Modelling Robot Manipulators with Multivariate B-splines.

Colin G. Johnson* and Duncan Marsh†

September 23, 1998

Abstract

In programming robot manipulators to carry out a wide variety of tasks it would be desirable to create a CAD system in which these tasks can be programmed at the task level, leaving the fine-grained detail of path planning and collision detection to the system. This paper describes the theoretical background to such a system, by providing a model in which robot motions are represented using multivariate B-splines, a standard representation for free-form shapes in the CAD environment. The paper also describes algorithms which take this representation and apply it to collision detection and path-planning.

1 Introduction.

Robots are typically programmed in one of two ways. The first of these is to guide the robot through a set of motions using a teach-pendant, which takes the robot out of the production line for the duration of the programming task. The second is to plan the motions using a robot programming language, possibly assisted by using a simulation package such as *CimStation* [1] in an iterative cycle of program-test-program-test-$\cdots$.

*Colin Johnson is a lecturer in the Department of Computer Science at the University of Exeter, The Old Library, Prince of Wales Road, Exeter, EX4 4PT, England.

†Duncan Marsh is a lecturer in the Department of Mathematics at Napier University, 219 Colinton Road, Edinburgh, EH14 1DJ, Scotland.

Much work (beginning with [2] and surveyed in [3, 4, 5, 6]) has come out of the artificial intelligence community over the last thirty years in increasing the level of automation of robot programming—that is hiding details of the programming process such as collision detection and and calculating the fine details of velocity and acceleration, liberating the programmer to concentrate on the task itself. In this paper we look at the mathematical and algorithmic foundation of how we can embed this style of robot programming in a CAD environment, and describe some algorithms which use these ideas to tackle collision detection and path-planning problems. This paper extends the work described in [7, 8].

2 B-spline curves and surfaces.

In this paper we make use of *B-spline* curve, surfaces and higher-dimensional shapes [9, 10]. These are a piecewise-polynomial representation used in free-form design, that is the design of arbitrary smooth shapes. The most popular kind of B-spline representation used in CAD is the NURBS (non-uniform rational B-spline) shape, which is easy to use for design and which allows the representation of circles, polyhedra and polynomial curves and surfaces.

The mathematical description of these shapes uses a set of *basis functions*, a typical set being illustrated in figure 1. A NURBS-curve consists of these basis functions combined linearly with respect to a number of *control points*, which are together termed a *control polygon*. The end-points of the control polygon describe the end-points of the curve, and the intermediate points are used as *shape parameters* to control the shape of the curve. Designing with these curves is therefore somewhere between using an interpolating function and freehand sketching.

The curve is described mathematically by the following formula

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

Where $\mathbf{P}_i$ are a set of control points. The w_i are an additional set of control parameters called *weights*, which in an intuitive sense are used to allow different points to have a different amount of influence on the shape of the curve [11]. More formally the w_i are the fourth coordinate in a homogeneous coordinate system, the three-dimensional curve being described mathematically

