

Figure 6-9 Elements in the Softcore

Figure 6-10 Schematic Design of the Softcore Approach

6.3.2.2 Machine Code

In the second part of the softcore approach, the command generation algorithm is written in C programming language and the resulting are cross-compiled to run on the designed softcore processor deployed on the FPGA. The written program is provided in APPENDICES A. After the necessary libraries are included and some definitions are written, the main function of the program begins. Before decoding the compressed motion data, the required field are obtained from the SDRAM and written into the related variables. The start and end addresses of these fields should carefully be obtained. Otherwise, unrelated commands may be generated from the system. After obtaining the fields, the decompression is done and the commands are printed on the console along with the time elapsed.

6.4 Performance Evaluation of the Method

The implementation of the proposed command generation paradigm described in the previous sections is carried out on two different test cases:

- Finishing of a plastic injection mold for a shampoo bottle using a high performance CNC vertical machining center.
- Stencil cutting of a Rabbit [99] via a CNC router.

The command generator is realized utilizing Altera Cyclone II FPGA DE1 Development Board. The trajectories generated for the above-mentioned applications are illustrated from Figure 6-11 to Figure 6-14 and important properties of these test cases are summarized in Table 6-1. It can be inferred from this table that the differenced data set cannot be represented less than 2 bytes regardless of the cases. The Bottle test case has a higher position resolution than the Rabbit test case. Therefore, the differenced data set of the Bottle has much more zero sequences in the compressed form. This results in a decrease in the

