PATH PLANNING FOR ROBOT-ASSISTED RAPID PROTOTYPING OF ICE STRUCTURES

Alessandro Ossino[1], Eric Barnett[2], Jorge Angeles[2], Damiano Pasini[2], Pieter Sijpkes[3]
[1] *Department of Electrical, Electronic and System Engineering, University of Catania, Catania, Italy*
E-mail: alex.ossino@virgilio.it
[2] *Department of Mechanical Engineering, McGill University, Montreal, QC H3A 2K6, Canada*
E-mail: ebarnett@cim.mcgill.ca; angeles@cim.mcgill.ca; damiano.pasini@mcgill.ca
[3] *School of Architecture, McGill University, Montreal, QC H3A 2K6, Canada*
E-mail: pieter.sijpkes@mcgill.ca

Received October 2009, Accepted November 2009
No. 09-CSME-52, E.I.C. Accession 3138

ABSTRACT

The development of a path-planning algorithm for the robot-assisted rapid prototyping (RP) of ice structures is reported here. The algorithm, written in Matlab code, first imports a stereolithography (STL) file, which contains the geometry of the part to be built, and a text file containing other configuration parameters. The algorithm then finds intersection contours between evenly-spaced horizontal planes and the part; these contours define the boundaries of the areas to be filled for each layer. Each contour is then grouped with the other contours that define the same area. Subsequently, support structure contours are generated automatically from the part model; a support structure CAD model is not required. Then, part and support areas are filled by iteratively shrinking each outer contour until inner boundaries are reached.

Keywords: robot-assisted rapid prototyping; path planning; ice structures.

PLANIFICATION DE TRAJECTOIRES POUR LE PROTOTYPAGE ROBOTISÉ DE STRUCTURES EN GLACE

RÉSUMÉ

Cet article présente un algorithme de planification de la trajectoire du prototypage rapide robotisé de structures de glace. Écrit en code Matlab, l'algorithme importe, dans un premier temps, un fichier de stéréolithographie qui contient les données à construire, puis un fichier texte comportant les autres paramètres de configuration. L'algorithme détermine ensuite les contours des intersections entre les plans horizontaux équidistants et la pièce. Ces contours définissent les frontières des aires de chaque couche à remplir. Chaque contour est ensuite mis dans un groupe avec les autres contours qui définissent la même aire. Les contours pour la structure portante sont ensuite générés automatiquement à partir du modèle de la pièce, sans avoir besoin d'un modèle CAO de la structure portante. Les aires de la pièce et du support sont ensuite remplies en réduisant de manière itérative les contours extérieurs jusqu'aux frontières intérieures.

Mots-clés : prototypage rapide robotisé; planification de trajectoire; structures de glace.

1. INTRODUCTION

Practical ice structures such as ice roads and igloos are critical for winter survival in remote areas. Moreover, recreational structures such as ice sculptures and hotels have become more and more popular in recent years. Traditionally, ice structures have been built manually, making them labour-intensive and costly. However, in the past two decades, CNC ice sculpting has become quite popular. Two of the larger companies currently working in this field are Ice Sculptures Ltd. based in Grand Rapids, MI[1], and Ice Culture Inc. based in Hensall Ontario, Canada[2].

The path-planning algorithm reported here is part of a joint research project between the Department of Mechanical Engineering, the Centre for Intelligent Machines, and the School of Architecture, all at McGill University, entitled "The New Architecture of Phase Change: Computer-Assisted Ice Construction." The main objective of this project is to expand the formal design capabilities of ice as a winter building material using computer-assisted fabrication techniques such as computer numerical control (CNC) and rapid prototyping (RP). A detailed description of two rapid prototyping systems currently under development for this project, including a brief mention of the path-planning algorithm, is given in [1].

RP is a Solid Free-form Fabrication (SFF) technique, [2], which means that solid parts are built by material deposition. No specific tooling is required for RP, as in traditional manufacturing techniques such as milling and drilling, which remove material. RP is a SFF technique that is often used in industry to produce prototypes quickly and at low cost. RP with ice has additional advantages, namely, further reduced cost, small environmental impact, and high part accuracy and surface finish. Since RP is being used frequently now in industry, path planning for RP is not a new topic. However, for industrial RP machines, the algorithms used are typically protected and/or machine-specific.

