Available online at www.sciencedirect.com

ScienceDirect

Procedia CIRP 33 (2015) 490 – 495

www.elsevier.com/locate/procedia

9th CIRP Conference on Intelligent Computation in Manufacturing Engineering - CIRP ICME '14

Generating milling tool paths for prismatic parts using genetic programming

Jack Barclay[a,*], Vimal Dhokia[a], Aydin Nassehi[a]

[a]*Department of Mechanical Engineering, University of Bath, Bath. BA2 7AY. UK.*

[*] Corresponding author. Tel.: +44 1225 384049; *E-mail address:* j.barclay@bath.ac.uk

Abstract

The automatic generation of milling tool paths traditionally relies on applying complex tool path generation algorithms to a geometric model of the desired part. For parts with unusual geometries or intricate intersections between sculpted surfaces, manual intervention is often required when normal tool path generation methods fail to produce efficient tool paths. In this paper, a simplified model of the machining process is used to create a domain-specific language that enables tool paths to be generated and optimised through an evolutionary process - formulated, in this case, as a genetic programming system. The driving force behind the optimisation is a fitness function that promotes tool paths whose result matches the desired part geometry and favours those that reach their goal in fewer steps. Consequently, the system is not reliant on tool path generation algorithms, but instead requires a description of the desired characteristics of a good solution, which can then be used to measure and evaluate the relative performance of the candidate solutions that are generated. The performance of the system is less sensitive to different geometries of the desired part and doesn't require any additional rules to deal with changes to the initial stock (e.g. when rest roughing). The method is initially demonstrated on a number of simple test components and the genetic programming process is shown to positively influence the outcome. Further tests and extensions to the work are presented.

Keywords: Computer numerical control (CNC); Milling; Genetic programming

1. Introduction

Tool path generation is primarily thought of as a geometric problem and many different methods have been developed and iterated upon that aim to produce tool paths that exhibit particular desired attributes. For example, iso-scallop machining uses desired scallop height to influence machining strategy and high-speed milling tool paths limit cutter engagement to allow for higher feed rates and depths of cut. Deciding which method to use is not always straight forward and often has to be made at the start of the process when there isn't much information to help inform the decision. This leads to a lot of manual effort - trying different methods and modifying parameters to check their effects upon the attributes when trying to meet the specification. This research looks upon tool path generation as an optimisation problem and aims to create a system where design intent is the focus and implementation specifics are handled automatically.

A simplified model for milling machining will be used to reduce the search space of the optimisation problem. The work piece is divided into layers and each layer discretised into a square grid. Fig 1 shows how the desired product shape can be represented as a shape built up from multiple small grid squares. The cutting tool can then also be described as occupying a certain square or number of squares at any one time.

Using this model, a tool path can be defined as the order of squares visited on the grid, or the distinct movements required to take that path. Using this model, the search space in which the optimal series of cutting tool movements exist is reduced, however, it is still too large to facilitate a deterministic development of the optimal solution. This provides the motivation for exploring evolutionary computation methods within this application.

Many studies have been performed that look into the optimisation of machining parameters to improve cycle times, reduce costs, improve accuracy, and for a whole host of other objectives [1–3]. A lot of these use genetic algorithms or other evolutionary computation techniques to perform the optimisation [4–7], leading to an improved or optimal set of feed rates, cutting speeds, tool engagement angles, depths of cut and more. The actual tool path generation aspect was not considered in these studies and they relied on traditional CAM generated or manually written tool paths to accompany their efforts in optimising cutting conditions.