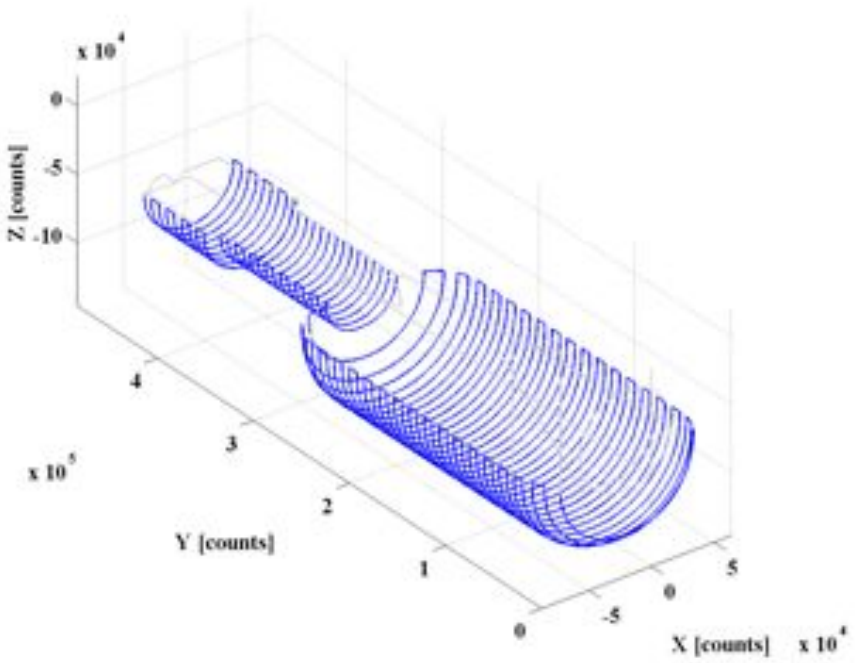


Figure 6-11 Trajectory of the Machine Tool for the Bottle Test Case

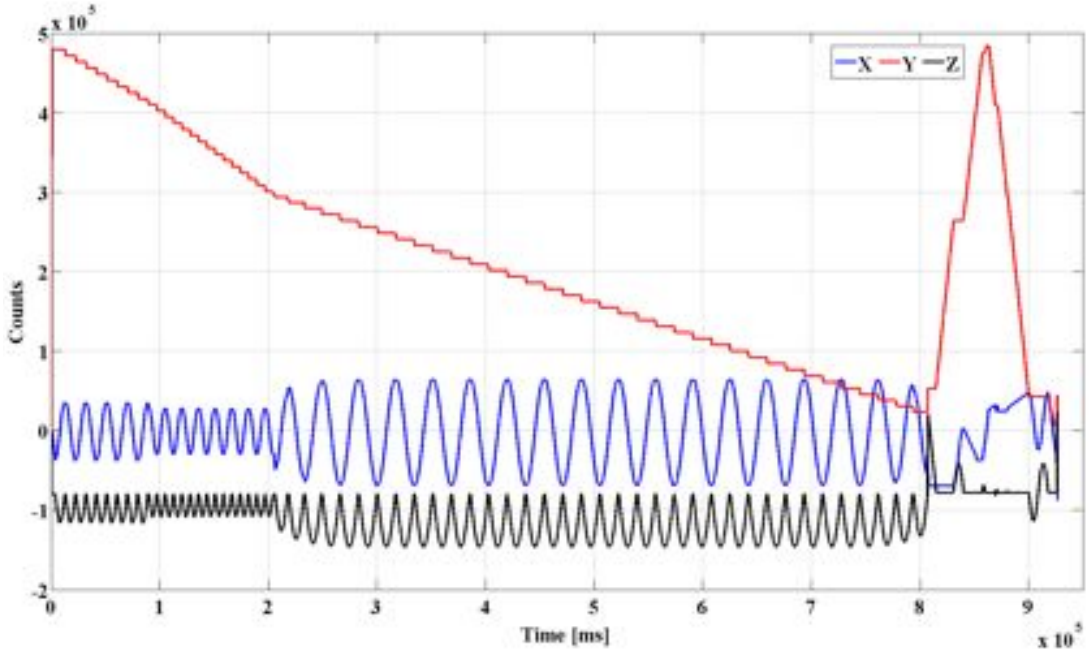


Figure 6-12 X, Y, and Z Axis Trajectories of the Bottle Test Case

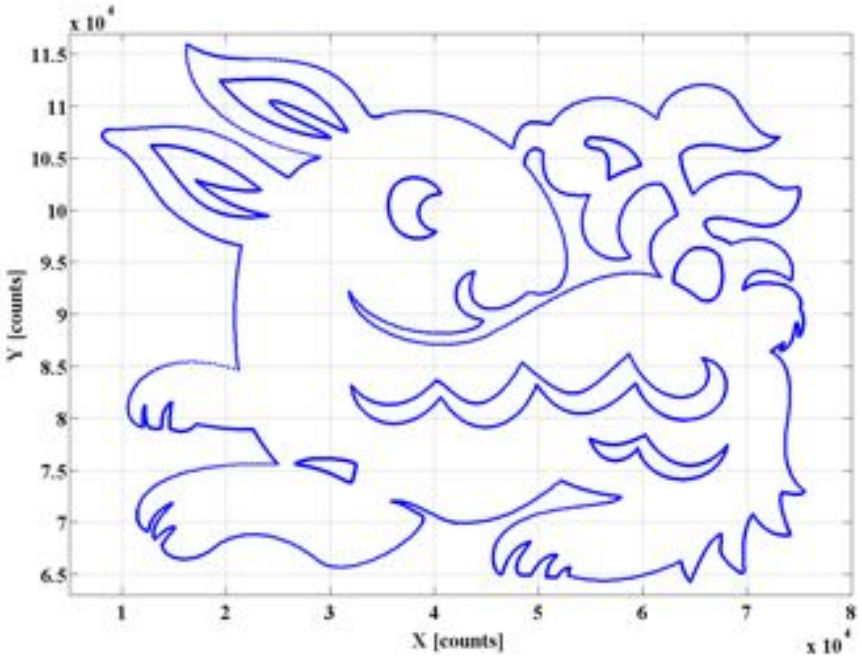


Figure 6-13 Trajectory of the Machine Tool for the Rabbit Test Case

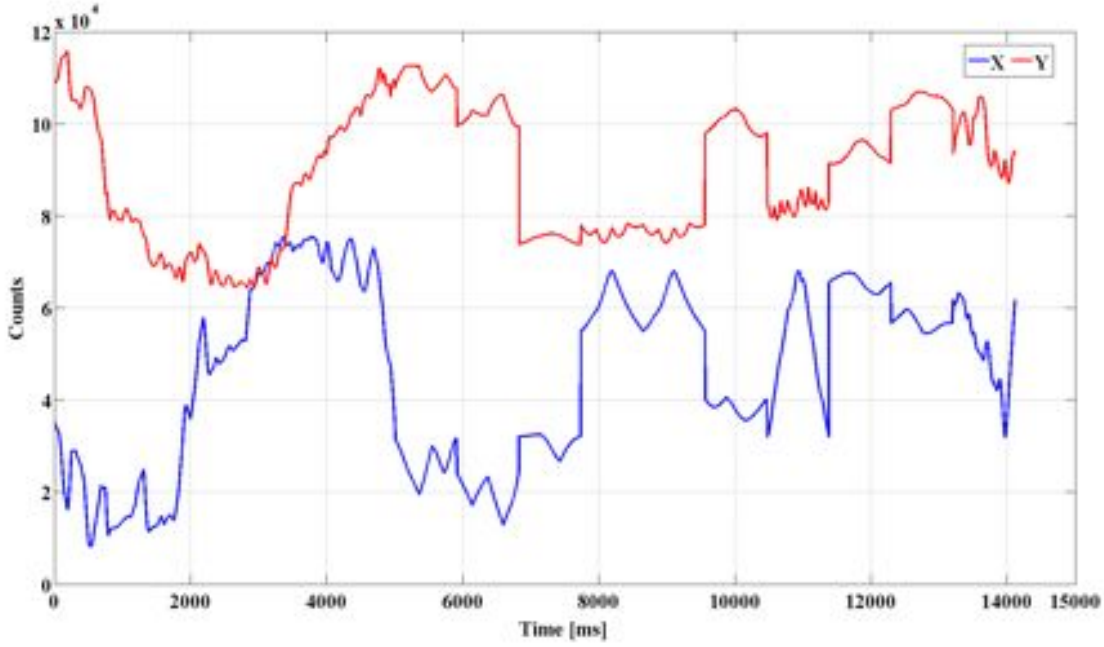


Figure 6-14 X and Y Trajectories of the Rabbit Test Case

Table 6-1 Attributes of the Test Cases

Case	Bottle	Rabbit
Number of Axes	3	2
Position Resolution [counts/mm]	10000	1000
Samples / Axis	926642	14120
Sampling Period [s]	0.001	0.001
Command Duration [s]	926.642	14.120
Range of Data [Byte]	3	3
Range of Data (∇) [Byte]	2	2
Range of Data (∇^2) [Byte]	2	3
Total Size of Data [kB]	8145	83
Average Compression Ratio [%]	0.18	17.17
Average Generation Time / Command [ms]	0.29	0.52
Average Generation Time / Command (NIOS II) [ms]	32	48

average generation time for the original commands since the generation of zero sequences takes less time when compared to the generation of regular points. Furthermore, due to the same reasons the average compression ratio for the Bottle is too low.

The comparison of the generated commands and the original trajectory are not given in this chapter since there is no difference between them. The designed command generator produces the original commands without any error. On the other hand, the effect of the speed on the generated commands is further elaborated in the study of Yaman and Dolen [66]. Due to the interpolation, there may be representation errors less than 20 encoder counts.

Table 6-2 shows the utilization of FPGA resources in terms of numbers and percentages (with respect to Altera Cyclone II 2C20 FPGA Chip) for different number of axis applications. In the last application of the FPGA implementations, the command generation algorithm is written in C programming language and the resulting are cross-compiled to run on a softcore processor (NIOS II) deployed on the FPGA. Then the resulted code is downloaded to the designed processor. The

pseudocode of the decompression algorithm is provided in Table 6-3. Given the compressed motion command data and the direction, the algorithm generates the differenced motion trajectory. After the integration is accomplished in an upcoming unit, the original trajectory is generated via interpolating according to the speed of generation.