Consequently, we have developed an algorithm that imports a stereolithography (STL) part model and plans the paths to build it with an Adept Cobra 600 robot installed in the Ice-Prototyping Laboratory. The algorithm is written in Matlab code, which is preferable to programming languages such as C or C++ because it is a superlanguage, which means many useful functions are available, less development time is needed, and debugging is easier. Matlab also allows users to compile code to make a program accessible to non-Matlab users. Specific objectives for our algorithm are to: (a) automatically generate support structure paths, when necessary; (b) generate paths that are as smooth as possible; and (c) export data in a format suitable for the Cobra 600.

Many steps are involved in the path-planning process. Literature on the subject typically focuses on one of the steps rather than the whole process [3–6]. In some cases, we have implemented subalgorithms similar to those proposed by other authors, while in others, our subalgorithms were developed from scratch. The one area of this work for which there is almost no pertinent literature is the arrangement of the path data in a format suitable for the Cobra 600.

The paper is organized as follows: In sections 2–6, the parts of the path-planning algorithm are described. In Section 2, input CAD models are sliced and closed contours are found for every layer; in Section 3, contours are divided into groups for every layer; in Section 4, the support structure is found by comparing contours in adjacent layers, starting from the top of

[1] http://machinedesign.com/ContentItem/60970/NCroutershapesiceart.aspx
[2] http://www.iceculture.com/main.cfm?id=5A166F80-1372-5A65-3BEEC7256C83B62C

the model; in Section 5, fill-in paths are found by shrinking outer contours; in Section 6, non-depositing paths are found and an output file is generated in a format suitable for the Cobra 600. Finally, the global software package scheme is outlined in Section 7.

For many parts of our algorithm, a data structure that accommodates entries of varying length is needed. Regular matrices and arrays do not accommodate this structure; however, the ''Cell Array'' structure in Matlab does, and is thus used whenever needed.

2. SLICING A STL FILE TO FORM INTERSECTION CONTOURS

The first part of the path-planning algorithm is to import the part geometry STL file and find intersection contours between evenly-spaced horizontal planes and the part.

2.1. Importing and Sorting STL data

The `slice` function first imports an input parameter file and an ASCII STL part geometry file, which can be generated by nearly all CAD software packages. In the STL format, a part is approximated by triangular facets, the acceptable accuracy of the approximation being specified by the user when generating the file. For each facet, three vertices and the normal vector are stored, and `slice` places this information in two matrices with single-precision entries. The latter are used because double precision is not necessary for this application, thereby shortening the algorithm execution time.

The vertices of each triangular facet are first sorted in ascending order of their z-coordinates. All the horizontal facets are ignored in this step, as they are parallel to the slicing planes.

As the layer thickness h is fixed by the user in the input TXT file, planes slicing facet i can be computed as

$$l^i_{min} = \lceil \frac{z^i_{min} - z_{min}}{h} \rceil + 1, \quad l^i_{max} = \lfloor \frac{z^i_{max} - z_{min}}{h} \rfloor + 1 \tag{1}$$

where $\lceil \cdot \rceil$ denotes the ceiling function, $\lfloor \cdot \rfloor$ denotes the floor function[3], while l^i_{min} and l^i_{max} are the lowest and highest planes slicing facet i. z^i_{min} and z^i_{max} are, respectively, the minimum and maximum z-coordinate of facet i, and z_{min} is the minimum z-coordinate among all the facets in the model. Using this technique, the range of layers that each facet intersects is found and stored in a cell array for later use.

2.2. Forming Bounding Contours on the Intersection Plane

For every slicing plane, the following procedure is applied: for each facet that intersects the slicing plane, two intersection points are found, thereby forming a segment of one of the bounding contours for that plane. The first point is obtained by intersecting the slicing plane with the segment joining the lowest and highest facet vertex. The other point is found by intersecting the slicing plane with the segment joining the middle vertex with the lowest or highest vertex, depending on whether the plane is lower or higher than the middle vertex. The two intersection points are stored in a matrix **P**. Figure 1 shows a graphical representation of a part being sliced.

[3] $\lceil \cdot \rceil$ returns the smallest integer greater than or equal to its floating point argument $(\cdot)$. Similarly, $\lfloor \cdot \rfloor$ returns the greatest integer less than or equal to its floating point argument $(\cdot)$.