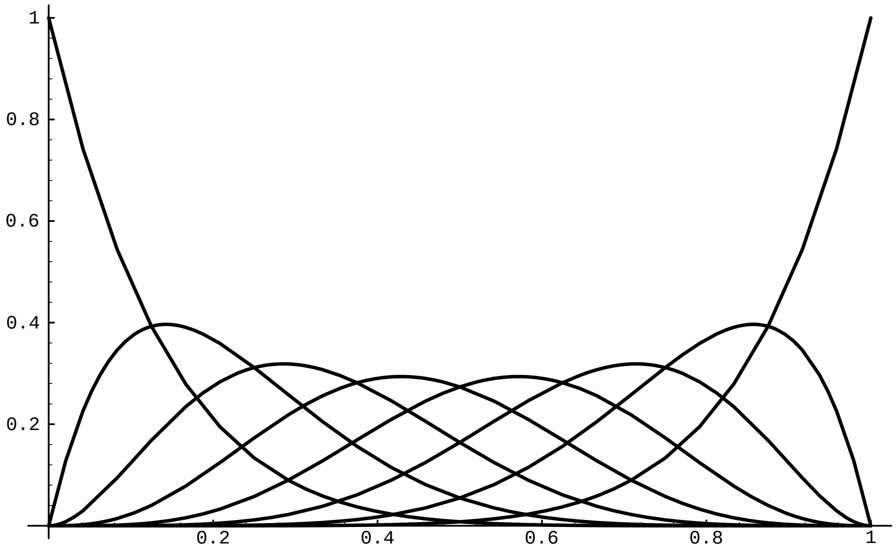

Figure 1: B-spline basis functions.

as the projection of a 4-dimensional non-rational curve into 3-dimensional space [12].

The $N_{i,p}(t)$ are the B-spline rational basis functions, defined recursively by

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

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

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. $p + 1 = k$). These are the basis functions described above and shown in figure 1. A typical NURBS-curve is illustrated in figure 2.

It is also possible to define higher-dimensional NURBS objects, for example surfaces (see figure 3). There are two ways of doing this, the one adopted

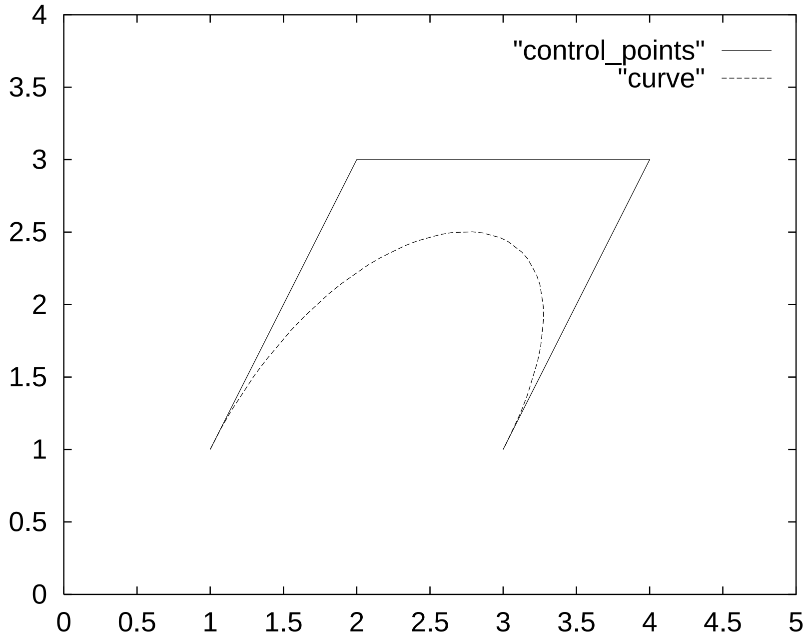

Figure 2: A B-spline curve.

here is to form the *tensor product* of k curves, the result being given by the following equation

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

where the $\mathbf{P}_{i_1,\ldots,i_k}$ are a topologically rectangular grid of points, with an associated set of weights $w_{i_1,\ldots,i_k}$. These points can lie in a space of dimensionality less than k, and we use this idea below to embed the configuration space [13] of a manipulator into the physical space $\mathbb{R}^3$.

There are two particular properties of B-spline shapes that we shall use here. The first is the *convex hull theorem* [12], which states that all points on the B-spline shape lie within the convex hull of the control polygon. The convex hull is an easily computed object [14], with an $O(n \log n)$ complexity algorithm for n 2-dimensional points, and at least an $O(n^2)$ complexity for 3-dimensional points. Therefore we have a powerful techniques for approximating complex curved objects with simpler polyhedral objects for intersection testing.