Table 6-2 FPGA Resources Used

Resources	X	X & Y	X, Y & Z	NIOS II (X)
Total Logic Elements	6144 (33%)	10984 (59%)	15267 (82%)	2433 (13%)
Total Combinational Functions	5684 (30%)	10989 (58%)	16294 (86%)	2225 (12%)
Dedicated Logic Registers	2134 (11%)	3881 (20%)	5626 (29%)	1463 (8%)
Embedded Multipliers	4 (8%)	8 (16%)	12 (24%)	0
Total Pins	85 (27%)	131 (42%)	157 (50%)	85 (27%)
Total PLLs	1 (25%)	1 (25%)	1 (25%)	1 (25%)

It can be inferred from Table 6-2 that as the number of axis included increases, the resources reserved for the implementation also increase. Since only one PLL is used for the SDRAM on the development board, its percentage remains the same. This is due to the fact that the pins connected to SDRAM are independent of the number of axis. There exists only one Memory Interface in each design. When the overall percentage of the resources utilized is considered, even a low-end FPGA chip is adequate to implement the proposed command generation paradigm. On

Table 6-3 Pseudocode for the Decompression of ΔY10

```
Inputs: Compressed Motion Command Data, Its Statistical Attributes and Direction
Outputs: Original Motion Commands
Let Φ denote the number of bits in the length field
i ← 0, counter ← 0, amplitude ← 0, signbit ← 0
while i ≤ Φ do
    if the consecutive bits in the length field are different then
        assign the length of the command to counter
        determine amplitude of the command
        if (amplitude == 0) and (counter > 1) then
            generate zero sequence
        end
        if amplitude ≠ 0 then
            fetch signbit
            if signbit = 0
                output amplitude
            else
                output −amplitude
            end if
        end
    else
        counter + +
    end if
    if direction == 0
        i + +
    else
        i − −
    end if
    counter ← 0, amplitude ← 0
end while
```

the other hand, the softcore approach utilizes less FPGA resources compared to the hardwired approach. It also does not employ any embedded multipliers in the implementation. Provided that the on board memory chips are used for the memory of the softcore, the hardware resources will remain constant when the number of axis is increased. Although it seems that the softcore approach has a solid advantage on the hardwired approach in terms of the resources, it is about

100 times slower than the hardwired implementation as seen in Table 6-1. This is the main disadvantage of the softcore processors provided by the FPGA vendors as Lysecky and Vahid [101] stated in their study. They proposed a hybrid approach to overcome the disadvantages of the softcores by transforming some of the critical regions to hardwired circuitry.

6.5 Conclusion

A motion command generation system capable of generating commands at variable feedrates for motion control systems is implemented on an FPGA development board via two different approaches and tested on two different test cases having different attributes. With the designed hardware, the trajectories for any robotics or CNC application can be generated directly without using any intermediate programming files provided that the encoded data is transferred to the memory of the hardware.

The encoding scheme of the realized command generation paradigm has two main phases. In the first phase, higher order differences of the original command sequence are calculated. Then in the second phase the differenced data sequence is compressed with ΔY10 compression algorithm. Afterwards the compressed commands are decoded and the commands are generated via interpolation according to the current value of the feedrate and fed to the motion control system. The main points and contributions of the chapter can be summarized as follows:

- The employed compression algorithm is developed to compress the integer command sequences. A detailed analysis on the performance of ΔY10 compression algorithm can be viewed in the study of Yaman and Dolen [66]. Along with its compression performance, it is also decoded faster than the conventional methods such as Huffman, LZW and Arithmetic Coding. This is due to the fact that the compression scheme does not

employ any dictionary and it is not necessary to scan all the compressed data to generate the motion commands.

- The developed FPGA design is independent of the trajectories. Provided that the number of integrator units in the hardware design is equal to the order of difference in the encoding session, the user only needs to store the compressed data to SDRAM and then start the system. If there is one less integrator unit on the hardware, then instead of position commands velocity profile is generated.
- Since the design utilizes successive number of integrators, the velocity and acceleration commands may also be generated along with position commands. These higher order trajectories may be further used in advanced motion controller topologies.
- After the hardware implementation of this novel command generation paradigm is realized, it may be integrated to the motion control units of printer equipments, textile machinery, industrial robots and different kinds of manufacturing machines.
- Although two different approaches are utilized to implement the command generator, the hardwired approach is elaborated in a detailed manner in the chapter since the command generation speed is becoming more important with the improvements in the manufacturing and robotics industries.
- In future studies, it is planned to integrate a curve offset generation algorithm to the current command generation paradigm and realize it with a more powerful FPGA chip. Thus, the motion trajectories of pocketing operations and 3D printers can be generated more efficiently.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

In this dissertation, various command generation paradigms for different CNC and robotics applications are proposed along with the necessary algorithms to generate trajectories. The thesis starts with discussing the motivation behind the study. After sufficient reasons are stated in the first chapter, the literature behind the fields of command generation is reviewed in the following chapter. Before the introduction of the proposed paradigms, the third chapter focuses on one of the mostly used aspects of the command generation systems, which is curve offset generation. In this chapter, different algorithms are proposed and evaluated in terms of their time and memory complexities. The first variations of the command generation paradigms proposed in the fourth chapter do not utilize the curve offset generation methods. They do employ various data compression algorithms to decrease the memory requirements of the tool trajectories. The compressed motion data are then decoded on the machine side and fed to the motion control units of the CNC machinery. Another command generation paradigm is proposed in the preceding chapter. The paradigm takes advantages of the symmetric properties of the machining parts and utilized the curve offset generation algorithms presented in the third chapter. With the paradigm, the tool trajectories are defined by the given commands of the method on a program. The compiled program is then installed on the hardware and the commands are generated within the machine. A sample hardware implementation of the command generation

paradigm proposed in the fourth chapter is explained in the upcoming chapter. The finite state machines utilized in the FPGA implementation are explained and the utilized hardware resources are discussed in the chapter. The main contributions of the studies discussed in the chapter can be summarized as follows.

In the first section of the second chapter, curve offset generation algorithms are classified into four different fields based on image processing (morphological operations), NURBS, polynomial approximations, and Voronoi diagrams. The ones proposed in this dissertation do depend on the morphological operations. The memory requirement problem of these techniques is surmounted by utilizing only the relevant data. Then in the second section, the state of the art command generation systems for CNC machinery are discussed. The hardware implementations of various compression algorithms are evaluated in the following section. The command generation paradigm proposed in the fourth chapter of the thesis do also utilize a novel compression algorithm and its sample FPGA implementation is evaluated in the sixth chapter. Before speculating on the possible further research fields, command generation systems based on FPGA are evaluated.

The third chapter of the dissertation focuses on the curve offset generation methods. Five different methods are proposed in the chapter. Four of these methods are using morphological operations to obtain the offsets of the given trajectories while the last one utilizes polygon operations in the computations. The first four methods are firmly linked to each other. The primitive one (MOBI) is employed on the binary images. Thus, it requires more memory than its evolved versions. The ones (MOBS, IMOBS, and AMOBS) proposed after the MOBI do not use binary images, rather they work with the boundary sets of the curves. With the utilization of the boundary data set, the memory requirement problem of the MOBI is solved, but the time complexity of the MOBS is still quadratic. Then with the introduction of IMOBS, the time complexity of the method is reduced to linear-in-time. This is accomplished by employing a grid search algorithm on the MOBS. The last method (AMOBS) based on morphological operations further

improves the performance of IMOBS by not generating the possible boundary points that are to be eliminated in the future steps of the algorithm. Due to the linear-in-time properties of the last two methods, they can be utilized in the hardware implementation of the command generation paradigm proposed in the fifth chapter of the thesis.