Some studies do look at tool path optimisation issues, however, most are limited to models that have been reduced to imitate a travelling salesman (TSP) type problem [8–10]. These

doi:10.1016/j.procir.2015.06.060

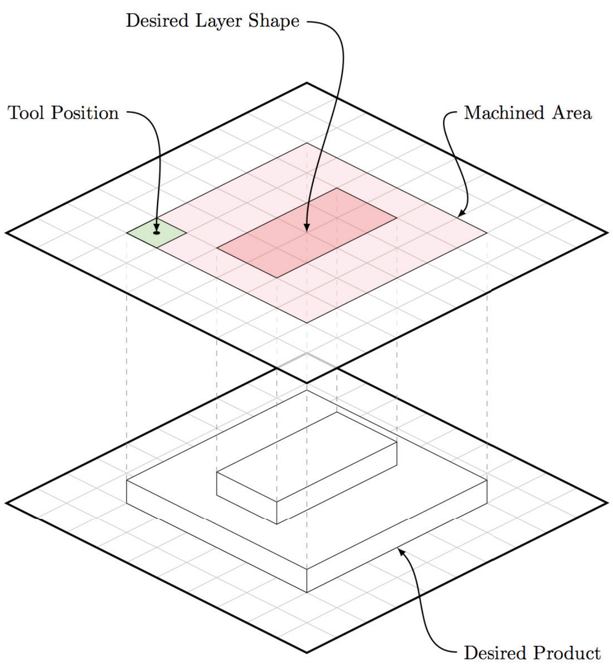

Fig. 1. The 3D work piece is simplified down into a series of 2D layers. Each layer is discretised into a grid, with the cutting area and desired shape being formed from a combination of the resultant squares.

are relevant for some machining processes such as CNC laser cutting [11] and in the optimisation of non-productive (or non-cutting) time [12], where the aim is to reduce the required tool path motion between a set of pre-defined points.

Agrawal et al. [13] used a genetic algorithm (GA) to minimise machining distance in iso-scallop machining of parametric surfaces. They were using the GA to find the globally optimal master cutting path from which the rest of the machining passes were derived. More recently, another group looked at multi-objective optimising of tool paths [14], where the trade-off between cutting force, cycle time, and scallop height was examined and pareto-optimal solutions were found and presented.

2. The Principles of Genetic Programming

An evolutionary algorithm mimics the process of Darwinian evolution through natural selection whereby ‘fitter’ individuals are more likely to pass on their genetic material to the next generation. Specific traits that contribute to the success of an individual are more likely to be present in the population as iteration through generations is continued. The process, as illustrated by Fig. 2, is followed until an individual emerges that exhibits performance above a given metric or until a maximum number of generations is reached.

Tree-based Genetic Programming (GP) represents candidate solutions with a tree structure that encodes all the logic and information about the program. The internal nodes of the tree are the functions of the program; operators that perform actions using their arguments (child nodes). The leaf nodes are called terminals and can be variables, random constants, or functions that take no arguments (0-arity functions). The function set and the terminal set are collectively known as the primitive set; they are the building blocks that make up the programs that solve the target problem.

2.1. Initial Population

Individuals are generated, by randomly selecting items from the primitive set to fill out the tree. The initial population will most likely consist of a large number of very poor solutions to the problem, as they have been randomly created and have not been through the progression and development of multiple evolution cycles.

2.2. Fitness Evaluation

Each individual in the population is evaluated and tested for their ability to solve the problem. A fitness score is given to each individual depending on the extent at which they solve the question asked of them, such that solutions can be compared and the fittest individuals identified. Fitness evaluation is very problem specific and depends upon the specifics of the application and experiment.

2.3. Parent Selection

Before a genetic operator can be applied, the parent individual(s) must be selected. Selection is fitness-proportionate and performed with replacement, meaning that an individual can be selected multiple times to be a parent. Therefore, fitter individuals are more likely to have more chances to pass their successful genetic material to the next generation, hopefully leading to progression and growth in the average fitness of the population.