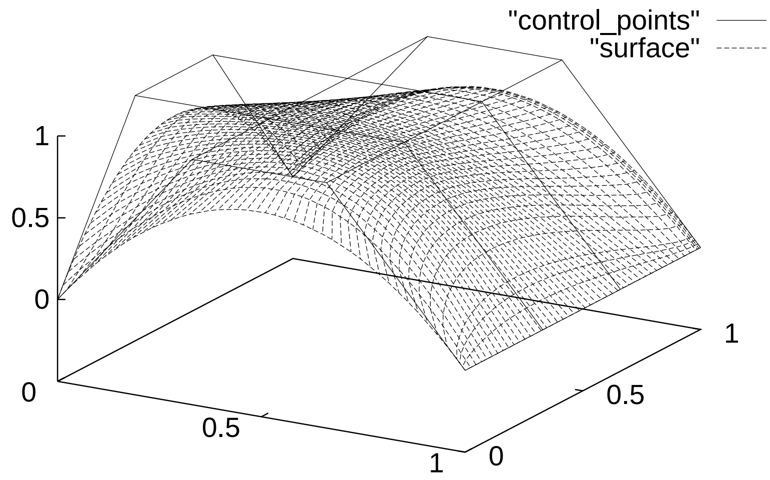

Figure 3: Designing a B-spline surface.

The second useful property here is the existence of a *subdivision* algorithm. By repeated linear interpolation along (in 2 dimensions) the edges of the curve, we can find an intermediate point on the curve, and also find two sub-control-polygons specifying the sub-curves on either side of that intermediate point. This can be extended to the subdivision of a NURBS-shape of any dimensionality by an isoparametric hyperplane of codimension 1. Details of these algorithms can be found in [12, 10].

This section has given a brief introduction to this large subject. More details can be found in [9, 10, 12].

3 A new model of workspace.

In this section we give mathematical constructions which give multivariate B-spline mappings to describe the space swept out when a robot moves. Models are described of both the space consisting of all possible locations for the robot in space (section 3.1) and the space swept out when a robot makes a given motion (section 3.2).

The robot arms commonly used in industry have an *open-chain* kinematic structure. Such mechanisms consist of a chain of *links* connected with *joints*. These joints are either *revolute joints* which rotate around an axis, or *prismatic joints* which move along an axis.

In order to specify the geometry and kinematics of such a robot needs three items of information. Firstly, the spatial position/orientation of the robot is given by a cartesian coordinate frame. Secondly, the physical geometry of the links is given by by a NURBS surface $\mathbf{S}_l$ for each link l (it is simple to extend this to several surfaces per link). Finally, we specify how the links are connected together, using the Denavit-Hartenberg notation—the standard notation used in kinematics [1, 15]. We describe this briefly as follows (see figure 4). We begin by taking a line ℓ_i through the axis of each joint of the mechanism, i.e. the axis that a link either rotates around (revolute joint) or slides along (prismatic joint). Each pair ℓ_i, ℓ_{i+1} is joined by their unique common perpendicular (unless they are parallel, in which case any common perpendicular will suffice). Next we specify the kinematic relationship between these links exactly using four parameters. Two of these parameters, the *link length* a_{i-1} and the *link twist* α_{i-1} specify the fixed relationship between the two axes forced by the physical link. The remaining two, the *link offset* d_i (which is variable for a prismatic joint) and the *joint angle* θ_i (which is variable for a revolute joint) specify the relationship between two adjacent links.

3.1 Workspace generation.

We use the Denavit-Hartenberg specification to generate a set of mappings $\omega_i : \mathbb{R}^2 \times \mathbb{R}^i \to \mathbb{R}^3$, where i ranges from $1, \ldots, d$, where d is the number of degrees of freedom of the robot. The function ω_i specifies the region of space occupied by the robot in a particular position. Consider the mapping $\omega_i : (u, v) \times (r_1, \ldots, r_i) \mapsto (x, y, z)$. This takes a value of the parameters (u, v) which specify a point in the domain of $\mathbf{S}_i(u, v)$, and $(r_1, \ldots, r_i)$, which specify the values of d_j (for $(j = 1, \ldots, i)$, when the jth link is prismatic) or θ_j (for $(j = 1, \ldots, i)$, when the jth link is revolute). The image (x, y, z) is a point in $\mathbb{R}^3$ which specifies where the point $\mathbf{S}_i(u, v)$ is found when the robot is in the position specified by $(r_1, \ldots, r_i)$.

