Path Planning for Robot-Assisted Rapid Prototyping of Ice Structures

TR-CIM-09-02 February, 2009

Alessandro Ossino[1], **Eric Barnett**[2]

[1]Graduate in Automation and Complex Systems Control Engineering,
Department of Electrical, Electronic and System Engineering,
University of Catania, Catania, Italy,
alex.ossino@virgilio.it

[2]Centre for Intelligent Machines, Department of Mechanical Engineering,
McGill University, Montreal, QC H3A 2K6, Canada,
ebarnett@cim.mcgill.ca

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. The contours are then grouped according to the fill areas they define. 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. Finally, an output file is generated, containing the depositing and non-depositing paths, in a format suitable for the Adept Cobra 600 robot.

Contents

List of Figures

List of Tables

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 work reported here is part of a joint research project between the Department of Mechanical Engineering and the School of Architecture 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 is given in [1].

RP is a Solid Freeform 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 in Matlab that imports a stereolithography (STL) part model and plans the paths to build it with an Adept Cobra 600 robot installed in the Ice-Prototypig Laboratory. Specific objectives we had for our algorithm were to: (a) automatically generate support structure paths, when necessary; (b) generate paths that are as smooth as possible; (c) be accessible to users who do not have Matlab; and (d) export data in a format suitable for the Cobra 600.

There are many steps involved in the path-planning process. Literature on the subject typically focuses on one of the steps rather than the whole process [3; 4; 5; 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 arrangment 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 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

[1] http://machinedesign.com/ContentItem/60970/NCroutershapesiceart.aspx

[2] http://www.iceculture.com/main.cfm?id=5A166F80-1372-5A65-3BEEC7256C83B62C

the Cobra 600. Finally, the global software package scheme is outlined in Section 7. Steps of the path-planning algorithm are shown in Fig. 1.

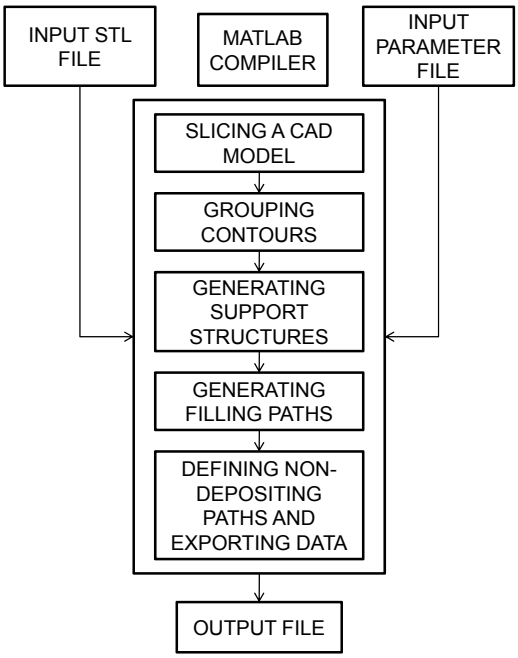


Figure 1: Flow diagram for the path-planning algorithm

For many parts of our algorithm, a data structure that accomodates entries of varying length is needed. Regular matrices and arrays do not accomodate 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, and the acceptable accuracy of the approximation is 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

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

where $\lceil \cdot \rceil$ denotes the ceiling function, $\lfloor \cdot \rfloor$ denotes the floor function,[3] 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, 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 $\mathbf{P}$. Figure 2 shows a graphical representation of a part being sliced.

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 $\mathbf{P}$ 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 $\mathbf{P}$ is defined as the first segment of the first contour. $\mathbf{P}$ 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 $\mathbf{P}$ 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. 5(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

[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 $(\cdot)$.

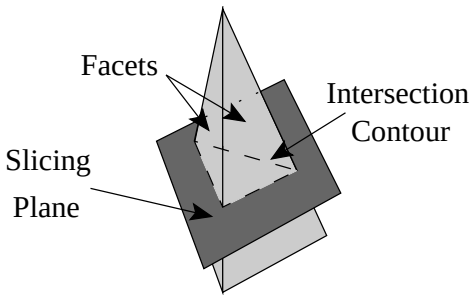


Figure 2: Slicing a STL part

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

2.4 Results with the `slice` Function

Figure 3 is an example of contours obtained by using the `slice` function, where a 3D model of the Chartres maze has been designed with Pro/Engineer, and all the closed contours lying in a slicing plane have been considered. It is important to notice that in Fig. 3, only two closed contours are present: the first is the central flower, while the second represents the entire maze.

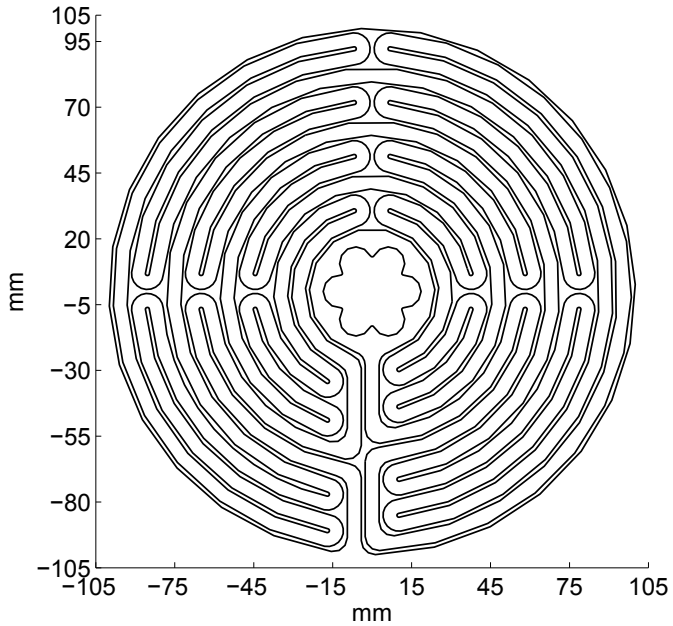
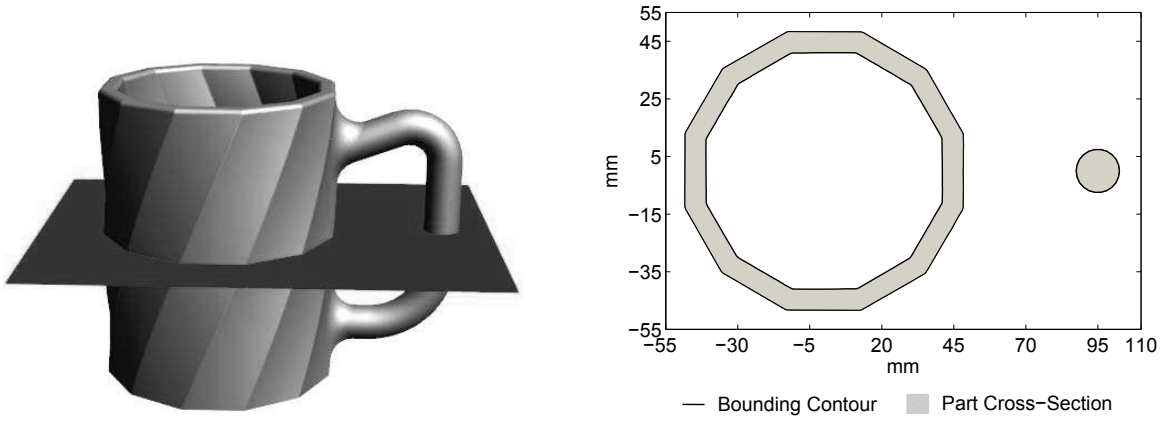