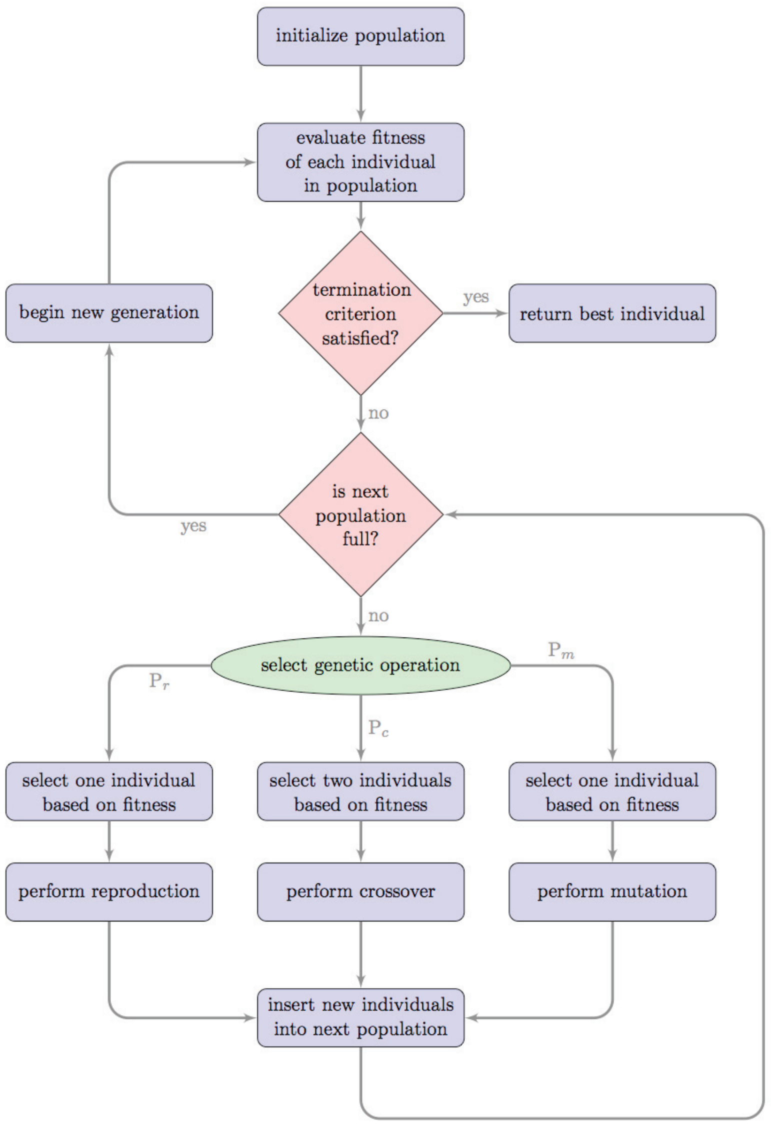
initialize population
evaluate fitness of each individual in population
termination criterion satisfied?
yes
return best individual
no
begin new generation
is next population full?
yes
no
select genetic operation
P_r
P_c
P_m
select one individual based on fitness
select two individuals based on fitness
select one individual based on fitness
perform reproduction
perform crossover
perform mutation
insert new individuals into next population

Fig. 2. Flowchart of a genetic evolutionary procedure.

2.4. Genetic Operation

There are three main types of genetic operator: reproduction, crossover, and mutation. Reproduction simply copies an individual into the next generation. Crossover selects two parent individuals and combines them to produce two new offspring. For tree-based GP, a random node in each of the parent individuals is selected, and the sub-trees below this crossover point are swapped to create two new offspring. The mutation operation makes a small change to an individuals genotype - a node is selected at random from the parent individual and the sub-tree below this point is replaced with a newly generated sub-tree. Mutation helps to introduce fresh genetic material to the population, ensuring that the variety and diversity of individuals is sustained. The type of operation used is selected proportionately according to predetermined ratios. Crossover is the most commonly applied operator; mutation is used relatively rarely.

2.5. Termination Criterion

After sufficient genetic operations to create a new population of the desired size have been completed, a new generation is initiated and the cycle of evaluation, selection, and breeding is repeated. The process is continued until either an optimal solution is found, or the prescribed maximum number of generations completed. Target fitness criteria can be set so that the cycle will terminate when a 'good enough' solution to the problem has been found.