The first type of the command generation paradigms is proposed in the fourth chapter. The paradigms depend on compressing the original trajectories via various compression approaches in advance and generating them on the embedded hardware of the CNC machinery. Before compressing the trajectories, their higher order differences are taken in order to decrease the range of the original commands. This differencing process results in the generation of acceleration, velocity and position trajectories at the same time provided that the order of difference is at least two. The proposed paradigms are evaluated according to the compression performances on different test cases. The proposed compression algorithm ΔY10 does better than the other conventional compression methods in general. This command generation paradigm is also realized on an FPGA development board and its details are explained in the sixth chapter.

The second type of the command generation paradigm depends on the contextual modeling of the tool trajectories. The paradigm, named as VEPRO, utilizes the developed algorithms (curve offset generation and data compression) in the previous two chapters. Thus, it can be considered as the most advanced command generation paradigm of the thesis. In the VEPRO, the command trajectories of the tools or the machining heads are defined via the proposed commands. One of the main advantages of the written VEPRO programs is that by only changing the given base curves and the offset tables one may obtain totally different trajectories. For instance, in the fifth chapter three different trajectories are generated with only one VEPRO program. Furthermore, with the addition of few VEPRO commands the mirror image of the trajectories may be obtained as discussed in the chapter. When ΔY10 compression algorithm is also used to compress the base curves of the trajectories, original trajectories of 2.5D pocketing operations can be compressed to 0.4 %. If the compression algorithms

are not utilized, then the compression ratio becomes 4.2 % for the same test case. Thus, for higher compression rates it is better to compress base curve before loading them onto the memory of the VEPRO hardware.

The last chapter before the conclusion discusses the details of the sample hardware implementation of the paradigm proposed in the fourth chapter of the dissertation. With the proposed hardware architecture, any trajectories of robotics or CNC applications can be generated without the need of an intermediate programming file. The developed FPGA structure is independent of the trajectories. If the necessary information is supplied in the header part of the compressed data set, the trajectories can be generated without any error. One of the main advantages of the implementation and the paradigm is that since there are at least two integrator units in the design, the velocity and the acceleration trajectories can also be transferred to the control units along with the position sequences. Although the implementation is carried out in two different ways (hardwired and softcore approaches), it is suggested to use hardwired approach due to its faster command generation capability.

To summarize, the developed command generation paradigms in the dissertation can be embedded into the control units of the robotics and CNC applications in order to make use of their advantages over the conventional systems.

7.2 Future Work

Addition to the studies completed within the scope of the dissertation, there are still some major points that can further be investigated. These topics can be categorized as the improvement of the CCO part of the curve offset generation algorithms, realization of the command generation paradigm (based on ΔY10) on a motion simulator, developing post processors for CAM software aimed to generated VEPRO programs, hardware implementation, the test of the VEPRO on a real test case, and generation of 3D offsets.

The current CCO algorithms of the curve offset generation algorithms (MOBS, IMOBS, and AMOBS) do employ the nearest neighbor technique to connect the offset boundaries without considering the overall distributions of the points. This approach results in a number of segmented curve offsets. With a better approach, these problems of the current CCO can be eliminated.

Another future study can be the realization of the command generation paradigm based on the ΔY10 compression algorithm. For this approach, a motion simulator is planned to be used, but any CNC machinery or robotics application can also be utilized. When the realization of the paradigm is accomplished, the disadvantages and the deficiencies of the paradigm will undoubtedly be observed.

The VEPRO programs can be generated within the CAD and/or CAM software in order to decrease the time spent for developing the programs and decrease the possible errors in the programs. This can be accomplished by writing post processors for the corresponding software. After the user defines the details of the machining operations, the developed post processor generates the VEPRO program.

One of the obvious future studies can be the hardware implementation of the VEPRO command generation paradigm since it is not studied within the scope of the dissertation as opposed to the paradigm based on ΔY10 compression algorithm. The implementation of the VEPRO will be obviously more difficult than the implementation of the ΔY10 since the written and compiled VEPRO programs are to be processed on this hardware. After the hardware is realized, the system is planned to be tested on a desktop CNC milling machine. Beside classical machining operations, 3D printing operations can also be tested by simply changing the tool with an extruder.

The curve offset generation algorithms developed in the dissertation are only applicable for 2D trajectories. On the other hand, 3D offsets are necessary for different kinds of CNC and robotics operations. For the VEPRO to be valid in these areas, an algorithm generating 3D offsets should be included into the

context of VEPRO. It should have linear time and memory complexities as in the 2D algorithms.

REFERENCES

[1] Cheng CW, Tsai MC, Cheng MY. Real-time variable feedrate parametric interpolator for CNC machining. 15th IFAC World Congress 2002.

[2] Piegl L, Tiller W.The NURBS Book. Second ed.. Springer New York; 1997.

[3] Kramer, TR. Evaluating manufacturing machine control language standards: an implementer's view.Proceedings of the 2007 Workshop on Performance Metrics for Intelligent Systems. 2007;267-274.

[4] Shahabi HH, and Ratnam MM. On-line monitoring of tool wear in turning operation in the presence of tool misalignment. The International Journal of Advanced Manufacturing Technology 2007;38:718-727.

[5] Jimeno AM, Macía F, and García-Chamizo J. Trajectory-Based Morphological Operators: A Morphological Model for Tool Path Computation. Proceedings of the International Conference on Algorithmic Mathematics and Computer Science 2004.

[6] Molina-Carmona R, Jimeno-Morenilla AM, and Rizo R. Morphological offset computing for contour pocketing. Journal of Manufacturing Science and Engineering 2007;129:400–406.

[7] Jimeno-Morenilla A, Lopez V, Espi R, and Cuenca S. A morphological-based method for inverse offset generation. An application for surface reconstruction using mechanical digitizers. The International Journal of Manufacturing Technology 2011;54:1067-1076.