Figure 3: Bounding contours of the Chartres maze

Figure 4(a) shows a STL model of a beer mug designed with Pro/Engineer and sliced by a plane, while bounding contours lying in that plane are shown in Fig. 4(b).

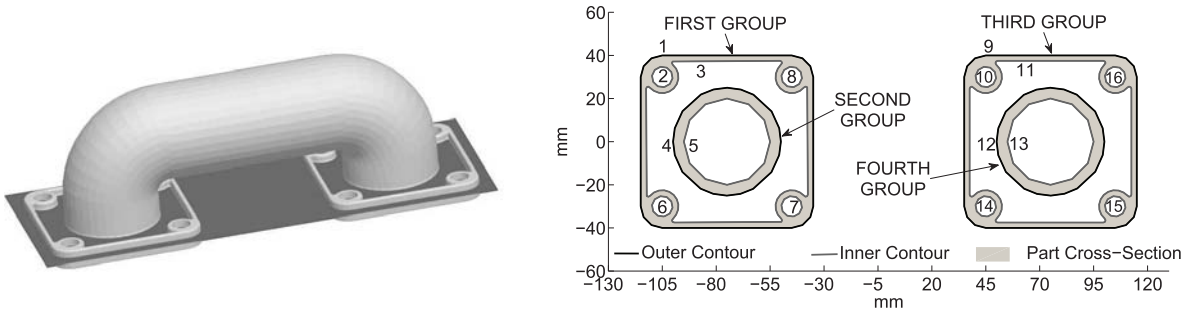


(a) (b)

Figure 4: (a) A STL model of a beer mug with the slicing plane shown; (b) Bounding contours for a slice of the beer mug

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. 5. Figure 6 shows the main steps implemented in the `group` function.



(a) (b)

Figure 5: (a) A STL model of a pipe fitting with the slicing plane shown; (b) Classification of bounding contours

3.1 Determining the Hierarchy Among Nested Contours

The `group` function is used to create an array $\mathbf{q}$ 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 $\mathbf{q}$

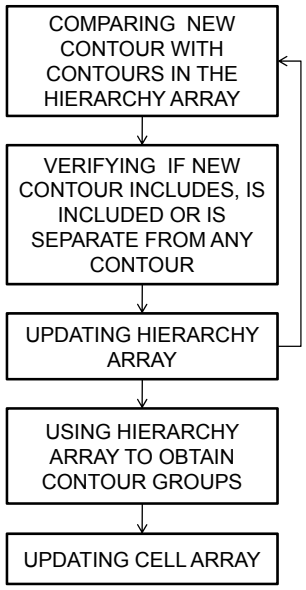


Figure 6: Flow diagram for the `group` function

is equal to the parent contour index p, that is, the closest contour that contains γ_j. If no contours include γ_j, then $\mathbf{q}_j$ is set to zero.

Entry $\mathbf{q}_j$ 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 $\mathbf{q}$, by calling the `inpolygon` Matlab function. `inpolygon` can determine if a point is inside, outside or on the edge of a given polygon. If γ_j is separate from all the contours classified in $\mathbf{q}$, then $\mathbf{q}_j$ is set to zero. If γ_j includes γ_k and $\mathbf{q}_k = 0$, $\mathbf{q}_j$ is set to zero and $\mathbf{q}_k$ is set to j. If γ_j is included in γ_k, γ_j is compared to all contours γ_s inside γ_k; $\mathbf{q}_j$ is set to r, where γ_r is the most nested contour in γ_k that includes γ_j. For all contours γ_s included in γ_j, $\mathbf{q}_s$ is set to j. The hierarchy array for the slice shown in Fig. 5(b) is given in Table 1.

Table 1: Hierarchy array $\mathbf{q}$ for the contours shown in Fig. 5(b)

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathbf{q}_j$	0	1	1	3	4	1	1	1	0	9	9	11	12	9	9	9

3.2 Forming Contour Groups to Define Areas to be Filled

The hierarchy array $\mathbf{q}$ is used to find out how deeply nested each contour γ_j is. First, the number of contours n_c that include γ_j is calculated as follows. If $p = \mathbf{q}_j$ is equal to zero, then n_c is set to zero. Otherwise, n_c is increased by one and we set $p = \mathbf{q}_p$. The loop continues until a zero value for $\mathbf{q}_p$ is found.

A cell array of contour groups $\mathbf{A}_i$ is then defined for l_i as follows: each entry of $\mathbf{A}_i$ is an array, whose first entry is a outer contour index and other entries indicate contours inside this outer contour, all defining the same area. If n_c is even or zero, then γ_j must be an outer

contour, and its index is stored in the first entry of an array of $\mathbf{A}_i$, with the other entries being the indices of its included contours, that is, the contours γ_k for which $\mathbf{q}_k = j$. If n_c is odd, then γ_j must be an inner contour; the parent contour index $p = \mathbf{q}_j$ is stored as the first entry of an array of $\mathbf{A}_i$, with the other entries being j and the indices of all other contours contained in the parent contour. When all the groups in layer l_i have been stored in cell array $\mathbf{A}_i$, the procedure stops. The structure of $\mathbf{A}_i$ for the slice shown in Fig. 5 is given in Fig. 7.

