A robot programming environment based on free-form CAD modelling.

Colin G. Johnson.
Department of Computer Science,
University of Exeter,
Exeter, EX4 4PT, England, U.K.

and

Duncan Marsh.
Department of Mathematics,
Napier University, 219 Colinton Road,
Edinburgh, EH14 1DJ, Scotland, U.K.

Abstract

This paper presents the mathematical and computational foundations of a robot programming environment embedded within a CAD *system. The key ideas behind this system is that it will work offline, it will allow a high-level of task abstraction and it will be usable by designers and engineers who have a good knowledge of the desired task but only a basic grounding in robot engineering. In this paper we begin with a discussion of how robot workspace can be modelled using free-form* CAD *design concepts. The core of the paper is concerned with the application of these to well known problems of collision detection and path planning, showing how algorithms developed in* CAD *can be applied to these new problem-areas in an efficient way. In the closing section we use these ideas to consider the development of new, graphically-based, robot programming systems.*

Much work has been carried out on designing systems which attempt to simplify the process of robotic automation, making it easier for a robot programmer to prepare a robot to carry out a task. The important aspects of such automation are that it much work off-line, allowing programs to be prepared whilst the robot is engaged on another task, and that it must work at a high-level of abstraction—the intelligence about the fine detail of the robot's mechanics needs to be kept within the machine, liberating the designer to work on the task design not the fine-details of the programming.

If this work is to find its way into industrial practice, then it must be grounded in techniques which are currently used in industry, so that skills learned in other industrial design fields can be transferred to robot programming. Our perspective here is to incorporate robot modelling into computer-aided design systems, and to represent the complicated geometry of the workspaces by trimmed B-spline curves and surfaces and generalizations thereof.

1. A model of workspace.

This section gives a brief outline of the model which we have developed for robot manipulator workspace. Further details are given in our earlier papers [7, 9, 8].

1.1. B-spline free-form design.

A key concept in contemporary computer-aided design systems is the existence of *free-form design* systems. In geometric design these is a need not just for simple shapes, such as lines, circles and tori, but also for more general smooth shapes. This work, stemming from the work of Bézier and others in the automobile industry (see [5]), has led to a large body of theoretical and practical results on free-form design.

In this paper we make use of a free-form representation known as NURBS (non-uniform rational B-splines). To design a NURBS curve or surface, the designer manipulates interactively a number of *control parameters*, the most important of which are the *control points*. By adjusting the position of the control points, the surface generated is changed in a geometrically intuitive way. An example is shown in figure 1.

Mathematically the NURBS-curve is a piecewise polynomial curve, and the NURBS-surface a tensor product surface

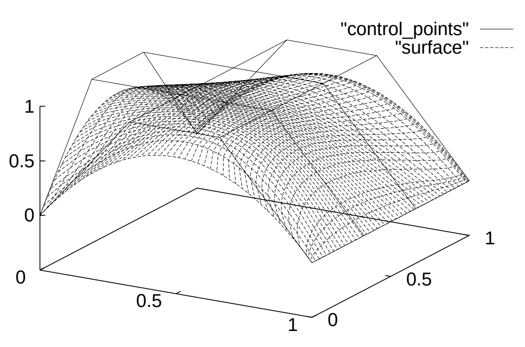


Figure 1. Designing a B-spline surface.

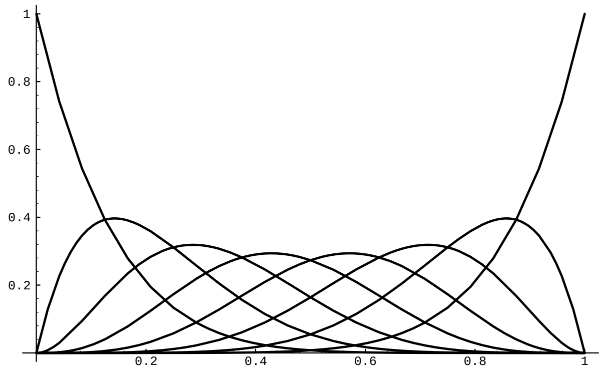


Figure 2. B-spline basis functions.