[8] Yingjie Z, and Liling G. Image-Based Approach to Generation of Offset Curves from Point Cloud. IEEE 3rd International Conference on Bioinformatics and Biomedical Engineering 2009.

[9] Chamberlain PB. Discrete Algorithms for Machining and Rapid Prototyping Based on Image Processing, Ph.D. Dissertation, University of Utah, Salt Lake City, UT, 2004.

[10] Piegl LA, and Tiller W. Computing offsets of NURBS curves and surfaces. Computer-Aided Design 1999; 31:147-156.

[11] Elber G, Kwon LI, and Kim MS. Comparing offset curve approximation methods. Computer Graphics and Applications 1997;17:62-71.

[12] Li YM, and Hsu VY. Curve offsetting based on Legendre series. Computer Aided Geometric Design 1998;15:711-720.

[13] Held M. Voronoi diagrams and offset curves of curvilinear polygons. Computer-Aided Design 1998;30:287-300.

[14] Danielsson PE. Converting a curve to right angled increments. Nord. Tidsk. Infomationsbehandling (BIT) 1963;3: 213-221.

[15] Koren Y. Interpolator for a computer numerical control system. IEEE Trans. on Computers 1970;C-19:32-37.

[16] Liang H, Hong H, Svoboda J. A combined 3D linear and circular interpolation technique for multi-axis CNC machining. J. Manuf. Sci. Eng 2002;124:305-312.

[17] Vickers GW, Bradley C. Curved surface machining through circular arc interpolation.Comput. lndust. 1992;19:329-337.

[18] Huang JT, Yang DCH. A generalized interpolator for command generation of parametric curves in computer controlled machine.ASME Flexible Automation1992;1: 393-399.

[19] Shpitalni M, Koren Y,Lo CC. Real time Curve Interpolators. Computer-Aided Design1994;832–838.

[20] Yang DCH, Kong T.Parametric interpolator versus linear interpolator for precision CNC machining.Computer-Aided Design 1994:225–234.

[21] GE Fanuc Automation. Operator's Manual Series 16i/160i/18i/180i 2002.

[22] Siemens AG. Operator's Manual Sinumerik 840D/840Di/810D 2002.

[23] Heng M, Erkorkmaz K, Design of a NURBS interpolator with minimal feed fluctuation and continuous feed modulation capability. International Journal of Machine Tools and Manufacture 2010;50:281–293.

[24] Cheng CW, Tsai MC, Maciejowski J. Feedrate control for non-uniform rational B-spline motion command generation. Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture 2006;220:1855-1861.

[25] Xu HY, Tam HY, Zhou Z, Tse PW. Variable feedrate CNC interpolation for planar implicit curves. Advanced Manufacturing Technology 2001;18:794 – 800.

[26] Rutkowski L, Przybyl A, Cpalka K. Novel Online Speed Profile Generation for Industrial Machine Tool Based on Flexible Neuro-Fuzzy Approximation. IEEE Transactions on Industrial Electronics 2012;59(2): 1238−1247.

[27] Salomon D, Motta G. Handbook of Data Compression. Fifth ed.. Springer-Verlag; 2010.

[28] Yaman U,Mutlu BR, Dolen M, Koku AB. Direct command generation methods for servo-motor drives.Proc. of the 12th International Conference on Electrical Machines and Systems 2009.

[29] Yaman U, Dolen M, Koku AB. A novel command generation method with variable feedrate utilizing FGPA for motor drives.Proc. of the 8th IEEE Workshop on Intelligent Solutions in Embedded Systems 2010:67-72.

[30] Kroger T, Wahl FM. Online trajectory generation: Basic concepts for instantaneous reactions to unforeseen events. IEEE Transactions on Robotics 2010;26(1):94-111.

[31] Haschke R, Weitnauer E, Ritter H. On-line planning of time-optimal, jerk-limited trajectories. IEEE/RSJ International Conference on Intelligent Robots and Systems 2008:3248-3253.

[32] Yong T, Narayanaswami R. A parametric interpolator with confined chord errors, acceleration and deceleration for NC machining. Computer-Aided Design 2003;35(13):1249-1259.

[33] Guarino Lo Bianco C, Ghilardelli F. A Discrete-Time Filter for the Generation of Signals with Asymmetric and Variable Bounds on Velocity, Acceleration, and Jerk. IEEE Transactions on Industrial Electronics 2013;61(8):4115:4125.

[34] Cover TM, Thomas JA. Elements of Information Theory, Second ed., John Wiley & Sons, Inc. 2006.

[35] Dickson K. Cisco IOS Data Compression. Cisco Syst. San Jose; 2000.

[36] Rigler S, Bishop W, Kennings A. FPGA-Based lossless data compression using Huffman and LZ77 algorithms. Canadian Conf. on Electrical and Computer Engineering 2007;1235-1238.

[37] De Araujo TMU, Pinto ER, De Lima JAG, Batista LV. An FPGA implementation of a microprogrammable controller to perform lossless data compression based on the Huffman algorithm. 13th IBERCHIP Workshop 2007.

[38] Abd El Ghany MA, Salama AE, Khalil AH. Design and implementation of FPGA-based systolic array for LZ data compression. IEEE Int. Symposium on Circuits and Systems 2007;3691-3695.

[39] Cui W. New LZW data compression algorithm and its FPGA implementation. Picture Coding Symposium 2007.

[40] H'ng GH, Salleh MFM, Halim ZA. Golomb coding implementation in FPGA. Elektrika Journal of Electrical Engineering 2008:36-40.

[41] Koch D, Beckhoff C, Teich J. Hardware decompression techniques for FPGA-based embedded systems. ACM Transactions on Reconfigurable Technology and Systems 2009;2(2): 1−23.

[42] Lin MB, Lee JF, Jan GE. A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 2006;14(9): 925−936.

[43] Lee T, Park J. Design and implementation of static Huffman encoding hardware using a parallel shifting algorithm. IEEE Transactions on Nuclear Science 2004;51(5): 2073−2080.

[44] Yongming Y, Jungang L, Jianmin W. LADT arithmetic improved and hardware implemented for FPGA - Based ECG data compression. Proceedings of 2nd IEEE Conference on Industrial Electronics and Applications 2007; 2230-2234.

[45] Valencia D, Plaza A. FPGA-Based hyperspectral data compression using spectral unmixing and the pixel purity index algorithm. Computational Science 2006;881- 891.