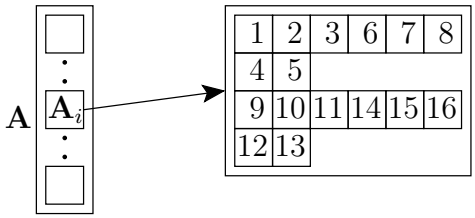


Figure 7: Cell array $\mathbf{A}$ for the contours shown in Fig. 5(b)

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 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} define the cross-sectional area in l_{i+1} that must be supported by all layers below; contours in l^*_{i+1} are found by merging the part and support contours in l_{i+1}.

4.1 Cut-and-Merge Operations

Two cases must be considered when forming support structure bounding contours. As shown in Fig. 8, 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 Sections 2 and 3 is exploited to identify the support structure bounding contours each layer.

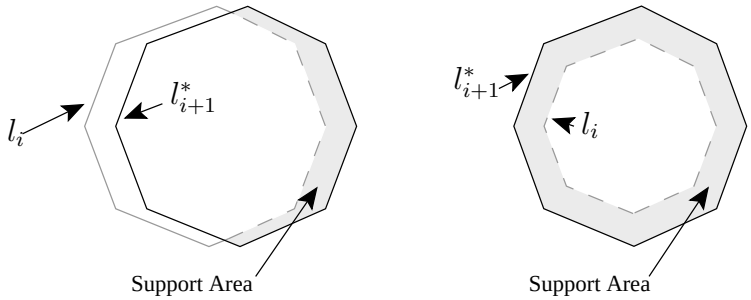


Figure 8: Support areas: (a) intersecting contours in adjacent layers; (b) non-intersecting contours in adjacent layers

4.2 Case 1: Contours in Adjacent Layers Intersect

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. 9. A scalar parameter t_k varies between 0 and 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

$$\begin{aligned} \mathbf{p}_1 &= \mathbf{p}_{1a}(1-t_1) + \mathbf{p}_{1b}t_1 \\ \mathbf{p}_2 &= \mathbf{p}_{2a}(1-t_2) + \mathbf{p}_{2b}t_2 \end{aligned} \tag{2}$$

Parallelism between segments g_1 and g_2 can be readily detected by $\sin\sigma$, with σ indicated in Fig. 9. In terms of the unit vectors $\mathbf{e_1}$ and $\mathbf{e_2}$,

$$\sin\sigma = \mathbf{e}_2^T\mathbf{E}\mathbf{e}_1, \qquad \mathbf{E} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \tag{3}$$

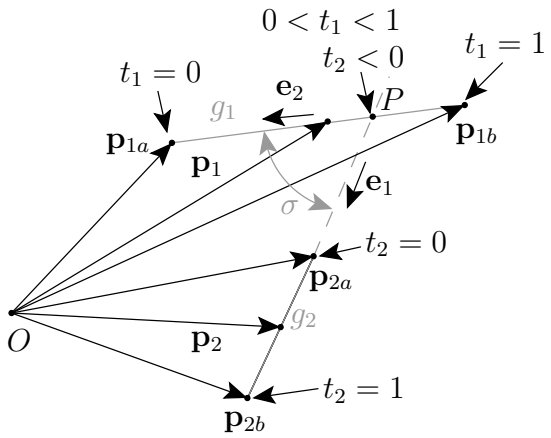


Figure 9: The intersection of segments g_1 and g_2 in layers l_i and l^*_{i+1}

If $|\sin\sigma| < \epsilon$, 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 eqs. 2:

$$t_1 = \frac{(\mathbf{p}_{2a} - \mathbf{p}_{2b})^T \mathbf{E} (\mathbf{p}_{2a} - \mathbf{p}_{1a})}{(\mathbf{p}_{2a} - \mathbf{p}_{2b})^T \mathbf{E} (\mathbf{p}_{1b} - \mathbf{p}_{1a})} \tag{4a}$$

$$t_2 = \frac{(\mathbf{p}_{1a} - \mathbf{p}_{1b})^T \mathbf{E} (\mathbf{p}_{2a} - \mathbf{p}_{1a})}{(\mathbf{p}_{2a} - \mathbf{p}_{2b})^T \mathbf{E} (\mathbf{p}_{1b} - \mathbf{p}_{1a})} \tag{4b}$$

If $0 \leq t_1 \leq 1$ and $0 \leq t_2 \leq 1$, then segments g_1 and g_2 intersect. Otherwise, the lines containing these segments intersect, but the segments do not, as seen in Fig. 9. 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.

The second step in `support` is to cut the contours in l_i and l^*_{i+1} at the computed intersection points. Half of the contour pieces are needed to define the support structure, while the other half define areas that overlap with the part cross section for l_i. The following procedure is implemented to remove the unwanted contour pieces.

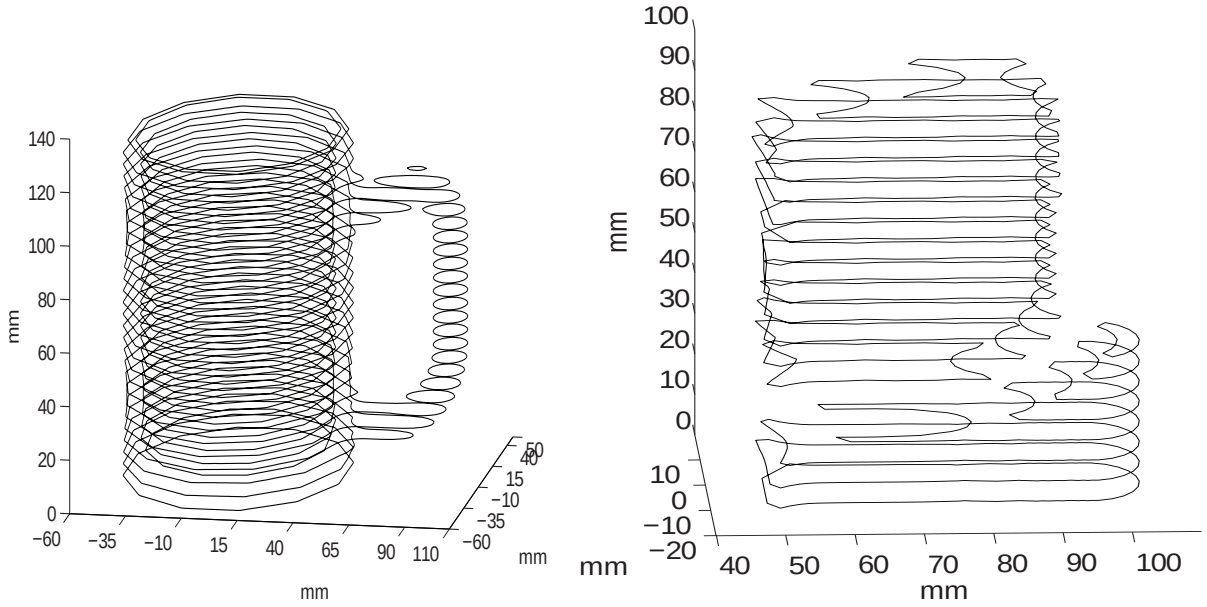
For each contour lying in layer l_i, the cross product vector $\mathbf{w}$, given by $(\mathbf{p}_{1b} - \mathbf{p}_{1a}) \times (\mathbf{p}_{2b} - \mathbf{p}_{2a})$, is calculated. If w_z is negative, then the contour piece starting at the intersection point on segment g_1 and ending at the next intersection point on the contour being considered is kept. The next contour piece to be kept starts at the third intersection point and ends at the fourth intersection point of the contour being considered. In this way, only pieces starting with an odd intersection point are kept. If the z-coordinate of $\mathbf{w}$ is positive, then only contour pieces starting with an even intersection point are kept. All kept contour pieces on layer l_i are stored in a new cell array $\mathbf{B}^s_i$. Contour pieces that are not kept are stored in cell array $\mathbf{B}^m_i$ for later use. This technique works because of the way inner and outer contours were defined in Section 2.3.