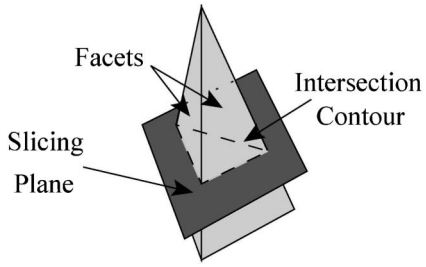


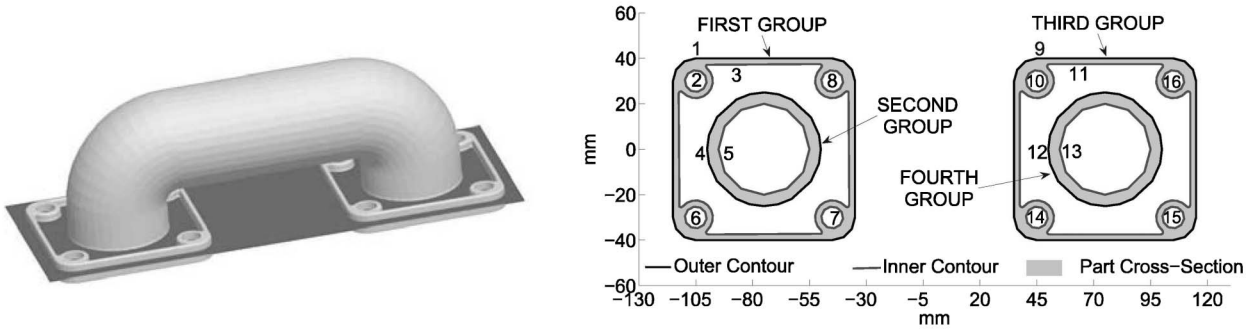
Fig. 1. Slicing a STL part.

If a facet has two vertices lying on the slicing plane, both are stored as intersection points. Of course, facets with only the highest or lowest vertex lying in the slicing plane are ignored. Moreover, since a large amount of redundant information is stored in the STL format, in some cases facet triangles will overlap, and duplicate segments could be produced. In order to avoid this, a segment is added to **P** only if no segments identical to itself are already present in the array.

Once all the intersection segments between the slicing plane and every facet have been found, the segments are joined to form bounding contours using the following technique: the first segment in **P** is defined as the first segment of the first contour. **P** is searched to find the second segment, which will have an identical point to the end-point of the first segment. The procedure is iterated, and when the first and last points of the contour coincide, the procedure is complete. If **P** is empty at this point, all contours in the slicing plane have been found; otherwise, the procedure is started over to define another contour.

2.3. Sorting and Filtering Bounding Contours

Next, outer contour points are sorted clockwise and inner contour points are sorted counterclockwise. Inner and outer contours are defined as shown in Fig. 2(b). Contours are distinguished as inner and outer contours using the z-coordinate z_1 of $\mathbf{v}_{1,2} \times \mathbf{v}_n$, where $\mathbf{v}_{1,2}$ is the vector from the first to the second contour point, and $\mathbf{v}_n$ is the vector normal to the facet that $\mathbf{v}_{1,2}$ lies on. If z_1 is negative, the order of the contour points is reversed. Contour points are sorted in this way in order to simplify many of the other functions used later in the path-



(a) (b)

Fig. 2. (a) STL model of a pipe fitting with the slicing plane shown; (b) classification of bounding contours.

Table 1. Hierarchy array **q** for the slice shown in Fig. 2(b).

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
q_j	0	1	1	3	4	1	1	1	0	9	9	11	12	9	9	9

planning algorithm. Finally, each closed contour is filtered to eliminate adjacent points that are too close, collinear points, and sequential segments in opposite directions.

3. GROUPING BOUNDING CONTOURS

Once all the closed bounding contours on each slicing plane have been found, a `group` function is executed to organize the contours in such a way that the cross-sectional area to be filled will be clearly defined for subsequent steps. Each outer contour is stored in a contour group, along with all contours inside of it that define the same area, as shown in Fig. 2.

3.1. Determining the Hierarchy Among Nested Contours

The `group` function is used to create an array **q** that defines the hierarchy among nested contours for layer l_i. Each contour γ_j is associated with array index j. The j^{th} entry of **q** is equal to the parent contour index p, that is, the closest contour that contains γ_j. If no contours include γ_j, then q_j is set to zero. q_j is computed by determining whether γ_j contains, is contained in, or is separate from any contour γ_k, which has already been classified in **q**. The entries of **q** are updated as necessary at each step. The hierarchy array for the slice shown in Fig. 2(b) is given in Table 1.

3.2. Defining a Cell Array of Contour Groups

The hierarchy array **q** is used to find out how deeply nested each contour γ_j is. If γ_j is contained by an even number of other contours, it must be an outer contour. A cell array of contour groups, $\mathbf{A}_i$, is then defined for l_i: each entry of $\mathbf{A}_i$ is an array whose first entry is an outer contour index and other entries indicate contours inside this outer contour, all defining the same area. The structure of $\mathbf{A}_i$ for the slice shown in Fig. 2(b) is given in Fig. 3.