The next stage is to give these mappings are given a NURBS structure. Place the control net for the surface $\mathbf{S}_1(u, v)$ as it would be when the robot is in its home position. Form the tensor product of $\mathbf{S}_1(u, v)$ with a motion-

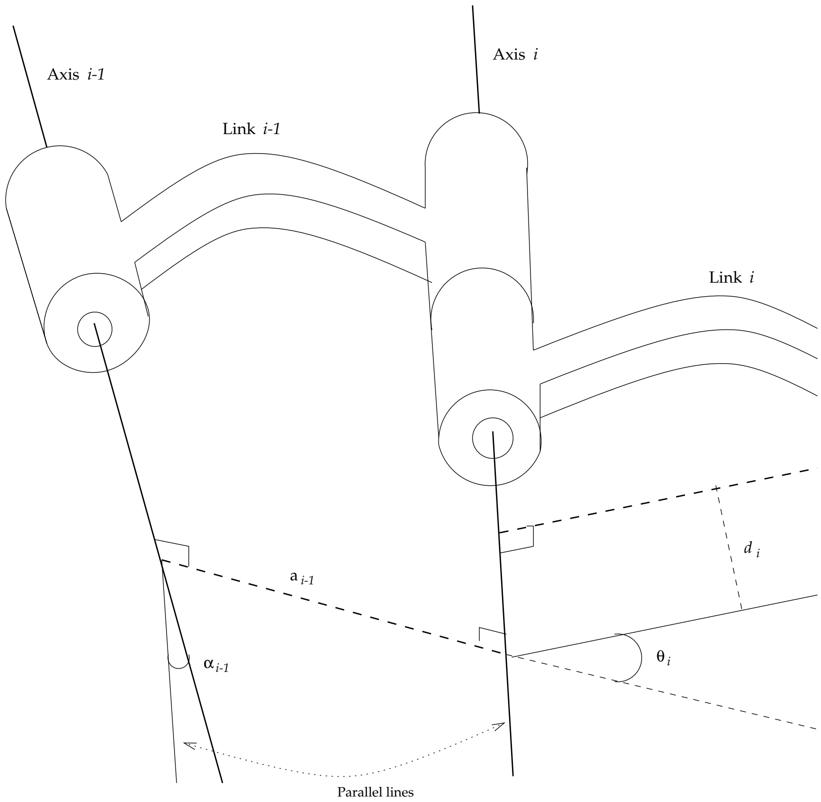

Figure 4: Denavit-Hartenberg parameters.

curve $C(r_1)$, which is an arc (ranging between the upper and lower limits of θ_1) of a unit circle around the base-axis if the joint 1 is revolute, and a NURBS line (ranging between the upper and lower limits of d_1) along the axis if the joint is prismatic.

The penultimate part of the algorithm is to give a basic structure on which to place the surface making up link 2. To do this a NURBS arc/line segment $D(r_1)$ is constructed having a radius/length a_{i-1}. The control net $\mathbf{S}_2(u, v)$ is placed at the base position, and then displaced by a_{i-1}, rotated by α_{i-1}, and finally rotated/translated by whichever of θ_i and d_i is fixed. Finally a tensor product between the line/arc $C(r_2)$ and the transformed $\mathbf{S}_2$ is formed, and a further tensor-product with $D(r_1)$ gives the 4-variable NURBS function $\omega_2(r_1, r_2, u, v)$. We repeat this process until the occupancy functions $\omega_n(r_1, \ldots, r_n, u, v)$ for all links have been generated.