The procedure described in the previous paragraph is then repeated for l^*_{i+1} to form the array $\mathbf{B}^s_{i+1}$, except the sign of w_z implies the opposite of what it did for l_i.

Contour pieces stored in $\mathbf{B}_i^s$ and $\mathbf{B}_{i+1}^s$ with a common end-point must be merged, in order to create closed contours. To achieve this, a contour piece in $\mathbf{B}_{i+1}^s$ is considered and another piece in $\mathbf{B}_i^s$ with a common end-point is linked to it. If the contour is not still closed, the procedure is repeated by searching in $\mathbf{B}_{i+1}^s$ for another piece to be added, followed by an additional search in $\mathbf{B}_i^s$. When the contour is closed, a new starting piece from $\mathbf{B}_{i+1}^s$ is considered. The loop stops when all pieces in $\mathbf{B}_i^s$ and $\mathbf{B}_{i+1}^s$ have been considered.

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

The contour pieces in $\mathbf{B}_i^m$ and $\mathbf{B}_{i+1}^s$ with common end-points are merged together using a procedure similar to that described above, forming the contours for l_i^*, which defines the area that must be supported by l_{i-1}. Finally, the `group` function is applied to the all contours lying in l_i^*.



(a) (b)

Figure 10: A beer mug model: (a) part contours; (b) support structure contours

4.3 Case 2: Contours in Adjacent Layers Do Not Intersect

A different procedure is applied for a contour that does not intersect any contour on an adjacent layer. We will denote such a contour γ_p if it lies on l_i, and γ_q if it lies on l_{i+1}^*. Contour γ_p will bound an area to be filled and will be stored in $\mathbf{B}_i^s$ only if its projection on l_{i+1}^* lies on an area to be filled, that is, only if the projection is contained by an outer contour of l_{i+1}^* and not by any of its inner contours. Otherwise, contour γ_p is stored in $\mathbf{B}_i^m$. Contour γ_q will bound an area to be filled and will be stored in $\mathbf{B}_{i+1}^s$ and $\mathbf{B}_i^m$ only if its projection on l_i does not lie on an area to be filled, that is, only if the projection is contained by an outer contour of l_i and one of its inner contours, or is not contained by any outer contour. Then

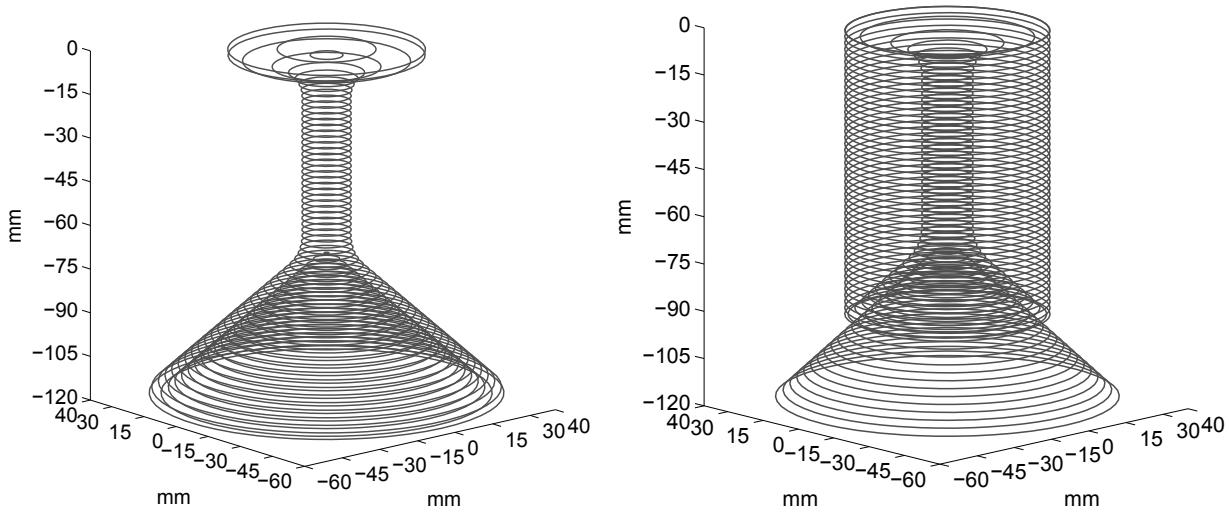


Figure 11: A martini glass model: (a) part contours; (b) support structure contours

the `group` function is applied to contours stored in $\mathbf{B}_i^s$ and $\mathbf{B}_{i+1}^s$. The order of the contour points is reversed if an outer contour becomes inner one or vice versa. Also, for this case, $\mathbf{B}_i^m$ contains the contours in l_i^*

Using the procedures described in this section, a contour cell array $\mathbf{B}$ is defined for the support structure, which is equivalent to the contour cell array $\mathbf{A}$ defined for the part to be built. In Fig. 10, a sliced beer mug model and the support structure for its handle are shown. For the beer mug model, the deposition path width p_w is equal to 1 mm and the layer thickness h is 5 mm. In Fig. 11, a sliced martini glass model and support structures for its base and top part are shown. For the martini glass model, p_w and h are both set to 1 mm. Finally, Fig. 12(a) shows the flow diagram for the `support` function.