over piecewise polynomial bases. The curve is of the following form

$$\mathbf{x}(u) = \frac{\sum_{i=0}^{n} w_i \mathbf{P}_i N_{i,p}(t)}{\sum_{i=0}^{n} w_i N_{i,p}(t)} \tag{1}$$

Where $\mathbf{P}_i$ are a set of points called *control points*. The w_i are a set of *weights*, one corresponding to each point. By changing these weights the shape of the curve can be modified [14]. Mathematically the weights can be thought of as the fourth coordinate in a homogeneous coordinate system, defining the projection of a 4-dimensional non-rational space curve into 3-dimensional space [5]. The $N_{i,p}(t)$ are the B-spline rational basis functions, defined recursively by

$$N_{i,0}(t) = \begin{cases} 1 & \text{if} \quad t_i \le t < t_{i+1} \quad \text{and} \quad t_i < t_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$$N_{i,p}(t) = \frac{t - t_i}{t_{i+p} - t_i} N(t)_{i,p-1} + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} N(t)_{i+1,p-1}(t) \tag{3}$$

Here the $t_0, \ldots, t_n$ is a *non-uniform knot vector* which is a list of non-decreasing numbers, where the first and last numbers are repeated k times, where k is the order of the curve. We define p to be the degree of the curve (i.e. $p+1 = k$). The B-spline consists of a rational linear combination of these basis functions (an example of which are illustrated in figure 2), and forms a piecewise polynomial function over the interval spanned by the knots.

There are a number of reasons why this type of curve is use in CAD systems (see [5, 15] for details). Firstly it is possible to interactively modify the curve by adjusting the control points and other control parameters, in a way which is geometrically intuitive. A second reason is the existence of powerful geometrical algorithms which act on these curves. Below we shall make use of the *subdivision* algorithm, which takes a B-spline curve and gives control polygons corresponding to each half of the curve. This is a powerful tool for rendering the curve, as the curve can be rapidly broken down into approximate straight-line segments. A third advantage is that there are valuable geometrical properties of the form, for example the curve always lies within the *convex hull* of its control points. Finally there are a wide spectrum of design tools which can be used for B-spline curves, for example it is simple to design circles, straight lines, and free-form blendings between them.

We can form *tensor products* of NURBS to produce surfaces, volumes and hypervolumes in NURBS form. This involves taking a topologically rectangular/cuboidal grid of control points, and creating piecewise polynomial functions in each direction along the lines of the rectangle. A NURBS surface with its control polygon is illustrated in figure 1. The formula for the general n-variate NURBS mapping is given by

$$\mathbf{x}(u_1, \ldots, u_k) = \frac{\sum_{i_1=0}^{n_1} \cdots \sum_{i_k=0}^{n_k} \mathbf{P}_{i_1,\ldots,i_k} w_{i_1,\ldots,i_k} N_{i_1,p_1}(u_1) \ldots N_{i_k,p_k}(u_k)}{\sum_{i_1=0}^{n_1} \cdots \sum_{i_k=0}^{n_k} w_{i_1,\ldots,i_k} N_{i_1,p_1}(u_1) \ldots N_{i_k,p_k}(u_k)} \tag{4}$$

Using these mappings we can form the image of a high-dimensional space (such as the configuration space [11] of a manipulator) in a lower dimensional space (such as $\mathbb{R}^3$). This approach is followed below.