3. A Simplified Model of the Machining Process

The resolution of a CNC machine is the smallest achievable change in the position of the tool. This reduces the positions within space that the tool can occupy to a finite number and can

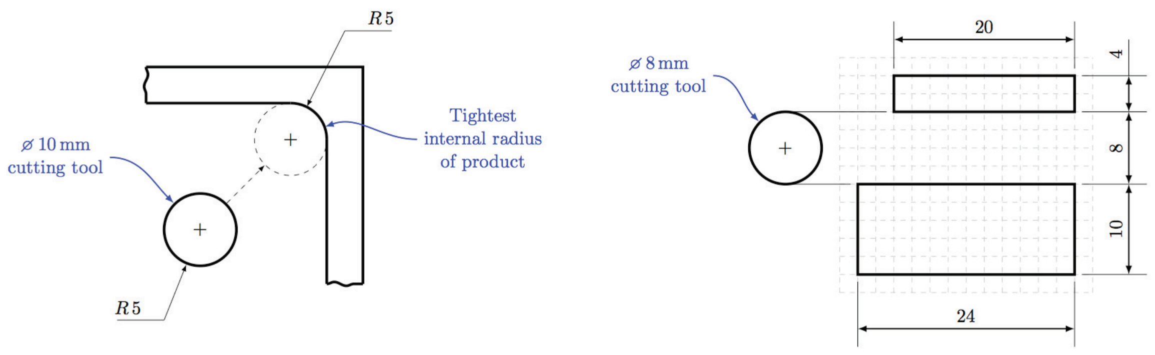

Fig. 3. The grid size is determined by the highest common factor of the dimensions within the desired shape. In this example, that calculation results in a grid size of 2mm. The cutting tool size is determined by either the tightest internal radius specified (left), or the smallest gap between two parts of the product shape (right).

be visualised as a grid of squares that the tool can move between. Resolutions of as low as 0.001-0.01 mm are common amongst modern CNC milling machines, which even at the higher end of that range equates to a grid size of 10000 x 10000 for a 100 mm x 100 mm area. Despite the reduction to a finite number of tool positions, the combination of squares when considering a routing between one side of the grid to the other is effectively infinite. In order to reduce the search space of the optimisation problem to something practical, a simplified model of this machining process is required. This is achieved by creating a model with a reduced spacial resolution - i.e., a larger grid size. Within this grid, the desired shape then can be specified by selecting which squares the tool should and should not visit.

3.1. Grid Spacing and Cutting Tool Selection

There is a simple rule that can be followed in order to determine the optimal grid size for a given work piece; it should be equal to the highest common factor of all the dimensions within the desired product shape. This ensures that a grid overlay is produced that will align with all the edges of the desired shape, allowing an accurate definition of the target product to be built.

For cutting tool selection, one convention is to choose the largest possible tool that will fit the job. A larger tool can remove material faster than a smaller one could, thereby enabling potential savings on machining time. The limitation on tool diameter is usually either the size of the internal curves/arcs of a part, or the smallest gap between two parts of the desired product shape. For example, a 10 mm cutting tool can not cut any curves with a radius tighter than 5 mm, nor can it fit through a gap of less than 10 mm. If a smaller radius or gap is present in the desired product shape, the 10 mm tool would not be suitable.

Fig. 3 illustrates the grid spacing and cutting tool selection limitations. One key point to note is that the cutting tool may be larger than a single grid square as the two limitations are independent of each other. It will always be at least as big as the grid spacing and within the context of this model can only be a multiple of the grid size.

3.2. Tool Motion

With the work piece divided up into a discrete square grid, the tool movements can now be defined within the problem domain. From any single grid square, there are four permissible moves; to any one of the four adjacent squares in the direction of the machine axes (diagonal movements are prohibited within this model). In order to try and influence the tool paths to favour straight trajectories over many small turns, tool motion was modelled as a single movement (forward one unit) and two directional modifiers (turn left and turn right). Any tool motion within the 2D discretised grid can be formulated using a combination of these singular elements.