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 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, as shown in Fig. 13(a). 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. The zig-zag technique and our `fill` function are compared in Fig. 13 for an annulus-shaped part cross-section. Figure 12(b) shows the main steps implemented in the `fill` function.

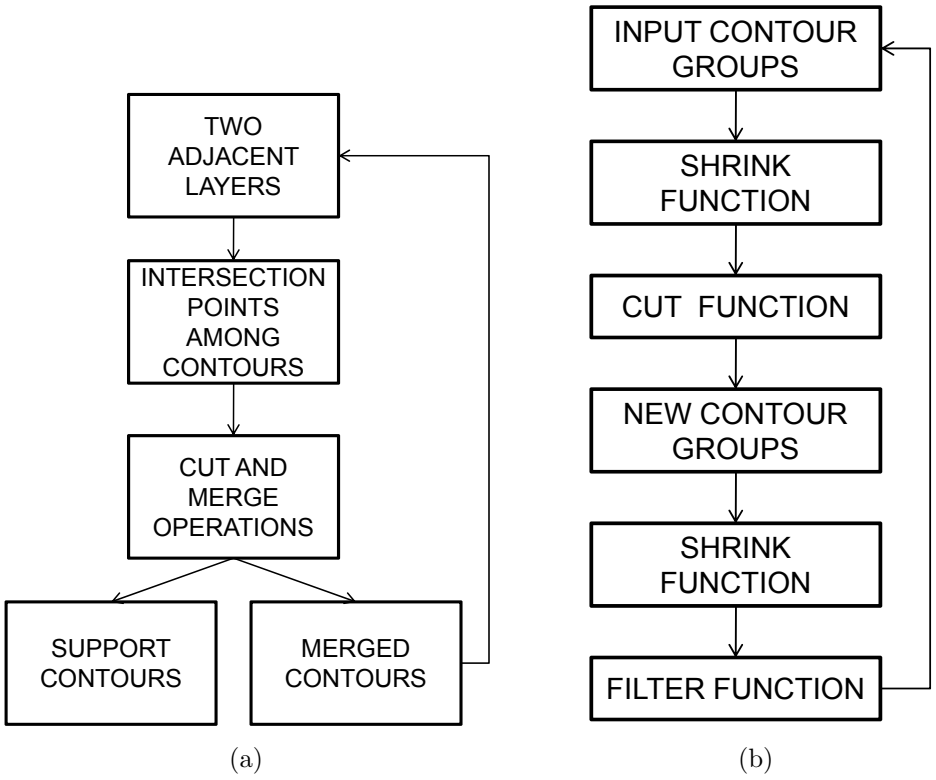


Figure 12: Function flow diagrams: (a) for the `support` function; (b) for the `fill` function

Before filling paths can be found, the bounding contours must first be shrunk 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.

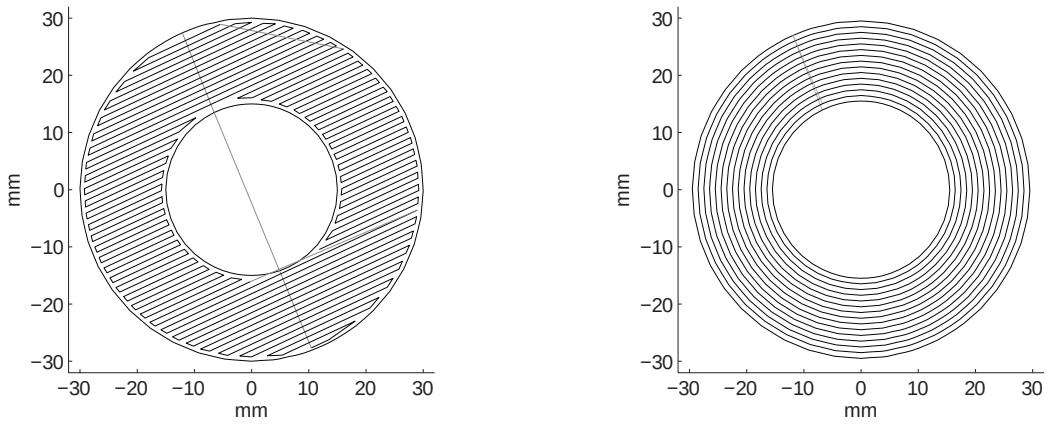
For layer l_i, each shrunk contour group is considered, by recalling each entry in the cell array $\mathbf{A}_i$. Every group is input to the `fill` function together with a threshold value t_h, which allows the user to specify the minimum distance between filling path points. Note that t_h needs to 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's controller are limited,[5] so it is desirable to have as few points as possible to avoid processing delays.

Our `fill` function first calls a `cut` function in order to recognize whether the contour to be shrunk is too narrow at any point, or if it is too close to inner contours. When necessary, cut-and-merge operations are performed to define new bounding contours. Note that examples of both of these cases can be seen in Fig. 14(a).

To create the filling paths, `shrink` is used to shrink the outermost contours of each

[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.



(a) (b)

Figure 13: Contour filling techniques: (a) the zig-zag technique; (b) our `fill` function (grey lines indicate non-depositing paths)

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 `cut` or `filter`.

5.1 The `shrink` Function

The `shrink` function is applied to a single contour as follows. We define segment $g_{i,j}$ as the line joining $\mathbf{p}_i$ to $\mathbf{p}_j$ on the contour being considered, so that segments of the form $g_{j-1,j}$ are part of the contour, and all other segments divide the contour into two pieces. We define $\mathbf{v}$ as the vector that bisects the angle formed by g_1 and g_2. Contour point $\mathbf{p}_i$ is considered along with the two contour segments g_1 and g_2 that connect to $\mathbf{p}_i$. If the contour is to be shrunk 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

$$r = \frac{s}{\sin(\sigma/2)} \tag{5}$$

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 shrunk, respectively, because of the order of contour points imposed in Section 2.3. A point $\mathbf{p}'_i$ on the shrunk contour is given by

$$\mathbf{p}'_i = \mathbf{p}_i + \mathbf{v} \tag{6}$$

5.2 The `cut` Function

The `cut` function considers the bounding contour γ^r in each contour group. If the distance between $\mathbf{p}_i$ and segment $g_{j-1,j}$ of γ^r is less than t_h, the contour is divided into two new contours by segment $g_{i,j}$. The procedure is repeated for these two new contours, and the iterative process continues until no narrow parts are present in any countour. All cut contours that are too small are discarded.