3.2 Modelling specific motions.

In addition to the mappings for workspace generation we define mappings which give a NURBS model of the mapping which defines volume of space (or space-time) occupied by the robot during the execution of a given trajectory. More precisely for a given motion $\mathbf{M}$, specified as a NURBS path in configuration space [13], we define a function for each link $\Omega_i(M) : \mathbb{R}^2 \times \mathbb{R} \rightarrow \mathbb{R}^3$. This function takes a pair (u, v) specifying a point in the domain of the $\mathbf{S}_i(u, v)$ and a parameter t specifying the distance travelled along the motion $\mathbf{M}$, to obtain $\Omega_i(\mathbf{M}) : (u, v) \times t \mapsto (x, y, z)$, where (x, y, z) is the point occupied by the image of $\mathbf{S}_i(u, v)$ when the link is at the point in configuration space given by $\mathbf{M}(t)$.

We model this using a *geometric swept-volume algorithm.* This takes a template surface $\mathbf{S}(x, y)$ and moves it along a trajectory $\mathbf{T}(t)$ whilst also executing a local motion of the surface, producing a swept volume $\mathbf{V}(x, y, t)$ in space. We can express it thus

$$\mathbf{V}(x, y, t) = \mathbf{T}(t) + N(\mathbf{S}(x, y), t)$$

where N is a transformation of the surface with respect to its fixed position, which varies with changes in t. This has been used in planar kinematics, where the motion N was a multiplication of the control net of $\mathbf{S}(x, y)$ by control points in a of transformation matrices [16]. This allows us to calculate the volume swept out when we move the surface through space whilst simultaneously transforming the shape with respect to a moving coordinate frame.

We calculate these $\Omega_i(M)$ in two stages. In the first stage we take the link-surface $\mathbf{S}_i(u, v)$, placed with respect to a coordinate frame at the origin. If the ith joint is a revolute joint, we form a volume of revolution $\mathbf{V_i}(u, v, t)$ by forming a tensor product of S_i with an arc of a circle in NURBS form [10]. Similarly for a prismatic joint we tensor product the surface with a straight line along the axis to form a volume of extrusion. The length and parameterization of these lines and arcs are derived by reparameterizing a standard NURBS circle or line by a function specifying the motion of the link.

For the first joint this volume is $\Omega_1(M)$, the space swept out by S_1 as the first joint moves around a fixed coordinate frame. However for the other joints the axis itself is moving, so we have a second stage. Take a point at the ith joint and apply the rotation/extrusion to that, giving a NURBS-curve $\mathbf{T}(t)$ in space, having a knot vector which we shall call U_t. Then use degree-raising and knot insertion to equate U_t with U_r, the knot-vector of $\mathbf{V_i}(u, v, t)$ in the t direction. This produces a new set of control points for $\mathbf{T}(t)$ which we call $(T_0, \ldots, T_{n_t})$. Create a set of new points P_{ijk} from the control points V_{ijk} of $\mathbf{V}(u, v, t)$ and the control points T_i of $\mathbf{T}(t)$.

$$P_{ijk} = RV_{ijk} + T_i$$

Where R is the rotation matrix (an affine transformation) that carries the frame based at the origin into the Frenet frame moving along the curve. The desired swept surface $\mathbf{V}(u, v, t)$ is defined by

$$\mathbf{V}(u, v, t) = \sum_{k=0}^{n_t} \sum_{j=0}^{n_u} \sum_{i=0}^{n_v} P_{ijk} R_{i,d_u}(u) R'_{j,d_v}(v) R''_{k,d_t}(t)$$

Where $R_{i,d_u}(u)$, $R'_{j,d_v}(v)$ and $R''_{k,d_t}(t)$ are the non-uniform rational basis functions defined over U_u, U_v (the knot vectors of $\mathbf{S}(u, v)$) and U_t respectively.

Note that isoparametric surfaces for fixed t values of $\mathbf{V}(u, v, t)$ consist of $\mathbf{S}(u, v)$ transformed to an appropriate position along the curve. That the placement of these surfaces at the control points is sufficient to describe the entire motion follows from the the affine invariance property of B-splines [12].

3.3 Comments.

This representation restricts the motions allowed to those which can be represented in NURBS form. It could be well argued that this is not a *restriction* at all. Firstly, we can approximate any motion as accurately as desired using a NURBS path. Secondly, we have to design a motion using *something*, and NURBS, with their properties of local control, control over their smoothness and their ability to incorporate many other kinds of motion such as straight-line interpolants and circles [9] offer an intuitive and geometrically elegant method for this.