3.3. Limitations

By virtue of dividing the machining space into a grid of squares with each being designated cut or don't cut, the resultant shapes will only consist of straight horizontal or vertical lines. This makes it impossible to machine a perfect circle or curve between two points, but just an approximation using the discrete 'pixels' of the system. As Fig. 4 illustrates, a higher resolution grid will allow for a better approximation of the shape to be made, however, this will come at the expense of complexity and may be detrimental to the ability of the program to generate and optimise good tool paths.

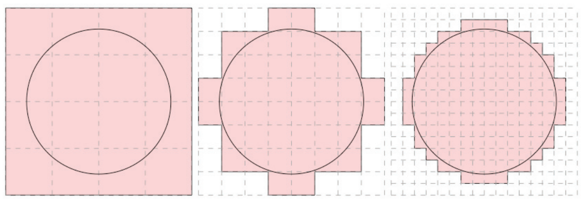

Fig. 4. When using a discrete grid model, the resultant shapes are made up of small squares or 'pixels'. This means that true curves can't be made, only approximations to the desired shape. This effect can be lessened by reducing the size of the grid, however, this increases complexity of the model and makes it harder to generate good tool paths.

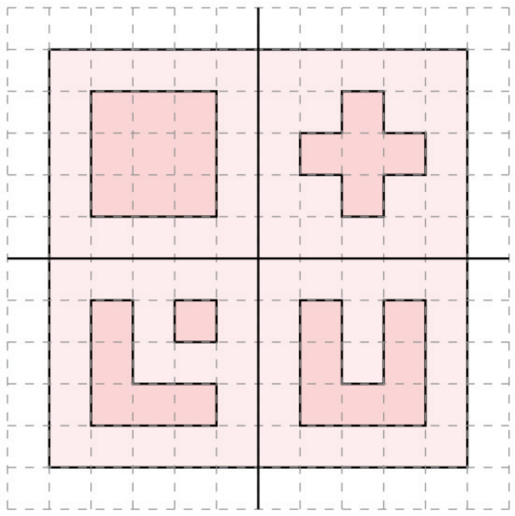

Fig. 5. Geometry of 4 initial test cases and their combination into larger, more complex case.

3.4. Mapping into the GP Domain

The terminal set components are the elements that make up the tool motion. Alongside this, there is a function set that has two functions that inform the structure of a program (progn2 and progn3) and two functions that allow the program to make decisions based on its immediate surroundings (if-shape and if-cut). The progn* functions take 2 and 3 arguments respectively and will run those arguments in order. This introduces a form of sub-routine where multiple commands can be evaluated one after the other. Nested progn* functions will produce longer sub-routines, where a longer length of commands can be run in order. The if-shape function will check if the square immediately in front of the tool (i.e., the square it would move to next) is part of the desired product shape. If it is, the first argument will be evaluated, and if not, the second one will. The if-cut function does the same, but the check is for whether the square in front has already been machined instead.

Every generation, each individual has their fitness measured. This involves evaluating the program tree repeatedly until the termination criteria are fulfilled - i.e all the desired grid squares have been cut, or a prescribed maximum number of moves has been reached. Listing 1 shows the Lisp code used in this investigation. The fitness function itself uses parameters such as the number of missed grid squares (those that it was trying to cut but didn't), the number of grid squares it cut that it shouldn't have, and a measure of relative tool path length (compared to number of squares it had to cut).

4. Test Cases & Results

To begin with, four very simple test cases were designed to test basic functionality of the algorithm. These simple 5 by 5 grid sections, as shown in Fig. 5, proved to be too simple and it was found that near-optimal solutions were being generated during initialisation of the first population.