[46] Ouyang J, Feng P, Kang J. Fast Compression of Huge DNA Sequence Data. In: 5th International Conference on BioMedical Engineering and Informatics, 2012:885−888.

[47] Monmasson E, Cirstea MN. FPGA Design Methodology for Industrial Control Systems — A Review. IEEE Transactions on Industrial Electronics 2007; 54(4): 1824−1842.

[48] Kim S, Jeong WS, Ro WW, Gaudiot JL. Design and evaluation of random linear network coding Accelerators on FPGAs. ACM Transactions on Embedded Computing Systems (TECS) 2013;13(1): 13.

[49] Cho JU, Le QN, Jeon JW. An FPGA-Based Multiple-Axis Motion Control Chip. IEEE Transactions on Industrial Electronics 2009;56(3): 856−870.

[50] Su KH, Hu CK, Cheng MY. Design and Implementation of an FPGA-based Motion Command Generation Chip. In: IEEE International Conference on Systems, Man and Cybernetics, 2006:5030−5035.

[51] Jeon JW, Kim YK. FPGA based acceleration and deceleration circuit for industrial robots and CNC machine tools. Mechatronics 2002;12: 635−642.

[52] Osornio-Rios RA, Romero-Troncoso RJ, Herrera-Ruiz G, Castañeda-Miranda R. FPGA implementation of higher degree polynomial acceleration profiles for peak jerk reduction in servomotors. Robotics and Computer-Integrated Manufacturing 2009; 25(2): 379−392.

[53] Kim HC, and Yang MY. An optimum 2.5 D contour parallel tool path. International Journal of Precision Engineering and Manufacturing 2006;8:16-20.

[54] Kulkarni P, Marsan A, and Dutta D. A review of process planning techniques in layered manufacturing. Rapid Prototyping Journal 2000;6:18-35.

[55] Maekawa T. An overview of offset curves and surfaces. Computer-Aided Design 1999;31:165-173

[56] Farouki RT, and Shah S. Real-time interpolators for Pythagorean-hodograph curves. Computer Aided Geometric Design 1996;13:583-600.

[57] Pottmann H. Rational curves and surfaces with rational offsets.Computer Aided Geometric Design 1995;12:175-192.

[58] Elber G, and Cohen E. Error bounded variable distance offset operator for free form curves and surfaces. International Journal of Computational Geometry and Applications 1991;1:67-78.

[59] Lee IK, Kim MS, and Elber G. Planar curve offset based on circle approximation. Computer Aided Design 1996;28:617-630.

[60] Maekawa T, and Patrikalakis NM. Computation of singularities and intersections of offsets of planar curves. Computer Aided Geometric Design 1993;10:407-429.

[61] Maekawa T. Self-intersections of offsets of quadratic surfaces: Part II, implicit surfaces. Engineering with Computers 1998;14:14-22.

[62] Patrikalakis NM, and Bardis L. Offsets of curves on rational B-spline surfaces. Engineering with Computers 1989;5:39-46.

[63] Rausch T, Wolter FE, and Sniehotta O. Computation of medial curves onsurfaces. The Mathematics of Surfaces 1997;7:43-68.

[64] Pottmann H. General offset surfaces. Neural, Parallel and Scientific Computations 1997;5:55-80.

[65] Brechner EL. General tool offset curves and surfaces. Geometry processing for design and manufacturing 1992:101-21.

[66] Yaman U, and Dolen M. Direct command generation for CNC machinery based on data compression techniques. Robotics and Computer-Integrated Manufacturing 2013;29:344-356.

[67] Kim HC. Tool path generation for contour parallel milling with incomplete mesh model. The International Journal of Advanced Manufacturing Technology 2010;48:443-454.

[68] Sun Y, Ren F, Zhu X, and Guo D. Contour-parallel offset machining for trimmed surfaces based on conformal mapping with free boundary. The International Journal of Advanced Manufacturing Technology 2012;60:261-271.

[69] Serra J. Image Analysis and Mathematical Morphology 1982. Academic Press, London.

[70] Chia TL, Wang KB, Chen LR, and Chen Z. A parallel algorithm for generating chain code of objects in binary images. Information Sciences 2003;149:219-234.

[71] Chang F, Chen CJ, and Lu CJ. A linear-time component-labeling algorithm using contour tracing technique. Computer Vision and Image Understanding 2004;93:206-220.

[72] Meer P, Sher CA, and Rosenfeld A. The chain pyramid: hierarchical contour processing. IEEE Transactions on Pattern Analysis and Machine Intelligence 1990;12:363-376.

[73] Matoušek J. Geometric range searching. ACM Computing Surveys 1994;26:422-461.

[74] Agarwal PK, and Erickson J. Geometric range searching and its relatives. Contemporary Mathematics 2009;223:1-56.

[75] Liang YD, and Barsky BA. An analysis and algorithm for polygon clipping. Communications of the ACM 1983;26:868-877.

[76] Rivero M, and Feito FR. Boolean operations on general planar polygons. Computers & Graphics 2000;24:881-896.

[77] Peng Y, Yong JH, Dong WM, Zhang H, and Sun JG. A new algorithm for Boolean operations on general polygons. Computers & Graphics 2005;29:57-70.

[78] Martínez JG, Rueda AJ, and Feito FR. A new algorithm for computing Boolean operations on polygons. Computers & Geosciences 2009;35:1177-1185.

[79] Liu YK, Wang XQ, Bao SZ, Gomboši M, and Žalik B. An algorithm for polygon clipping, and for determining polygon intersections and unions. Computers & Geosciences 2007;33:589-598.

[80] Greiner G, and Hormann K. Efficient clipping of arbitrary polygons. ACM Transactions on Graphics 1998;17:71-83.

[81] Murta A (2012) GPC – General Polygon Clipper Library. http://www.cs.man.ac.uk/~toby/gpc/. Accessed 20 July 2013.

[82] Vatti BR. A generic solution to polygon clipping. Communications of the ACM 1992;35:56-63.

[83] Bulbul R, and Frank AU. Intersection of Non-convex Polygons using the Alternate Hierarchical Decomposition. Geospatial Thinking 2010:1-23, Spinger-Verlag, Berlin.

[84] Liu XZ, Yong JH, Zheng GQ, and Sun JG. An offset algorithm for polyline curves. Computers in Industry 2007;58:240-254.

[85] Zhiwei L, JianzhongF, Yong H, and Wenfeng G. A robust 2D point-sequence curve offset algorithm with multiple islands for contour-parallel tool path. Computer Aided Design 2013;45:657-670.