4. SUPPORT STRUCTURE GENERATION

When a part is slanted by more than a certain limiting angle, a supporting structure must be built to ensure that part material is deposited at the correct elevation. The limiting angle can range from 0 to 45°: this value depends on the desired accuracy and the ratio of the path height to the path width . The material used for the support must be different from the part material so that it can be removed without too much difficulty after the part is completed. For rapid freeze

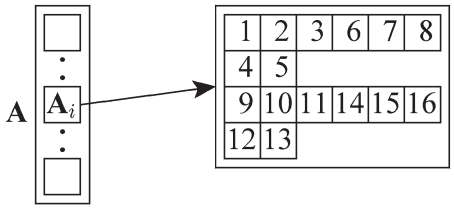


Fig. 3. Cell array **A** for slice shown in Fig. 2.

prototyping, we use a brine solution for the support structure; when the build completes, the brine is safely melted away in a freezer maintained at approximately -4°C, leaving the part intact. The selection of the specific brine solution used is discussed in [1].

Support structures can be modelled as separate parts, though this step can be time-consuming, and support models must often be quite complex. Therefore, it is desirable to develop an algorithm that will generate the paths for building the support structure using only the part geometry.

In the literature, many algorithms have been reported that generate support structures for CAD models to be built using RP. Chalasani et al. [7] consider only the 2D contours in each slicing plane, while other works [5, 8] consider the 3D model, analyzing triangular facets. We have developed a `support` function in Matlab which is similar to the first method: support-structure build paths are generated by comparing the contours of two adjacent layers at a time.

The `support` function is implemented after the contours for every layer of the part have been grouped. This function generates possible support areas for l_i by comparing the contours in l_i and l^*_{i+1}, starting with the highest layer. Note that the contours stored in l^*_{i+1} are the result of merging the part and support contours in l_{i+1}.

4.1. Cut-and-Merge Operations to Form Support Structure Bounding Contours

Two cases must be considered when forming support structure bounding contours. As shown in Fig. 4, in some cases two of the contours in layers l_i and l^*_{i+1} intersect, while in others they do not. For the first case, cut-and-merge operations are necessary to form the support-structure bounding contours. The second case is simpler, as the support area, if necessary, is defined directly by two contours in l_i and l^*_{i+1}. For both cases, the data scheme implemented in Section 3 is exploited to identify the contours that define the support structure area for each layer.

4.2. Finding Intersection Points For Intersecting Contours in Layers l_i and l^*_{i+1}

First, all intersection points between contours in layers l_i and l^*_{i+1} are found. Since each contour is represented by an array of points which form a closed polygon when joined, the intersection points are determined by finding intersections between the line segments of the two contours at hand. In order to speed up the algorithm, intersection points are only computed when the rectangles formed by the x- and y-coordinate extrema for the two contours overlap.

Each segment g_k in l_i or l^*_{i+1} is associated with two end-points represented by two-dimensional vectors $\mathbf{p}_{ak}$ and $\mathbf{p}_{bk}$, as shown in Fig. 5. A scalar parameter t_k varies between 0 and

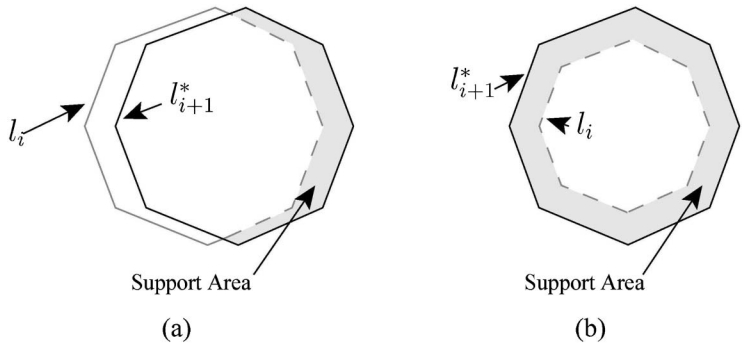


Fig. 4. Support areas: (a) intersecting contours in adjacent layers; (b) non-intersecting contours in adjacent layers.