1.2. Kinematic functions as B-splines.

One important aspect of kinematics is studying the space occupied by a robot as it moves. This allows us to examine potential collisions of the robot with its environment, allowing the avoidance of such collisions and the coordination of

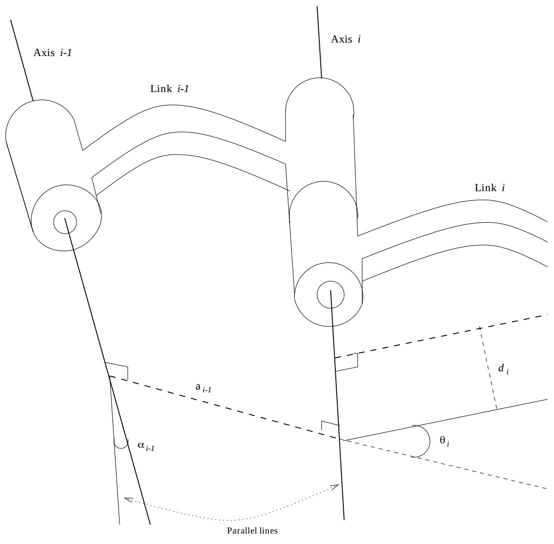


Figure 3. Denavit-Hartenberg parameters.

the robot's motions with other machines. Mathematically we consider two mappings. Firstly the *workspace occupancy mapping* ω_i, $(i = 1, \ldots, d)$, where d is the number of joints, known as the *degrees of freedom* of the mechanism.

$$\omega_i : \mathcal{C} \times \mathcal{P}_i \to \mathbb{R}^3 \tag{5}$$

This mapping takes a configuration $\mathcal{C}$ and a point on the ith link-surface $\mathcal{P}_i$ and maps it to the point occupied by that point. We are also interested in the volume swept out by a particular motion $m : [0, 1] \to \mathcal{C}$. This gives a composite mapping $\Omega_i = \omega_i \circ m$

$$\Omega_i : [0, 1] \times \mathcal{P}_i \to \mathbb{R}^3 \tag{6}$$

The image of this mapping is the volume of space swept out as the robot moves through the motion m.

A well-known way of notating the kinematics of manipulators is the *Denavit-Hartenberg* notation [2, 4]. This takes a set of axes (one for each link) and specifies the connections between these axes by four parameters, $a_i, \alpha_i, d_i, \theta_i$, one of the latter two being variable to represent the ability of the link to extend or rotate respectively (see figure 3).

We have demonstrated [9] that given a description of an open-chain mechanism in Denavit-Hartenberg form, and a description of the physical surface of the link in terms of NURBS, we can generate multivariate NURBS representing all of the functions Ω_i, ω_i for $i = 1, \ldots, d$, where d is the number of degrees of freedom of the robot.

2. Applications.

The focus of this paper is on applications of the theory explained above. This revolves around the geometrical problems in robotics which have application in task automation. In order to create a system which can be programmed at a high level of task abstraction within a CAD system, there is a need for problems such as collision detection and path planning to be embedded into the CAD environment. In this section we show how the work detailed above using B-splines facilitates this.

2.1. Collision Detection.

The first problem which we study is the collision-detection problem—given a motion m of the robot will it collide with obstacles within its environment. Here we outline an algorithm which takes the occupancy functions Ω_i corresponding to a motion m, and then subdivides Ω_i and the obstacles to check if any subregions of $\Omega_i(\mathcal{C})$ are occupied by obstacles. Linked-lists are used to structure the data in an efficient way.

```
Begin
Create the occupancy functions $\Omega_i$
Remove all obstacles that are impossible for the robot to
  reach in *any* configuration
For  $(i = 1; i \leq degrees\_of\_freedom; i++)$
     Create an linked-list of obstacles *obs_list*
     Create a pair $p := (obs\_list, \Omega_i)$
     Push $p$ onto an empty linked-list *main_list*
     integer *count* $:= 0$
     While *main_list* is non-empty and *count* $<$ *tolerance*
           pop a pair $\rho := (current\_list, current\_patch)$
             from *main_list*
           *pbb* := BoundingBox(*patch*)
           While *current_list* is non-empty
                 pop a patch *obstacle* from *current_list*
                 *obb* := BoundingBox(*obstacle*)
                 Test *obb* and *pbb* for intersection
                 If there is an intersection push *obstacle*
                   onto a list *temp_obstacles*
                 EndIf
           EndWhile
           If there have been any intersections
                 Subdivide *patch* into $patch_1$
                   and $patch_2$
                 Subdivide each obstacle on
                   *temp_obstacles*
                 Push $(patch_1, temp\_obstacles)$
                   onto the back of *main_list*
                 Push $(patch_2, temp\_obstacles)$
                   onto the back of *main_list*
                   onto the back of *main_list*
                 EndIf
           *count* $:=$ *count* $+ 1$
     EndWhile
     If *main_list* is empty there is no collision,
```