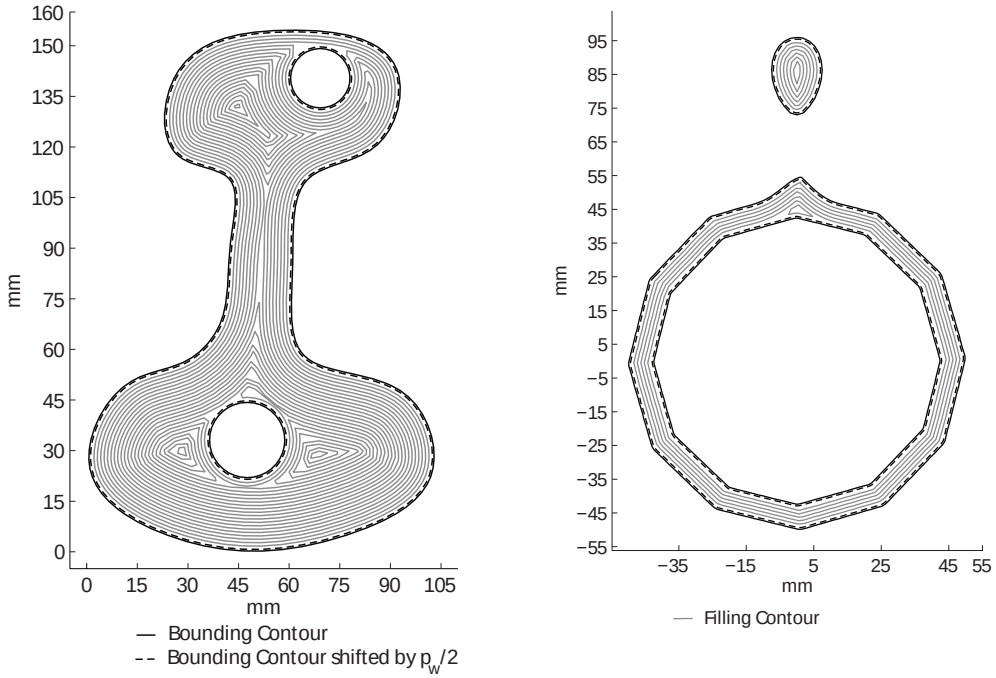
If γ^r is cut into more than one contour, then each cut contour $\gamma^{r'}$, along with the inner contours γ_s^r of γ^r, are input into `group` to create new contour groups. If $\gamma^{r'}$ is the only outer contour, it forms a new contour group along with contours γ_s^r. Then, $\gamma^{r'}$ is compared contours $\gamma_s^{r'}$ to determine if additional cut operations are necessary. If the distance between point $\mathbf{p}_i$ in one contour and segment $g_{j-1,j}$ in another is less than t_h, the cutting segment $g_{i,j}$ is stored in an array. Once all of the cutting segments for $\gamma^{r'}$ have been found, it is divided into many smaller pieces. Then, $\gamma^{r''}$ is defined as one of the cut contour pieces of $\gamma^{r'}$.

The contour piece of $\gamma_s^{r'}$ defined by the same cutting segments as $\gamma^{r''}$ is merged with it, forming a new closed contour, $\gamma^{r'''}$. Every new contour is filtered in order to delete narrow parts and, if it is too small, it is discarded. Then, if there is more than one cut contour, new contour groups are obtained by applying the `group` function to each outer contour, $\gamma^{r'''}$, and all contours contained by $\gamma^{r'}$ except inner contours $\gamma_s^{r'}$. When only one outer contour remains after filtering, it forms a new contour group along with all contours contained by $\gamma^{r'}$, except inner contour $\gamma_s^{r'}$. If a new group contains only one contour, the `cut` function completes, otherwise it restarts and is applied to the new contour group.

5.3 Filling Path Results

In Fig. 14, the filling paths generated with the `fill` function for two different part slices are shown. Figure 14(a) shows a slice of a dumb-bell shaped part with two holes, while Fig. 14(b) shows a slice of a beer mug. For the beer mug, $t_h = 1.8$ mm, while for the dumb-bell, $t_h = 2$ mm. Figure 15(a) shows a slice of a circular part with one hole, while Fig. 15(b) shows a hexagonal part. For the last two cases, the optimal value for t_h is found to be 2 mm. If t_h is too big, some areas will remain unfilled, while if t_h is too small, filling paths can have abrupt changes in direction. Figure 16 shows filling paths for a slice containing several circular nested contours, demonstrating the capability of the algorithm to handle cases where nested contours are present. A threshold value of 1.8 mm is used in this case. For all filling paths shown in this section, p_w is set to 1 mm.

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, where a PC with an Intel Core 2 Duo CPU @ 2.50 GHz and 3 GB of RAM is used. 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 cross-sections shown in Fig. 13, `fill` is clearly superior to the zig-zag technique. For the beer mug slice shown in Fig. 14(b), `fill` is also superior, though for the dumb-bell slice shown in Fig. 14(a), the advantages of using `fill` are less apparent, since there are some abrupt changes in direction.



(a) (b)

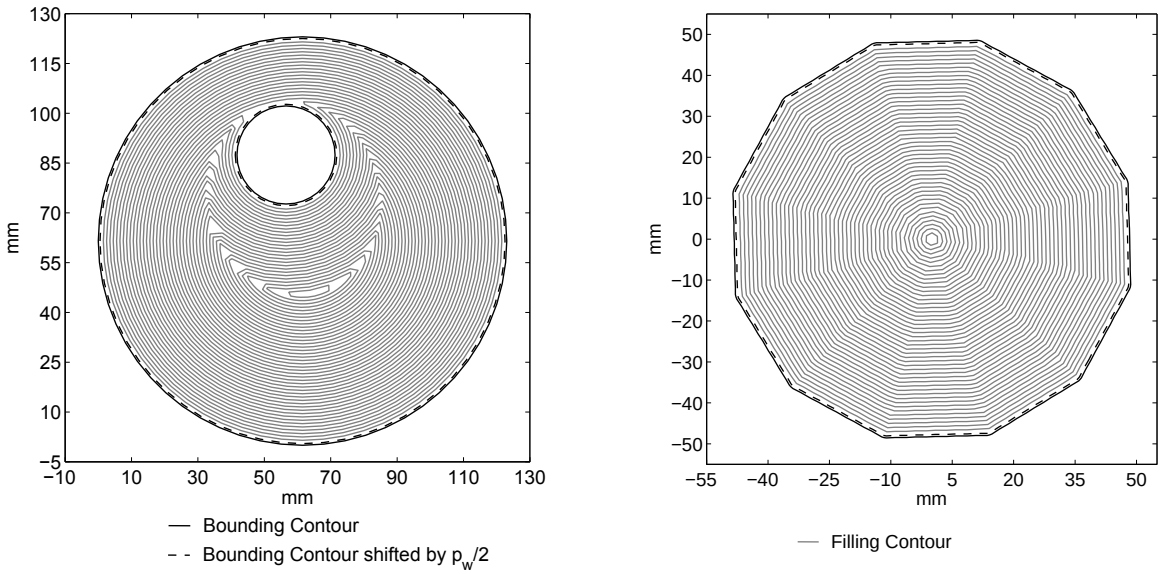
Figure 14: Contours with filling paths for part cross-sections: (a) a dumb-bell shaped part; (b) a beer mug

6 Exporting Data and Defining Non-Depositing Paths

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.