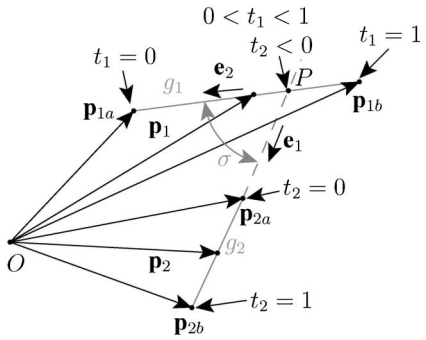


Fig. 5. The intersection of segments g_1 and g_2 in layers l_i and l^*_{i+1}.

1 along g_k. If g_1 and g_2 are arbitrary segments in l_i or l^*_{i+1}, vectors $\mathbf{p}_1$ and $\mathbf{p}_2$, which represent the position of points along g_1 and g_2, respectively, can be expressed as

$$\mathbf{p}_1 = \mathbf{p}_{1a}(1-t_1) + \mathbf{p}_{1b}t_1, \quad \mathbf{p}_2 = \mathbf{p}_{2a}(1-t_2) + \mathbf{p}_{2b}t_2$$

Parallelism between segments g_1 and g_2 can be readily detected by $\sin\sigma$, with σ indicated in Fig. 5. In terms of the unit vectors $\mathbf{e_1}$ and $\mathbf{e_2}$ of the same figure,

$$\sin\sigma = \mathbf{e}_2^T \mathbf{E} \mathbf{e}_1, \quad \mathbf{E} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \tag{2}$$

If $|\sin\sigma| < \varepsilon$, for ε small enough, then the segments are declared parallel; otherwise, the intersection point P is determined by the values of t_1 and t_2, which can be obtained from the solution of Eq. 2:

$$t_1 = \frac{(\mathbf{p}_{2a}-\mathbf{p}_{2b})^T \mathbf{E}(\mathbf{p}_{2a}-\mathbf{p}_{1a})}{(\mathbf{p}_{2a}-\mathbf{p}_{2b})^T \mathbf{E}(\mathbf{p}_{1b}-\mathbf{p}_{1a})}, \quad t_2 = \frac{(\mathbf{p}_{1a}-\mathbf{p}_{1b})^T \mathbf{E}(\mathbf{p}_{2a}-\mathbf{p}_{1a})}{(\mathbf{p}_{2a}-\mathbf{p}_{2b})^T \mathbf{E}(\mathbf{p}_{1b}-\mathbf{p}_{1a})} \tag{3}$$

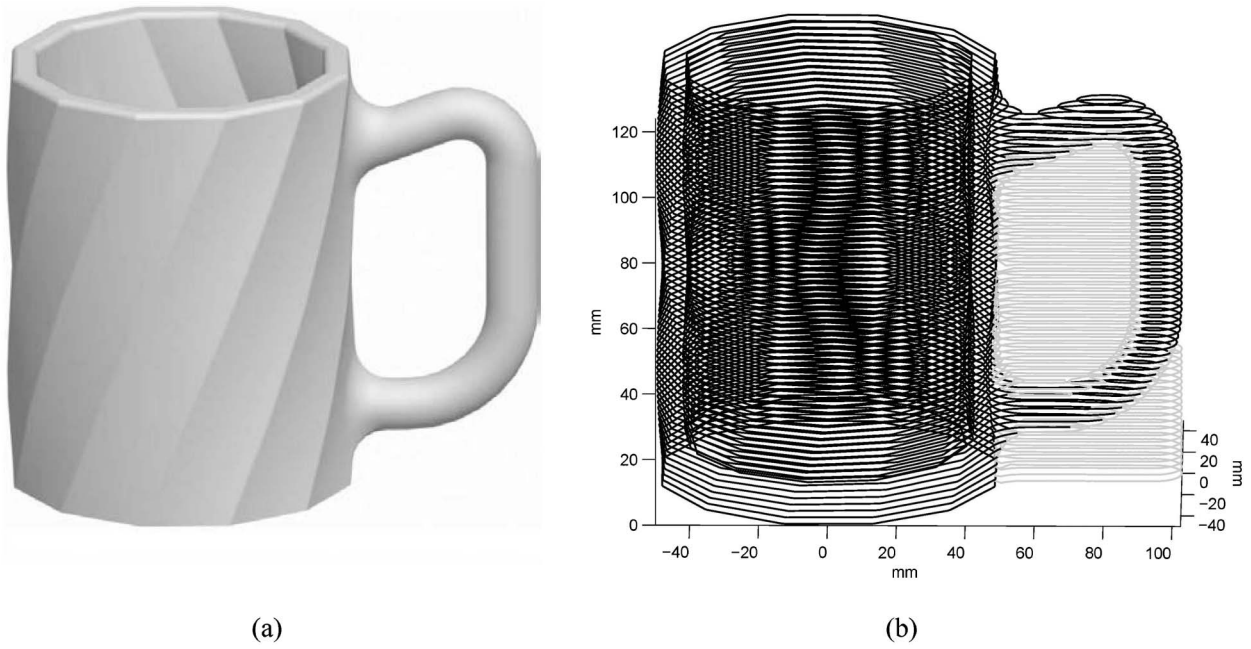
If $0 \le t_1 \le 1$ and $0 \le t_2 \le 1$, then segments g_1 and g_2 intersect. Otherwise, the lines containing these segments intersect, but the segments do not, as seen in Fig. 5. For intersecting segments, the intersection point is calculated, for robustness, as the mean value of the two expressions of Eq. (2). Then, this point and the segment indices in their respective contours are stored in new arrays.