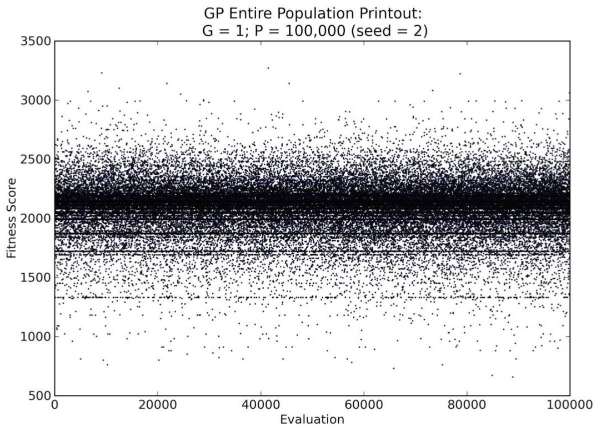

Fig. 6. A plot of 100,000 randomly generated programs. This population has not been directed towards an optimal solution and, as such, is just a random search across the solution space.

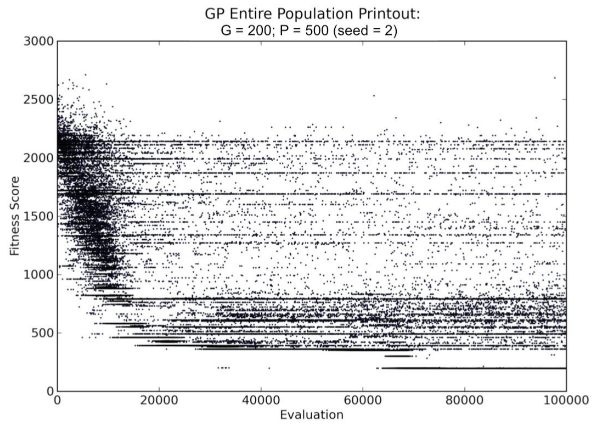

Fig. 7. A plot of all the individuals evaluated throughout the GP run, with a population size of 500 iterated through 200 generations. The plot shows quick progression initially, with most of the work being done before 20,000 evaluations (40 generations).

For the next step, all the previous 5 by 5 test cases were joined together into a single 10 by 10 test piece. This increased the complexity of the problem greatly and it was no longer trivial to find near-optimal solutions. In fact, many runs wouldn't even find a solution that would cut all the desired squares. The model of tool path movement was too general to consistently generate valid tool paths that completed the entire cutting operation. This results in a situation that is difficult for the fitness function to cope with as the initial priority must be to find valid tool paths that cut all the desired area, but then the priorities shift towards optimising those valid solutions to make them more efficient. Trying to perform both of these at the same time is hard and selection of the fitness function parameters and coefficients is a task that has little empirical evidence to help with the decisions. Despite this, Figs. 6 and 7 show that the GP optimisation was effective and actively directed the search towards better solutions.

By analysing Fig. 7, it can be seen that the majority of the work is being performed in the first 15 to 20 thousand evaluations. The incremental trend towards the lower fitness values

Listing 1. Common Lisp function for evaluating the fitness of an individual

```lisp
;; Fitness evaluation for individual programs
(defun evaluate-standard-fitness (program)
  "Evaluates a single program (argument) and reports its
   fitness, hits, and number of moves."
  (let ((standardised-fitness 0)
        (hits 0)
        (counter-hits 0)
        (moves-tally 0)
        (squares-to-hit *number-of-squares-to-cut*))
    (initialise)
    (catch :terminate-fitness-evaluation
      (dotimes (index *maximum-number-of-moves*)
        (when (or (>= hits squares-to-hit)
                  (>= moves-tally *maximum-number-of-moves*))
          (throw :terminate-fitness-evaluation
            (values standardised-fitness hits)))
        (eval program)
        (setf hits *hits*)
        (setf counter-hits *counter-hits*)
        (setf moves-tally *moves*)
        (setf standardised-fitness
          (floor (* 1 (+ (* 30 (- squares-to-hit hits))
                         (* 50 counter-hits)
                         (* 50 (/ moves-tally squares-to-hit))))))
        (values standardised-fitness hits)))))
```