Table 2: Computation times for calculating all build paths for a martini glass: height=120 mm; layer height = 1 mm; path width = 1 mm

Case	Time (s)
Zig-zag	18
`fill` function	55
`fill` function with support structure	737



(a) (b)

Figure 15: Contours with filling paths for part cross-sections: (a) a circular part with a hole; (b) a hexagonal part

6.1 Non-Depositing Paths

To ensure high part-accuracy, constant speed along depositing paths is desired. This means that the velocity vector of the end-effector should be tangent to the contour at at the starting- and end-points. Additionally, for the first and last point of a data set read-in by the Cobra, the end-effector will have zero velocity. Finally, no more than 1000 points should be imported at a time to avoid significant processing delays. To address these issues, tangent, non-depositing path segments a_1, a_2, and a_3, are inserted before the starting point and after the end-point of each contour γ, as shown in Fig. 17(a). Points $\mathbf{q}_1$ and $\mathbf{q}_n$, which define these segments, are calculated as

$$\mathbf{q}_1 = \mathbf{p}_1 + 10p_w \frac{\mathbf{p}_1 - \mathbf{p}_2}{\|\mathbf{p}_1 - \mathbf{p}_2\|} \qquad \mathbf{q}_n = \mathbf{p}_n + 10p_w \frac{\mathbf{p}_n - \mathbf{p}_{n-1}}{\|\mathbf{p}_n - \mathbf{p}_{n-1}\|} \tag{7}$$

6.2 Blending Curves for Non-Depositing Paths

An abrupt change in direction could result at two points along the non-depositing path segments, as can be seen in Fig. 17. Therefore, these two segments are blended with circular arcs, as shown in Fig. 17(b). If (a) $|\alpha_1| < \phi_{min}$ or (b) $|\alpha_2| < \phi_{min}$, where $\phi_{min} = 1$ rad, an additional point q_{new} is inserted between $\mathbf{q}_n^i$ and $\mathbf{q}_1^{i+1}$ according to

$$\text{(a)} : \mathbf{q}_{new} = \mathbf{q}_n^i + 4p_w \frac{\mathbf{Q}(\beta_1)\mathbf{v}}{\|\mathbf{v}\|} \qquad \text{(b)} : \mathbf{q}_{new} = \mathbf{q}_1^{i+1} - 4p_w \frac{\mathbf{Q}(-\beta_2)\mathbf{v}}{\|\mathbf{v}\|} \tag{8}$$

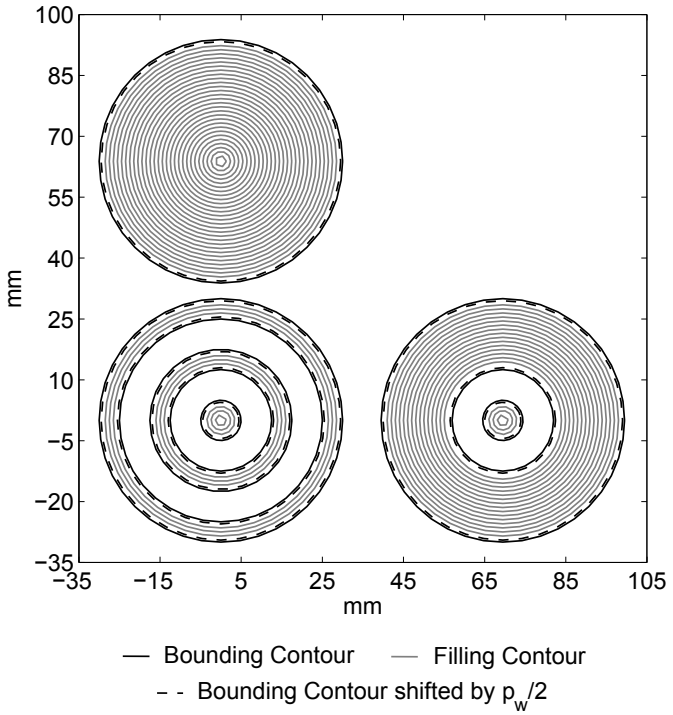


Figure 16: Contours with filling paths for part cross-sections: circular nested contours

where $\mathbf{v} = \mathbf{q}_1^{i+1} - \mathbf{q}_n^i$, $\beta_i = \text{sgn}(\alpha_i)(\phi_{min} - |\alpha_i|)$, and $\mathbf{Q}(\beta_i)$ is the rotation matrix for β_i.

Next, the circular arc blending curves are found. The centre of the blending curve is located on the line which bisects the angle formed by the two segments, at a distance ρ_i from their common point, given by

$$\rho_i = \frac{R}{\sin(\alpha_i/2)} \tag{9}$$

where α_i is the angle formed by the two segments and $R = 1.5$ mm is the desired blending curve radius.

Subsequently, the number of supporting points n_p for each circular arc is computed. Since the distance between two successive points must be greater than p_w, we have

$$n_p = \left\lfloor \frac{R(\pi - \alpha_i)}{p_w} \right\rfloor \tag{10}$$

Of course, when $\alpha_i \approx \pi$, we will obtain $n_p = 0$ because the transition between the two segments is smooth enough already and a blending curve is unnecessary.

An additional technique for computing blending curves was also considered. A software package written in `C` called `cursyn`, was previously developed by one of the authors to synthesize non-parametric cubic spline blending curves [12]. The advantage of using these blending curves is that `G2` (curvature) continuity is maintained at the blending points, so robots and other machines will follow the synthesized trajectory more accurately. For circular arc blending curves, the curvature is equal to the radius along the blending curve, while it is infinite along the blended segments if they are straight.