A `filter` function is applied to each support contour to delete points spaced less than twice the deposit path width, $2p_w$, from any contour segment. If a contour is too small, the `filter` function deletes the entire contour.

Using the procedures described in this section, a contour cell array **B** is defined for the support structure which is equivalent to the contour cell array **A** defined for the part to be built. In Fig. 6, a sliced beer mug model and the support structure for its handle are shown.

5. FILLING PATH GENERATION

Many authors [6, 9, 10] reportedly use zig-zag paths to fill object areas in their RP algorithms. Xu and Shaw [11] consider 2D material gradients within each layer to produce smooth filling



(a) (b)

Fig. 6. Generating support structures for a beer mug: (a) CAD model; (b) part (black) and support structure (grey) bounding contours.

paths. Zig-zag filling segments are relatively simple since the longest filling segments for a layer are all parallel, and these paths are joined by shorter segments near the bounding contours. The zig-zag technique is also relatively robust, since it will usually work quite well with complex parts. When zig-zag paths are followed by an RP machine, however, abrupt changes in direction are required, resulting in high dynamic loads and loss of accuracy for the part being built. Therefore, we are introducing a different technique that will generate much smoother filling paths for most objects. We have developed a `fill` function, which takes the outer contour in a contour group and iteratively shrinks it inwards until it becomes too close to inner contours or to itself.

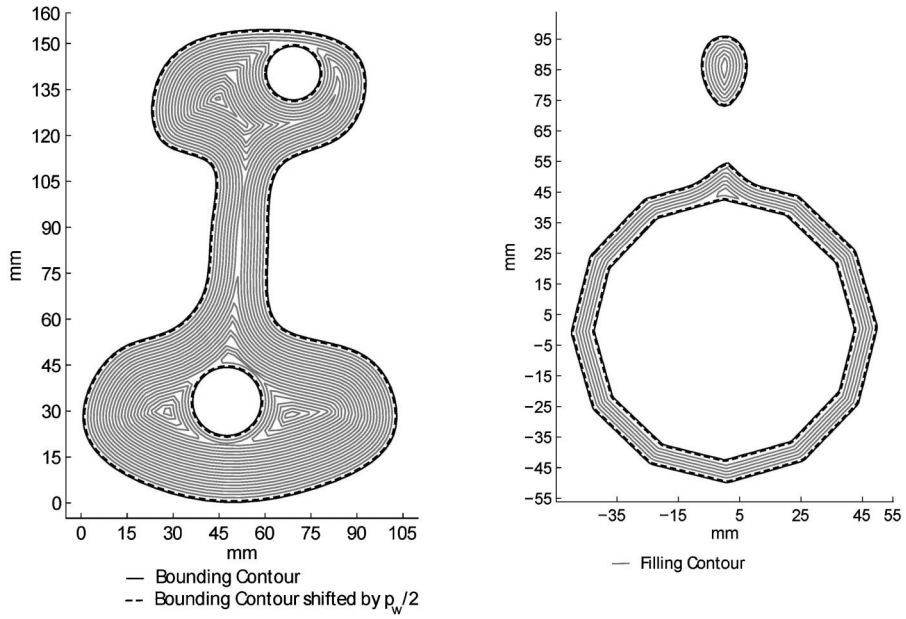
Before filling paths can be found, we must first shrink the bounding contours by $p_w/2$, so that the edges of the deposited water correspond to the edges of the part to be built. This is achieved by calling the `shrink` function once for every contour.