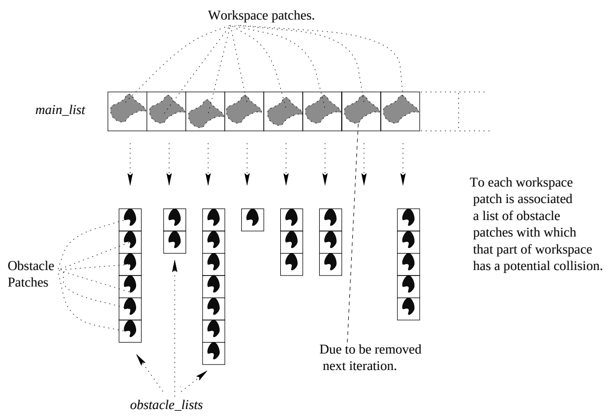

Workspace patches.
main_list
To each workspace patch is associated a list of obstacle patches with which that part of workspace has a potential collision.
Obstacle Patches
Due to be removed next iteration.
obstacle_lists

Figure 4. The collision-detection algorithm.

```
        so continue to link $i + 1$
    Else if tolerance has been reached and there are
        still things on *main_list*, then there is a collision
        Report(collision) and Stop
    EndIf
EndFor
Report(no collision)
End
```

We can adjust the tolerance to and desired number of subdivisions. In practice six or seven subdivisions in each direction is a useful limit (though of course this number of subdivisions will not be performed every time).

The `BoundingBox` procedure can be carried out in a number of different ways, for example rectangular bounding-boxes, spheres, oriented bounding boxes, swept spheres and convex hulls. This list is in rough order of complexity, the earlier ones being fast to calculate but offering a cruder approximation, the later ones offering tighter bounds but requiring more complex intersection algorithms. Variations on these bounding-box methods are commonly found in computer graphics—see [3, 6, 13, 18, 20] for details.

A graphical snapshot of the data-structures in the middle of this algorithm is shown in figure 4.

For collision detection in a dynamic environment, for example where the robot is moving amidst obstacles which are also moving, or where a system requires the coordinated motion of multiple robots, we can extrude the motion into a four-dimensional space-time, as in [1]. The mathematical details of this are given in [9, 8].

2.2. Path Planning.

More complicated problems are *path planning* and *accessibility checking*. Instead of using the computer to check human-designed paths, we create an algorithm which takes basic path requirements such as the initial and final configurations, and checks whether a path is feasible and if so finds such a path.

There are several approaches to this. Firstly we can consider working with individual paths and recombining these paths in a versions of a *genetic algorithm* [12]. This algorithm works by taking a wide variety of paths, specified as NURBS in configuration space, and iteratively splitting, combining and then selecting the best paths, gradually converging to a good path. This relies on a fast collision-checking algorithm such as the one outlined above.

```
Begin
Select a random set $P$ of $n$ paths in $\mathcal{C}$
Until (collision-free path found)
        Empty the set $P_{\text{new}}$
        For $(i = 1; i \leq 2n; i++)$
            Choose two members $p_1, p_2$ of $P$ at random
            Chop off a random number of control points
                from the beginning of $p_1$
            Chop off a random number of control points
                from the end of $p_2$
            Concatenate $p_1 \& p_2$ and put in a set $P_{\text{new}}$
        EndFor
        Test each member of $P_{\text{new}}$ for collision
        Rank the members of $P_{\text{new}}$ in order of
            amount of contact with obstacles
        Remove the $n$ members of $P_{\text{new}}$ with most obstacle contact
        Mutate a random selection from $P_{\text{new}}$
            by perturbing the control points
        $P := P_{\text{new}}$
        EndWhile
End
```