It can be seen that this can be extended to the case of creating a swept volume in four-dimensional space-time [17]. This is important for studying the interaction of a robot with other moving obstacles [18, 19, 20, 17], or attempting to detect self-intersections.

The main advantage of this representation is that it allows the motion, the shape of the links and the resultant swept volume to be represented in a common form, and has the added advantage that that form is a standard in CAD. Such advantages are not to be found in other swept-volume models of workspace such as [21].

4 Application to collision detection.

We now turn to applying this model to the collision detection problem. A robot executes a motion $\mathbf{M} : [0,1] \to \mathcal{C}$, where C is the configuration space of the robot. We recall the workspace mapping $\Omega_i(\mathbf{M}) : \mathbf{S_i} \times [0,1] \to \mathbb{R}^3$ from above, where $\mathbf{S_i}$ is the set of points on link i of the arm. Given a set of k obstacle surfaces $O_1, \ldots, O_k$ in $\mathbb{R}^3$ we say that there is a collision if the set $\{\Omega_i(\mathbf{M}) \cap \{O_1, \ldots, O_k\}\}$ is non-empty.

We calculate this collision by subdividing the obstacles and workspace image until either we are certain that there are no collisions or we have subdivided down to a certain level of tolerance and there are still collisions between approximating bounding regions. We use a linked-list structure to structure the data efficiently. The pseudocode for the algorithm follows, and the algorithm is summarized in diagrammatic form in 5.

`Begin`
Create the occupancy functions Ω_i
Remove all obstacles that are impossible for the robot to
  reach in *any* configuration
`For` $(i = 1; i \leq \mathit{degrees\_of\_freedom}; i++)$
    Create an linked-list of obstacles *obs_list*

```
    Create a pair p := (obs_list, Ω_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 ρ := (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,
      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
```

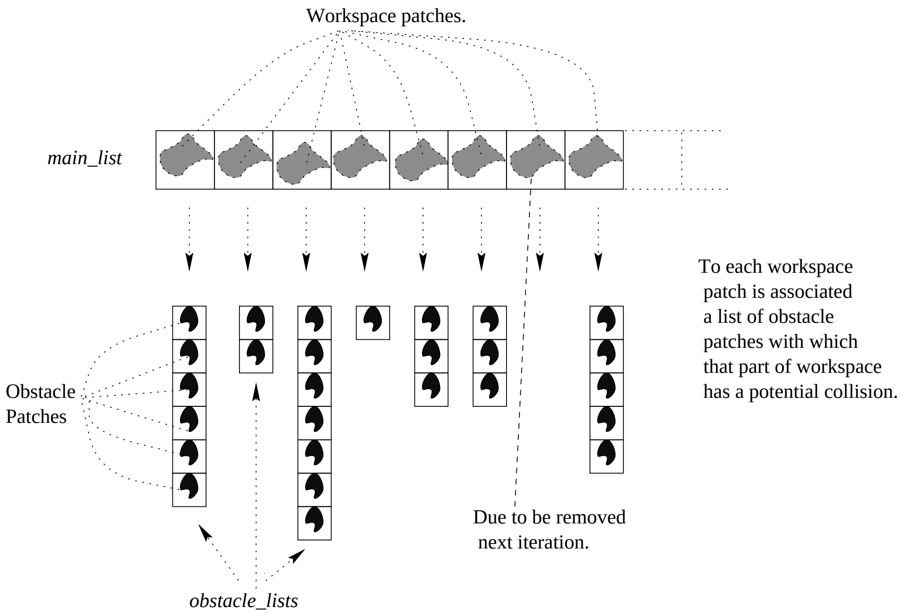

Figure 5: The collision-detection algorithm.

The advantage of this algorithm (over, say, [13]) is that the amount of subdivision is *adaptive*. Regions where there are many possibilities for collision get finely subdivided and approximated by very small bounding boxes, whilst broad free regions or obstacles irrelevant to the particular problem at hand are removed from (resp.) *main_list* and *obs_list* at an early stage of the algorithm.