5.1. Shrinking a Single Contour

Contour point $\mathbf{p}_i$ is considered along with the two contour segments g_1 and g_2 that connect to $\mathbf{p}_i$. The vector $\mathbf{v}$, which bisects the angle formed by g_1 and g_2, is found. If we shrink the contour by s, and the angle between g_1 and g_2 is σ, the distance that $\mathbf{p}_i$ must be moved along $\mathbf{v}$ is given by

$$r = \frac{s}{\sin(\sigma/2)} \tag{4}$$

Since there are two possible directions for $\mathbf{v}$, a filter must be applied to ensure the correct one is chosen. The z-coordinate z_1 of $\mathbf{v} \times (\mathbf{p}_{i+1} - \mathbf{p}_i)$ is considered. If z_1 is negative, the direction of vector $\mathbf{v}$ is reversed. Note that using this filter, inner and outer contours are expanded and shrunken, respectively, because of the order of contour points imposed in Section 2.



(a) (b)

Fig. 7. Contours with filling paths for part cross-sections: (a) a dumb-bell shaped part; (b) a beer mug.

If we let $\mathbf{p}_i$ and $\mathbf{p}'_i$ be points on the original and shrunken contours, respectively, we have

$$\mathbf{p}'_i = \mathbf{p}_i + \mathbf{v}. \tag{5}$$

Once we have shrunk all the bounding contours in every layer by $p_w/2$ in the correct direction, the `fill` function is called, in order to fill the object areas with several smooth paths.

For layer l_i, each shrunken contour group is considered, by recalling each entry in the cell array $\mathbf{A}_i$. Every group is input to the `fill` function together with a threshold value t_h, which allows the user to specify the minimum distance between filling path points. Note that t_h needs to be sufficiently small to ensure paths are smooth, but large enough so that points are adequately spaced for the the V+ programming language used with the Adept Cobra 600 robot. For example, if the desired speed along a path is v_p in mm/s, points must be spaced by at least $0.016v_p$ mm because the standard trajectory generation frequency for V+ is 62.5 Hz[4]. If the points are spaced closer than this, the actual speed observed will be lower than desired. Additionally, the computational capabilities of the Cobra controller are limited[5], so it is desirable to have as few points as possible to avoid processing delays.

5.2. Iteratively Shrinking Outer Contours and Performing Cut, Merge, and Filter Operations

Our `fill` function first calls a `cut` function in order to recognize whether the contour we want to shrink is too narrow at any point, or if is too close to inner contours. When necessary,

[4] With an optional software licence, trajectory frequencies of up to 500 Hz are possible.
[5] Our Adept C40 Compact Controller carries a AWC-II 040 Processor (25 MHz), 32MB RAM, and a 128MB CompactFlash disk.

cut-and-merge operations are performed to define new bounding contours. Note that examples of both of these cases can be seen in Fig. 7(a). To create the filling paths, `shrink` is used to shrink the outermost contours of each contour group iteratively inward by p_w.

Following each iteration, the `filter` function is applied to remove points on the contour that are too close to each other. The `fill` function continues until every filling path has become too small and has been removed by either `cut` or `filter`.

In Fig. 7, the filling paths generated with the `fill` function for two different part slices are shown. Figure 7(a) shows a slice of a dumb-bell shaped part with two holes, while Fig. 7(b) shows a slice of a beer mug.

The `fill` function described in this section is quite complex compared to the zig-zag techniques, as can be seen from the computational time results shown in Table 2. However, `fill` also produces smoother paths which can be followed more closely by the RP system, resulting in higher part-accuracy. Of course, the advantages of using our `fill` function depend to a great extent on the part being built. For the beer mug slice shown in Fig. 7(b), `fill` is clearly the superior technique, though for the dumb-bell slice shown in Fig. 7(a), the advantages of using `fill` are less apparent, since there are some abrupt changes in direction.

6. DEFINING NON-DEPOSITING PATHS AND EXPORTING DATA

All the paths needed to build the CAD model and its support structure must be written in an output format suitable for the V+ programming language used with the Adept Cobra 600 robot. Therefore, an `export` function is defined, which exports the paths as a series of points. Each point is represented by its (x, y, z) coordinates, and an additional parameter is defined at each point to control the `ON1`/`ON2`/`OFF` state of the deposition system. The deposition system states are: `ON1`: water is being deposited to form the part structure; `ON2`: brine is being deposited to form the support structure; `OFF`: nothing is being deposited. In this way, the end-point of every closed contour must mark the start of a non-depositing segment.

7. SOFTWARE PACKAGE

The path-planning algorithm is written in Matlab code, using version R2008a. The algorithm can be executed within the Matlab environment; however, a stand-alone executable has also been developed that can be executed by users who do not have Matlab installed. For both cases, the program reads in a part geometry STL file and a parameter TXT file and outputs a trajectory data file. The input TXT file contains the STL file name, the deposition path width p_w, the layer thickness h, and the threshold value used for the filling t_h.