[86] Electronic Industries Association. EIA Standard EIA-274-D Interchangeable Variable Block Data Format for Positioning, Contouring, and Contouring/Positioning Numerically Controlled Machines. EIA 1979.

[87] Numerical Control BCL Standards Association. NCBSA Standard Proposal for EIA 494 C, Basic Control Language (BCL) An ASCII Data Exchange Specification for Computer Numerical Control Manufacturing. Numerical Control BCL Standards Association 1996.

[88] ISO 10303-238. Industrial automation systems and integration - Product data representation and exchange - Part 238: Application protocol: Application

interpreted model for computerized numerical controllers. Geneva: International Organization for Standardization 2007.

[89] International Consortium for Advanced Manufacturing. Dimensional Measuring Interface Standard Part I, Revision 05.0.2004.

[90] Lorenz RD. Robotics and Automation Applications of Drives and Converters. Proceedings of the IEEE 2002;89:6:951-962.

[91] Huffman D. A method for the construction of minimum redundancy codes.Proc. of the IRE 1952;1098–1101.

[92] Moffat A, Radford N, Witten IH. Arithmetic coding revisited. ACM Transactions on Information Systems 1998;16:3:256–294.

[93] Welch TA. A technique for high-performance data compression.IEEE Computer 1984;17:6:8–19.

[94] Jamro MWE, K Wiatr. FPGA implementation of the dynamic Huffman encoder. Proceedings of IFAC Workshop on Programmable Devices and Embedded Systems 2006.

[95] Adluri R, Babu SG, Daniel P. FPGA Implementation of Multi-alphabet Arithmetic Coding Using Rotating Intervals. IJCA Special Issue on Electronics, Information and Communication Engineering ICEICE(2) 2011: 10-14.

[96] Naqvi S, Naqvi R, Riaz R, Siddiqui F. Optimized RTL design and implementation of LZW algorithm for high bandwidth applications. Przegląd Elektrotechniczny (Electrical Review) 2011: 279-285.

[97] Omirou SL. A locus tracing algorithm for cutter offsetting in CNC machining. Robotics and Computer-Integrated Manufacturing 2004;20:49-55.

[98] NIOS II Performance Benchmarks, Altera Co., 2013.

[99] Lin Z, Jianzhong F, Yong H, Wenfeng G. A robust 2D point-sequence curve offset algorithm with multiple islands for contour-parallel tool path. Computer-Aided Design 2013;45: 657-670.

[100] Kolmogorov AN. Three approaches to the quantitative definition of information. Problems of information transmission 1965;1(1):1-7.

[101] Lysecky R, Vahid F. Design and implementation of a MicroBlaze-based warp processor. ACM Transactions on Embedded Computing Systems (TECS) 2009;8(3): 22.

APPENDICES A : NIOS II C CODE

```
#include <stdio.h>
#include <stdlib.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "sys/alt_timestamp.h"
#include "alt_types.h"

#define uint8 unsigned char
#define uint16 unsigned short
#define uint32 unsigned long
#define int8 char
#define int16 short
#define int32 long

int main()
{
        alt_timestamp_start();
       alt_u32 time1,time2;
       uint8 sign_ = 0; uint8 amp_ = 0; uint8 term_ = 0; uint8 zero_ = 0; uint32
comlength;
      int n; n = IORD_32DIRECT(0x02000000,0) >> 28;
      comlength = 61440 & IORD_32DIRECT(0x02000000,0);
       uint32 original[comlength];
       float original_we[comlength];
       int32 l1 = comlength - n;       //Length of the Sign Field
       int32 l2 = IORD_32DIRECT(0x02000000,1)*4; //Length of Amplitude Field
       int32 count = 0;                //Counts the length of the amplitude value.
       int32 is = 0; int32 i1 = 0; int32 iz = 0; int32 j = 0; int k = 0; int32 k1
= 0;
       int32 kz = 0; int32 kzz = 0; int32 m1 = 0; int32 r = 0; int32 rs = 0; int32
r1 = 0;
       int32 rz = 0; int32 l = 0; int32 zeroSeq = 0; int32 s = 0;
       const int
twos[16]={1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768};
       uint8 term[1000]; uint8 amp[1000]; uint8 sign[1000]; uint8 zero[100];

       int i; int i_start; int i_end;
      i_start = (n+3)*6;  i_end = i_start + IORD_32DIRECT(0x02000000, 1);
       for (i=i_start; i<i_end; i++) { term[i-i_start] =
IORD_8DIRECT(0x02000000,i);}
      i_start = i_end + 2;  i_end = i_start + IORD_32DIRECT(0x02000000, 1);
       for (i=i_start; i<i_end; i++) {amp[i-i_start] =
IORD_8DIRECT(0x02000000,i);}
      i_start = i_end + 2;  i_end = 4*IORD_32DIRECT(0x02000000, 2)-1;
       for (i=i_start; i<i_end; i++) {sign[i-i_start] =
IORD_8DIRECT(0x02000000,i);}
      i_start = i_end + 2;  i_end = i_start + 100;
       for (i=i_start; i<i_end; i++) {zero[i-i_start] =
IORD_8DIRECT(0x02000000,i);}
       i = 0;
      time1 = alt_timestamp();   //Time measurement starts
       while (k < l2) {     //Loop until all bits of length field are processed
              i = k/8; r = k%8; k1 = k; i1 = i; r1 = r;   //Cursor position is
determined
              if (k == (l2-1)) {            //The last bit is processed
                     if ((((term[i]<<r)&128)==128)) {term[i]=term[i] & (254<<(7-
(r+1)));}
                     if ((((term[i]<<r) & 128)==0)) {term[i]=term[i] | (1<<(7-
(r+1)));}
```

```
                }
                term_ = term[i] << r;                 //Shifted length field byte
                if ((term_ & 128) == 128) {m1 = 0; count ++;}
                else {m1 = 128; count ++;}
                if (((r != 7) & (((term_
<<1)&128)==m1))|((r==7)&((term[i+1]&128)==m1))) {
                        for (j=0; j<count; j++) {
                                        // Amplitude Field
                                amp_ = amp[i1] << r1;
                                if ((amp_ &  128) == 128) { original[l] += twos[j];}
                                k1--; i1 = k1 / 8; r1 = k1 % 8;
                        }
                        if ((count > 1) & (original[l] == 0)) {
                                // Zero Field
                                kz = kz + count - 1; kzz = kz; iz = kzz / 8; rz =
kzz % 8;
                                for (j=0; j<count; j++) {
                                        zero_ = zero[iz] << rz;
                                        if ((zero_ &  128) == 128) { zeroSeq +=
twos[j];}
                                        kzz--; iz = kzz / 8; rz = kzz % 8;
                                }
                                for (j=0; j<zeroSeq; j++) { original[l] = 0; l++;}
                                zeroSeq = 0; kz++;
                        }
                        else {l++;}
                        count = 0;
                        if (original[l-1] != 0) {
                                    // Sign Field
                                is = s / 8; rs = s % 8;
                                sign_ = sign[is] << rs;
                                if ((sign_ & 128) == 0) { }
                                else { original[l-1]= - original[l-1] ;}
                                s++;
                        }
                }
                k++;
        }
         time2 = alt_timestamp();   //Time measurement ends
        //Prints the decoded commands on the console of the NIOS software
        for (k=0; k<l1; k++) {
                printf("%ld\n",original[k]);
        }
       //Prints the elapsed time
         printf ("Time elapsed = %u ticks\n", (unsigned int) (time2 - time1));
         printf ("Number of ticks per second = %u\n", (unsigned
int)alt_timestamp_freq());

  return 0;
}
```