Another approach which captures the geometry of the situation in a better way is to trim away those regions $r \subseteq \mathcal{C}$ where $\omega_i(r) \cap O_j \neq \emptyset$ for some i, j, where O_j are the obstacles in the robot's environment, leaving behind those free-space regions in which the robot can move without fear. Here we apply a useful property of the B-spline representation—we have a natural subdivision structure (as in section 1.1) which allows us to calculate the image $\omega_i(r)$ easily and quickly. The key idea is illustrated in figure 5.

Essentially our algorithm works like this. Find the region $\omega_1(\mathcal{C})$, and carry out intersection tests using bounding-boxes as above. If there are any intersections, draw an isoparametric line through $\mathcal{C}$ splitting it into $\mathcal{C}_1, \mathcal{C}_2$ and carry this split into $\mathbb{R}^3$ by carrying out the subdivision algorithm on ω_1 to give $\omega_1(\mathcal{C}_1)$ and $\omega_1(\mathcal{C}_2)$. Then test these against the obstacles, throwing away any obstacles which don't collide. Continue until a free-space region is found, or until a tolerance is reached, then progress to the next link.

```
Begin
```

```
Create the occupancy functions $\omega_i$
Remove all obstacles that are impossible for the robot to
   reach in *any* configuration
Create an linked-list of obstacles *obs_list*
Create a triple $p := (\textit{obs\_list}, \omega_1, 1)$
Make $p$ the root of a tree *main_tree*
Mark this root as the *current_node*,
   and make it a grey node
While *main_tree* still contains grey nodes
      take the triple $\rho := (\textit{current\_list}, \textit{current\_patch}, \textit{depth})$
         from *current_node* of *main_tree*
      $\textit{pbb} :=$ BoundingBox(*patch*)
      While *current_list* is non-empty
            pop a patch *obstacle* from *current_list*
            $\textit{obb} :=$ BoundingBox(*obstacle*)
            Test *obb* and *pbb* for intersection
            If there is an intersection push *obstacle*
               onto a list *temp_obstacles*
            EndIf
      EndWhile
      If there have been any intersections
            Subdivide *patch* into
               $\textit{patch}_1 \ldots \textit{patch}_{2^i}$
            Subdivide each obstacle on
               *temp_obstacles*
            Place $(\textit{patch}_1, \textit{temp\_obstacles}, depth + 1)$
            on a new daughter-node of *main_tree*
               onto the back of *main_tree*
             . . .
            Place $(\textit{patch}_1, \textit{temp\_obstacles}, depth + 1)$
            on a new daughter-node of *main_tree*
               onto the back of *main_tree*
            EndIf
      If *depth+1* $=$ *tolerance*
         mark *current_node* as blocked and traverse the
            tree until another grey-node is found
      EndIf
      If no collisions were detected
         If *current_node* is on last link
         mark the *current_node* as free and traverse the
            tree until another grey-node is found
         Else create $2^{i+1}$ daughter nodes,
            initialized to $(\omega_{i+1}, \textit{obs\_list}, 1)$
      EndIf
EndWhile
End
```

We have used a tree-structure [17] to store information about the patches as we continue subdividing (see figure 6). Each node of the tree (corresponding to a patch of the occupancy mapping) is shaded `grey` (if more subdivision is needed), `blocked` if an obstacle prevents than patch of $\mathcal{C}$ from being accessed) or `free` if that region is known to be accessible. This has a number of advantages. One valuable property that we use here is the existence of algorithms which find connected regions in trees [16], thus giving an algorithm for checking whether there exists a path or not. Once we have found such a contiguous free-space region, then we can interpolate a NURBS path through it.