The Matlab Compiler, which is included with a standard Matlab distribution, is used to create stand-alone executables. A runtime engine, called the Matlab Compiler Runtime (MCR), can be used to run any executable built with the Matlab Compiler. The MCR is also included

Table 2. Computation time results for a beer mug part: height = 125 mm; layer height = 2 mm; path width = 1 mm.

Case	Time (s)	Number of trajectory points
Zig-zag	36	24 793
`fill` function	364	70 274
`fill` function with support structure	971	88 205

with a standard Matlab distribution, and may be re-distributed free of charge to others who only need to run standalone executables.

8. CONCLUSIONS

We reported on a new path-planning algorithm, composed of several functions, to be used for robot-assisted rapid prototyping systems for ice construction. The algorithm is able to slice any CAD model and find closed contours for each layer, even when the part cross-section contains several nested contours. A support structure function has been written and tested on a martini glass model and a beer mug model. A filling function generates smoother paths than the simpler, more traditional zig-zag technique. Non-depositing segments have been inserted at the starting- and end-points of each contour to reduce speed variations along depositing paths. Finally, by using the Matlab Compiler, a stand-alone executable can be generated from the code.

Experimental tests will be done using the trajectories generated with the path-planning algorithm. We will then assess the quality of the built ice structures and the performance of the Cobra 600 while following the synthesized trajectories; the algorithm will be modified as needed.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support of The Social Sciences and Humanities Research Council of Canada (SSHRC), the Fonds québecois de la recherche sur la nature et les technologies, and the Fondation universitaire Pierre Arbour. The generous rebate received from Adept Technology is dutifully acknowledged.

REFERENCES

1. Barnett, E., Angeles, J., Pasini, D. and Sijpkes, P., ''Robot-assisted rapid prototyping for ice structures,'' in *IEEE Int. Conf. on Robotics and Automation*, Kobe, Japan, pp. 146–151, May 2009.
2. Crawford, R.H. and Beaman, J.J., ''Solid freeform fabrication,'' *IEEE Spectrum*, Vol. 36, No. 2, pp. 34–43, 1999.
3. Choi, S.H. and Kwok, K.T., ''A tolerant slicing algorithm for layered manufacturing,'' *Rapid Prototyping Journal*, Vol. 8, No. 3, pp. 161–179, 2002.
4. Haipeng, P. and Tianrui, Z. ''Generation and optimization of slice profile data in rapid prototyping and manufacturing,'' *Rapid Prototyping Journal*, Vol. 187–188, pp. 623–626, 2007.
5. Allen, S. and Dutta, D. ''Determination and evaluation of support structures in layered manufacturing,'' *Journal of Design and Manufacturing*, Vol. 5, No. 3, pp. 153–162, 1995.
6. Luo, R.C., Pan, Y.L., Wang, C.J., Huang, Z.H., ''Path planning and control of functionally graded materials for rapid tooling,'' in *IEEE Int. Conf. on Robotics and Automation*, Orlando, FL, pp. 883–888, May 2006.
7. Chalasani, K., Jones, L. and Roscoe, L., ''Support generation for fused deposition modeling,'' in *6th Solid Freeform Fabrication Symposium*, Orlando, FL, pp. 229–241, 1995.
8. Huang, X., Ye, C., Wu, S., Guo, K. and Mo, J., ''Sloping wall structure support generation for fused deposition modeling,'' *Int. Journal of Advanced Manufacturing Technology*, 2008, DOI: 10.1007/s00170-008-1675-2.

9. Luo, R.C., Chang, C.L., Tzou, J.H. and Huang, Z.H., "Automated desktop manufacturing: Direct metallic rapid tooling system," in *IEEE Int. Conf. on Robotics and Automation*, Barcelona, Spain, pp. 584–589, May 2005.
10. Chen, H., Xi, N., Sheng, W., Chen, Y., Roche, A. and Dahl, J., "A general framework for automatic CAD-guided tool planning for surface manufacturing," in *IEEE Int. Conf. on Robotics and Automation*, Taipei, Taiwan, pp. 3504–3509, Sept. 2003.
11. Xu, A. and Shaw, L.L. "Equal distance offset approach to representing and process planning for solid freeform fabrication of functionally graded materials," *Computer-Aided Design*, Vol. 37, pp. 1308–1318, 2005.