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 [22, 23, 24, 25, 26] for details.

5 Path planning and accessibility checking.

In this section we describe two algorithms which can be used for *path-planning*. That is instead of using the computer to test human designed paths, we specify the robot and its environment and the algorithm automatically finds a free-space path between desired start and end points, or reports that this is not possible.

5.1 Using a genetic algorithm.

The first algorithm uses a kind of *genetic algorithm* [27, 28] for path planning. The algorithm begins by creating at random a large set of possible paths without considering whether these paths cause collisions or not. These paths are generated by selecting NURBS control points at random in configuration space, then adding the desired start and end point to the list. A standard Bézier-style knot vector is used. There then follows an iterative process, in which the first stage is to combine at random pairs of paths by clipping the control-point sets from each end, and concatenating these subsets together. This set of solutions is then ranked by total amount of obstacle contact and total length, such that a long path with large amounts of contact is ranked low whilst a short path with small amounts of obstacle contact is ranked high.

Here is the pseudocode for the algorithm.

`Begin`
Select a random set P of n paths in $\mathcal{C}$
Set a *tolerance* = maximum number of iterations which will
  be carried out before assuming that there is no collision-free path
`Until` (collision-free path found
    or until *tolerance* number of iterations)
    Empty the set P_{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_{new}
    `EndFor`
    Test each member of P_{new} for collision
    Rank the members of P_{new} in order of
      amount of contact with obstacles

and by control-polygon length
Remove the n members of P_{new} with
lowest ranking
Mutate a random selection from P_{new}
by perturbing the control points
$P := P_{\text{new}}$
`EndWhile`
`End`

5.2 Trimming the workspace.

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 regions where the robot is free to move. The NURBS structure is particularly valuable here, as we can apply the *subdivision algorithm* to split the obstacles and the robot's occupancy space into subregions . The basic idea is illustrated in figure 6.

Essentially our algorithm works like this. Find the region $\omega_1(\mathcal{C})$, and carry out intersection tests using bounding-boxes as in section 4 (see also [22, 23, 24, 25, 26]). 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 so many subdivisions have been done that it is safe to say that there is no possibility of a free-region being found within a certain tolerance. If there is a free region then investigate the free regions for the next link, and so on until the algorithm halts because it cannot find a free region for a given link or until it runs out of links having found a free motion for each link. It is then a comparatively simple task to interpolate a path through these free-space regions.

Here is the pseudocode for the algorithm.