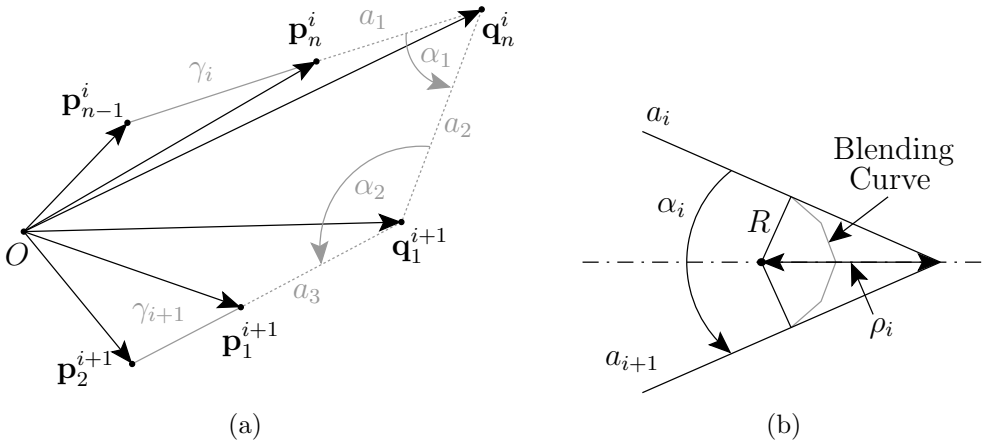


Figure 17: (a) Non-depositing segments in between contours γ_i and γ_{i+1}; (b) Blending curve for non-depositing segments

For a number of reasons, we have opted to use the circular arc blending curves rather than blending curves synthesized with `cursyn`. Firstly, the blending curves we are computing are for non-depositing paths, so they need not be extremely smooth: only smooth enough to prevent abrupt changes in direction. Secondly, the continuity along the blending curves themselves is limited because of the minimum point spacing required for the V+ trajectory generator. Thirdly, using `cursyn` would greatly increase the complexity and computational time required for the Matlab function currently being used. For a typical part, `cursyn` would need to be called several thousand times. Lastly, to ensure convergence and obtain reasonable blending curves, in many cases the problem must be reformulated before it is input into `cursyn`, by calculating several points between the two blended segments.

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, the threshold value used for the filling t_h, and an option number in order to determine which operations the algorithm must perform (slicing the CAD model, filling object areas, or build support structures).

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 with a standard Matlab distribution, and may be re-distributed free of charge to others who only need to run standalone executables. Matlab Compiler 4.9 supports all Matlab 7 features, most of the toolboxes, as well as user-developed GUIs.

(a) (b)

Figure 18: (a) A beer mug with its handle support; (b) the final object after support removal

8 Conclusions

We reported on a new path-planning algorithm, composed of several functions, to be used by 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. In fact, a specific function is able to find a hierarchy among nested contours, and can obtain contour groups bounding each part area. A support structure function has been written to find supports for a model, if necessary, by finding intersections among contours on adjacent layers, and performing cut-and-merge operations. This function has been 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, though it is more complex in terms of the computational time. The filling function has been tested on circular, hexagonal, and dumb-bell shaped parts; some of these shapes have one or multiple inner holes, which lead to nested contours. Non-depositing segments have been inserted at the starting- and end-points of each contour to reduce speed variations along depositing paths, and minimize errors in building the object. Finally, by using the Matlab Compiler, a stand-alone executable can be generated from the code and executed by users who do not have Matlab installed.

Experimental tests will be done using the trajectories generated by the path-planning algorithm. The algorithm will be optimized based on the quality of the ice structures built and the performance of the Adept Cobra 600 robot when following the synthesized trajectories.

Figures 18 and 19 show some objects built a previous rapid prototyping system developed by the authors; these models will be compared to those produced using the discussed path-planning algorithm. Figure 18(a) shows a beer mug and the support structure made for its handle, while in Fig. 18(b) the beer mug is shown after the support structure has been removed. Figure 19 shows a martini glass after supports are melted.

Figure 19: A martini glass after removing supports

9 Acknowledgements

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

Bibliography

[1] E. Barnett, J. Angeles, D. Pasini, and P. Sijpkes. Robot-assisted rapid prototyping for ice structures. to be presented at *IEEE Int. Conf. on Robotics and Automation*, Kobe, Japan, May 2009.

[2] R.H. Crawford and J.J.Beaman. Solid freeform fabrication. *IEEE Spectrum*, 36(2):34–43, 1999.

[3] S.H. Choi and K.T. Kwok. A tolerant slicing algorithm for layered manufacturing. *Rapid Prototyping Journal*, 8(3):161–179, 2002.

[4] P. Haipeng and Z. Tianrui. Generation and optimization of slice profile data in rapid prototyping and manufacturing. *Rapid Prototyping Journal*, 187-188:623–626, 2007.

[5] S. Allen and D. Dutta. Determination and evaluation of support structures in layered manufacturing. *Journal of Design and Manufacturing*, 5(3):153–162, 1995.

[6] R.C. Luo, Y.L. Pan, C.J. Wang, and Z.H. Huang. Path planning and control of functionally graded materials for rapid tooling. In *IEEE Int. Conf. on Robotics and Automation*, pages 883–888, Orlando, FL, May 2006.

[7] K. Chalasani, L. Jones, and L. Roscoe. Support generation for fused deposition modeling. In *6th Solid Freeform Fabrication Symposium*, pages 229–241, Orlando, FL, 1995.

[8] X. Huang, C. Ye, S. Wu, K. Guo, and J. Mo. Sloping wall structure support generation for fused deposition modeling. *Int. Journal of Advanced Manufacturing Technology*, 2008. DOI: 10.1007/s00170-008-1675-2.

[9] R.C. Luo, C.L. Chang, J.H. Tzou, and Z.H. Huang. Automated desktop manufacturing: Direct metallic rapid tooling system. In *IEEE Int. Conf. on Robotics and Automation*, pages 584–589, Barcelona, Spain, May 2005.

[10] H. Chen, N. Xi, W. Sheng, Y.Chen, A. Roche, and J. Dahl. A general framework for automatic CAD-guided tool planning for surface manufacturing. In *IEEE Int. Conf. on Robotics and Automation*, pages 3504–3509, Taipei, Taiwan, September 2003.

[11] A. Xu and L.L. Shaw. Equal distance offset approach to representing and process planning for solid freeform fabrication of functionally graded materials. *Computer-Aided Design*, 37:1308–1318, 2005.

[12] C.P. Teng, S. Bai, and J. Angeles. Shape synthesis in mechanical design. *Acta Polytechnica*, 47(6):56–62, 2007.