APPENDICES B : MATLAB FUNCTIONS

FUNCTIONS	INPUTS	OUTPUTS	EXPLANATIONS
cogen	• Binary image • Offset radius	• Offsetted boundary	Generates curve offset for binary images
curoff	• Boundary set • Offset radius • Number points in the structuring element	• Set of offsetted curve segments • Number points in the structuring element	Employs CBS part of MOBS curve offset generation algorithm
curoff2Dot	• Boundary set • Offset radius • Number points in the structuring element	• Set of offsetted curve segments • Number points in the structuring element • Minimums and maximums of X and Y coordinates	Employs CBS part of IMOBS curve offset generation algorithm

amobs	• Boundary set • Offset radius	• Set of offsetted curve segments • Minimums and maximums of X and Y coordinates	Employs CBS part of AMOBS curve offset generation algorithm
chain	• Offsetted curve segments • Offset radius	• Connected segments • Unconnected segments	Employs CCO part of MOBS, IMOBS, and AMOBS curve offset generation algorithms
unify	• Curve segments • Offset radius	• Connected offsets	Reconnects the curve outputted with *chain* function
squash	• Offsetted curve segments	• Squashed set of curves • Indexes of removed curves	Removed the empty cells from the given cell vector
qhash	• Offsetted curve segments • Size of hash table • Minimums and maximums of X and Y coordinates	• Distribution of points in the hash table • Various properties of the hash table	Distributes the points in the offsetted curve segments to the cells in the hash table according to the coordinates of the points

discard	• Boundary set • Offset radius • Hash table • Properties of the hash table	• New hash table without invalid points	Removes the global invalid loops present in the layout
gcurve	• Boundary set • Regions • Number of points	• Resampled boundary set	Resamples the given curves at a higher rate using low pass interpolation
offsetError	• Reference curve • Offsetted curve	• Error at each offsetted point	Calculates the error of the offset points for temperature plots
dacomp	• Vector • Order of difference	• Compressed code • Dictionary • Ratio	Takes the higher order difference of the vector and then compresses the vector with Arithmetic compression algorithm
dhcomp	• Vector • Order of difference	• Compressed code • Dictionary • Ratio	Takes the higher order difference of the vector and then compresses the vector with Huffman compression algorithm

rlez_dhcomp	• Vector • Order of difference	• Compressed code • Dictionary • Content of zero table • Ratio	Takes the higher order difference of the vector and then compresses the vector with Huffman compression algorithm after employing RLEZ on the differenced data
lzwmain	• Vector • Order of difference	• Compressed code • Dictionary • Ratio	Takes the higher order difference of the vector and then compresses the vector with LZW compression algorithm
dycomp2	• Vector • Order of difference	• Compressed data structure	Takes the higher order difference of the vector and then compresses the vector with symbol DY09 compression algorithm

dydcomp2	• Compressed data structure	• Decompressed differentiated vector • Original vector	Decompresses the given data according the DY09 compression algorithm and then accumulates to get the original vector
dycomp3	• Vector • Order of difference	• Compressed data structure	Takes the higher order difference of the vector and then compresses the vector with symbol DY10 compression algorithm
dydcomp3	• Compressed data structure	• Decompressed differentiated vector • Original vector	Decompresses the given data according the DY10 compression algorithm and then accumulates to get the original vector
dymem	• Vector • Order of difference	• Required memories	Computes the memory needed to compress a given time sequence using DY09 & DY10 techniques

higherorderentr	• Vector • Order of difference • Number of commands in a set	• Sets of commands • Frequencies of the sets • Compressed data • Dictionary • Ratio	Takes the higher order difference of the vector, combines the commands and compresses the resulting data according to the Markov probability matrix
rentropy	• Vector • Order of difference	• Entropy • Matrix for occurrence frequencies • Resequenced vector	Takes the higher order difference of the vector and computes the entropy
vsint	• Vector • Feedrate vector	• Interpolated position sequence • Modified feedrate scale vector	Performs linear interpolation on a given trajectory
GCO	•	•	Employs AMOBS curve offset generation algorithm for VEPRO command generation paradigm

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Yaman, Ulas

Nationality: Turkish (TC)

Date and Place of Birth: June 15, 1984, Ankara

Marital Status: Married

Phone:+90 312 210 7240

E-mail Address: ulsymn@gmail.com

EDUCATION

Degree	Institution	Year of Graduation
MS	METU Mechanical Engineering	2010
Minor	METU Mechatronics	2007
BS	METU Mechanical Engineering	2007
High School	Ayrancı Anatolian High School	2002

WORK EXPERIENCE

Year	Place	Enrollment
2007-Present	METU Mechanical Engineering	Research Assistant

FOREIGN LANGUAGES

Fluent English, Elementary German

SELECTED PUBLICATIONS

Dolen M, and Yaman U. New morphological methods to generate two-dimensional curve offsets. The International Journal of Advanced Manufacturing Technology 2014;71(9-12):1687-1700.

Yaman U, and Dolen M. Direct command generation for CNC machinery based on data compression techniques. Robotics and Computer-Integrated Manufacturing 2013;29(2):344-356.