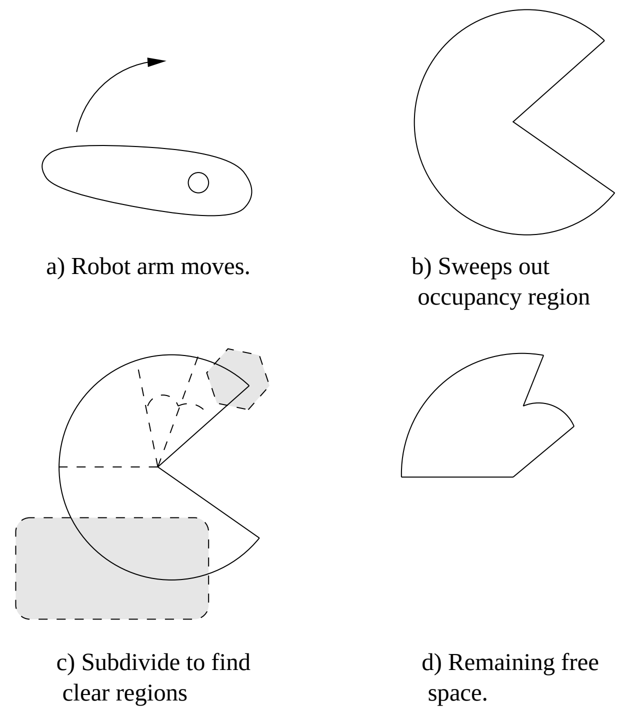

a) Robot arm moves.
b) Sweeps out occupancy region
c) Subdivide to find clear regions
d) Remaining free space.

Figure 5. Trimming away to find free-space.

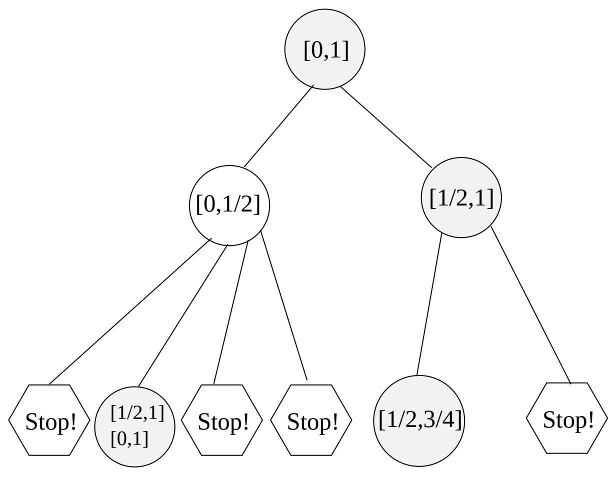
One major advantage of this (compared with, for example, [11]) is that the same structure works on any scale. If a large amount of space is free then these regions are marked off as free near the beginning of the algorithm, rather than being pointlessly further subdivided. Equally the algorithm concentrates on small regions where this is necessary, and the level of detail is decided automatically as the algorithm progresses—there is no need to set an initial level of desired detail.

We can extend the arguments about space-time swaths for motion in dynamic environments to the accessibility checking and path planning problems too.

3. Development of a system.

In the paper we have shown how to develop a NURBS model of robot workspace, and how to apply this model to a number of problems in automation. In these closing paragraphs we bring these ideas together and discuss how to embed this into intelligent robot programming systems.

A key to this lies in liberating robots from the constraints



Notes: figures in brackets refer to uppar and lower ranges of link angle, normalized to [0,1]. Shading indicates possibly blocked regions, open circles are free regions, hexagons are blocked regions.

Figure 6. Tree structure after the first few stages of the free-space algorithm.

of the designed environment. One exciting avenue to explore here is incorporating work in computer-vision and range sensing, and recent work by Wang and Wang [19] and Lavallée and Szeliski [10] use B-spline surfaces as the basis of visual reconstruction experiments, which strengthens our use of B-splines as a mathematical basis for the system.