(lower is better here) shows the process of active search. When this progression halts, the population then transitions out of the active search and back to a random search driven mainly by mutation and repeated crossover. The lines or peaks that seem to appear out of the plot at various fitness levels are where the best solutions are broken into their component parts by crossover during selection. At around the 60,000 evaluation mark a small jump in the fittest program found occurs. Here we can see the effect of this on the rest of the population - a line just above the 500 fitness mark seems to break down at this point, due to the subtle difference between the previous fittest individuals and the new fittest. In fact, it appears that the disturbance in this general area appears slightly before the new fittest is found, perhaps going on to contribute to its discovery soon after.

5. Conclusions

Traditional program generation methods use complex, intelligent, algorithms to dictate the tool paths for a given product. This paper has evaluated an alternative method where the individual elements of tool motion are defined and a system created that generates programs that use these to produce tool paths. These programs are then optimised by telling the system how to evaluate a given tool path for performance and allowing it to emulate evolution through natural selection.

There has been some success for using GP to generate tool paths for 2D milling. It has been shown to perform an active search and responds to changes in the product shape without problems. The method is compared to a random search algorithm that only found better solutions by chance and the behaviour of the population during evaluation was studied to identify emergent behaviour patterns.

References

[1] Tandon, V., El-Mounayri, H., Kishawy, H.. Nc end milling optimization using evolutionary computation. International Journal of Machine Tools and Manufacture 2002;42(5):595–605.

[2] Tolouei-Rad, M., Bidhendi, I.. On the optimization of machining parameters for milling operations. International Journal of Machine Tools and Manufacture 1997;37(1):1–16.

[3] Zain, A.M., Haron, H., Sharif, S.. Application of ga to optimize cutting conditions for minimizing surface roughness in end milling machining process. Expert Systems with Applications 2010;37(6):4650–4659.

[4] Rai, J.K., Brand, D., Slama, M., Xirouchakis, P.. Optimal selection of cutting parameters in multi-tool milling operations using a genetic algorithm. International Journal of Production Research 2011;49(10):3045–3068.

[5] Ganesan, H., Mohankumar, G., Ganesan, K., Kumar, K.R.. Optimization of machining parameters in turning process using genetic algorithm and particle swarm optimization with experimental verification. International Journal of Engineering Science and Technology Vol 3 No 2 Feb 2011 2011;.

[6] Cus, F., Balic, J.. Optimization of cutting process by ga approach. Robotics and Computer-Integrated Manufacturing 2003;19(1):113–121.

[7] Krimpenis, A., Vosniakos, G.C.. Rough milling optimisation for parts with sculptured surfaces using genetic algorithms in a stackelberg game 2009;20(4):447–461.

[8] Narasimhan, M.V.. Optimization of the tool path in a robotic environment 2007;.

[9] Petunin, A.. Tools path optimization for cnc cutting machines. Vestnik UGATU Systems Engineering and Information Technologies 2011;15(4 (44)):179–182.

[10] Medina-Rodriguez, N., Montiel-Ross, O., Sepulveda, R., Castillo, O.. Tool path optimization for computer numerical control machines based on parallel aco. Engineering Letters 2012;20(1):101.

[11] Vaupotic, B., Kovacic, M., Ficko, M., Balic, J.. Concept of automatic programming of nc machine for metal plate cutting by genetic algorithm method. Journal of Achievements in Materials and Manufacturing Engineering 2006;14(1-2).

[12] Gupta, A., Chandna, P., Tandon, P.. Hybrid genetic algorithm for minimizing non productive machining time during 2.5 d milling. International Journal of Engineering, Science and Technology 2011;3(1).

[13] Agrawal, R.K., Pratihar, D., Roy Choudhury, A.. Optimization of CNC isoscallop free form surface machining using a genetic algorithm 2006;46(7):811–819.

[14] Manav, C., Bank, H.S., Lazoglu, I.. Intelligent toolpath selection via multi-criteria optimization in complex sculptured surface milling 2013;24(2).