Moving on from this we intend to develop a system whereby robots can be programmed in a wholly graphical environment, rather than graphical systems being used to test text-based programs [2]. This will build upon the work above, allowing the environment to be designed in a CAD system and desired positions and orientations of the robot indicated by interaction with a 3D model, drawing on the path-planning algorithms outlined in section 2.2 to hide the details of the robot's kinematics. Further on from this, we are working towards the development of a graphical robot programming system based on *constraints* on the motion of the robot given by *graphical analogies*—for example regions where the robot is not allowed to visit are indicated by virtual "walls", pressures towards or away from a region of workspace are indicated by virtual "springs", et cetera. It is intended that this will provide a flexible way of programming robots which can respond to variations in the robot environment.

References

[1] S. Cameron. Using space-time for collision detection : solving the general case. In K. Warwick, editor, *Robotics, Applied Mathematics and Computational Aspects*, pages 403–415. Clarendon/IMA, 1993.

[2] J. Craig. *Introduction to Robotics*. Addison-Wesley, second edition, 1989.

[3] A. del Pobil and M. Serna. *Spatial Representation and Motion Planning*. Number 1014 in Lecture Notes in Computer Science. Springer, 1995.

[4] J. Denavit and R. Hartenberg. A kinematics notation for lower-pair mechanisms based on matrices. *Journal of Applied Mechanics (Transactions of the ASME)*, June 1955.

[5] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, third edition, 1993.

[6] R. Featherstone. A hierarchical representation of the space occupancy of a robot mechanism. In J.-P. Merlet and B. Ravani, editors, *Computational Kinematics (INRIA, September 1995)*. Kluwer, 1995.

[7] C. G. Johnson and D. Marsh. Modelling robot manipulators in a CAD environment using B-splines. In N. Bourbakis, editor, *Proceedings of the IEEE International Joint Symposia on Intelligence and Systems*, pages 194–201. IEEE Press, 1996.

[8] C. G. Johnson and D. Marsh. Geometric models of robotic mechanism motions using multivariate B-splines. In preparation, 1997.

[9] C. G. Johnson and D. Marsh. Multivariate B-splines for modelling robot manipulator workspace. In M. Dæhlen, T. Lyche, and L. L. Schumaker, editors, *Mathematical Methods for Curves and Surfaces II*. Vanderbilt University Press, 1998.

[10] S. Lavallée and P. Szeliski. Recovering the position and orientation of free-form objects from image contours using 3D distance maps. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(4):378–390, 1995.

[11] T. Lozano-Pérez. A simple motion-planning algorithm for general robotic manipulators. *IEEE Journal on Robotics and Automation*, RA-3(3):224–238, 1987.

[12] M. Mitchell. *An Introduction to Genetic Algorithms*. Bradford Books, 1996.

[13] Q. Peng. An algorithm for finding the intersection lines between two B-spline surfaces. *Computer Aided Design*, 16(4), July 1984.

[14] L. Piegl. Modifying the shape of rational B-splines. part 1 : curves. *Computer Aided Design*, 21(8):509–518, 1989.

[15] L. Piegl and W. Tiller. *The $\mathcal{NURBS}$ Book*. Springer, 1995.

[16] H. Samet. Connected component labeling using quadtrees. *Journal of the Association for Computing Machinery*, 28(3):487–501, 1981.

[17] H. Samet. The quadtree and related hierarchical datastructures. *ACM Computing Surveys*, 16(2), 1984.

[18] T. W. Sederberg and S. R. Parry. Comparison of three curve intersection algorithms. *Computer-Aided Design*, 18(1):58–63, January/February 1986.

[19] Y. Wang and J. Wang. On 3D model construction by fusing heterogeneous sensor data. *CVGIP-Image Understanding*, 60(2):210–229, 1994.

[20] J. Yen, S. Sprach, M. Smith, and R. Pulleyblank. Parallel boxing in B-spline intersection. *IEEE Computer Graphics and Applications*, pages 72–79, January 1991.