`Begin`
Create the occupancy functions ω_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 := (obs_list, ω_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 ρ := (current_list, current_patch, depth)
          from current_node of main_tree
        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 ... patch_{2^i}
                Subdivide each obstacle on
                  temp_obstacles
                Place (patch_1, temp_obstacles, depth + 1)
                on a new daughter-node of main_tree
                  onto the back of main_tree
                 ...
                Place (patch_1, 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 (ω_{i+1}, obs_list, 1)
      EndIf
EndWhile
End
```

We have used a tree-structure [29] to store information about the patches as we continue subdividing (see figure 7). 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 allows us to use the algorithm of [30] to find a connected path through the various subdivided regions.

One major advantage of this (compared with, for example, [13]) 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.

6 Conclusions and future directions.

In this paper we have outlined some of the mathematical and computational background needed to develop a robot programming system based on free-from CAD modelling. In order to develop this into a complete system a number of advances need to be made.

One key to the application of these techniques in real environments is to including sensing in order to free the system from the constraint of working in a wholly designed environment. One promising method here is to combine our model of robot workspace with a NURBS-based surface reconstruction method such as those described in [31, 32], which take visual or range data and interpret it to produce a model of the world in terms of NURBS-surfaces. Another exploration in this direction would be to consider ways of representing uncertainty about the environment in a geometrically intuitive way.

The next stage in this would be to combine this model and the sensing work into a unified interactive graphical environment for robot task design.

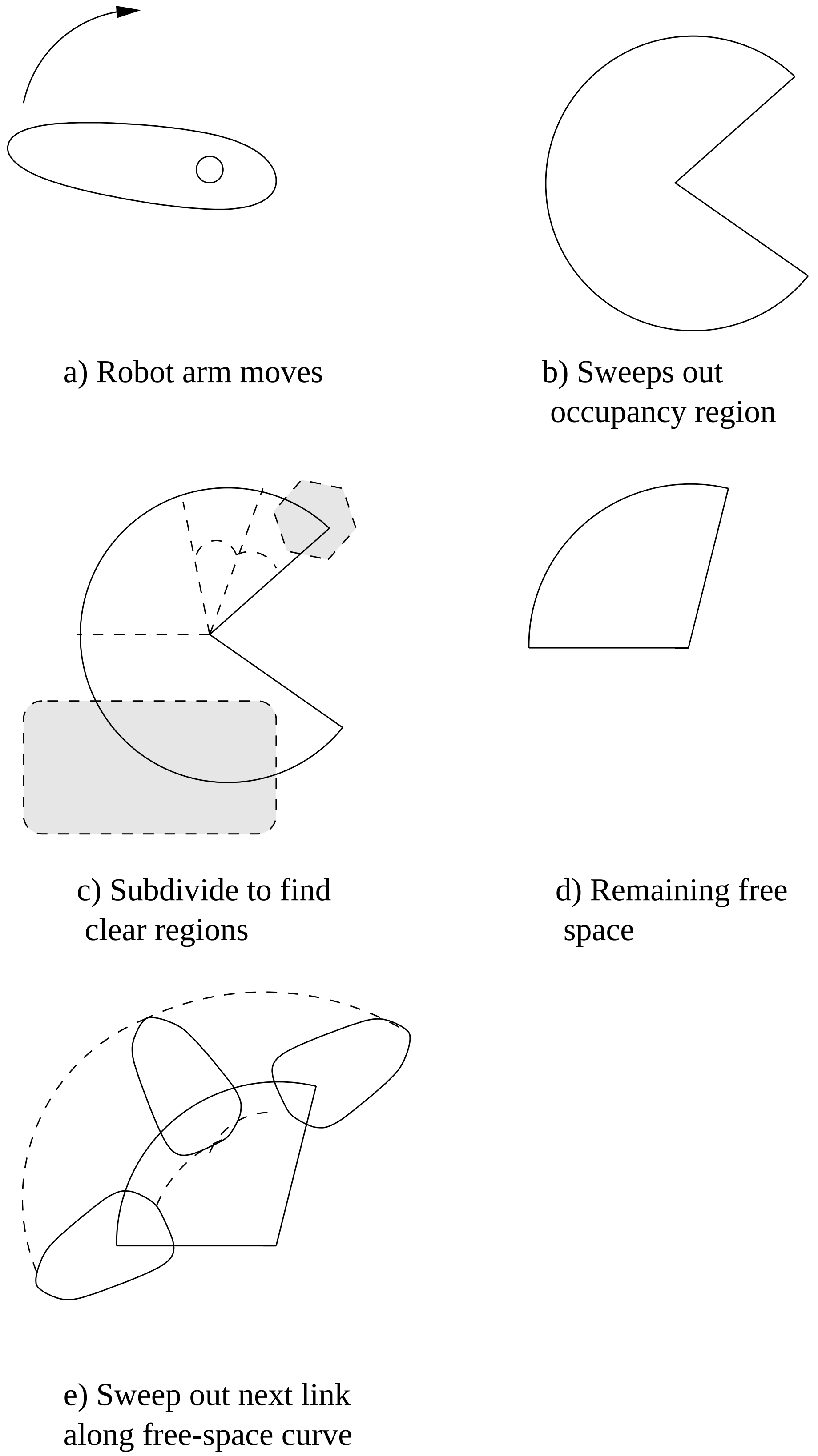

Figure 6: Trimming away to find free-space.

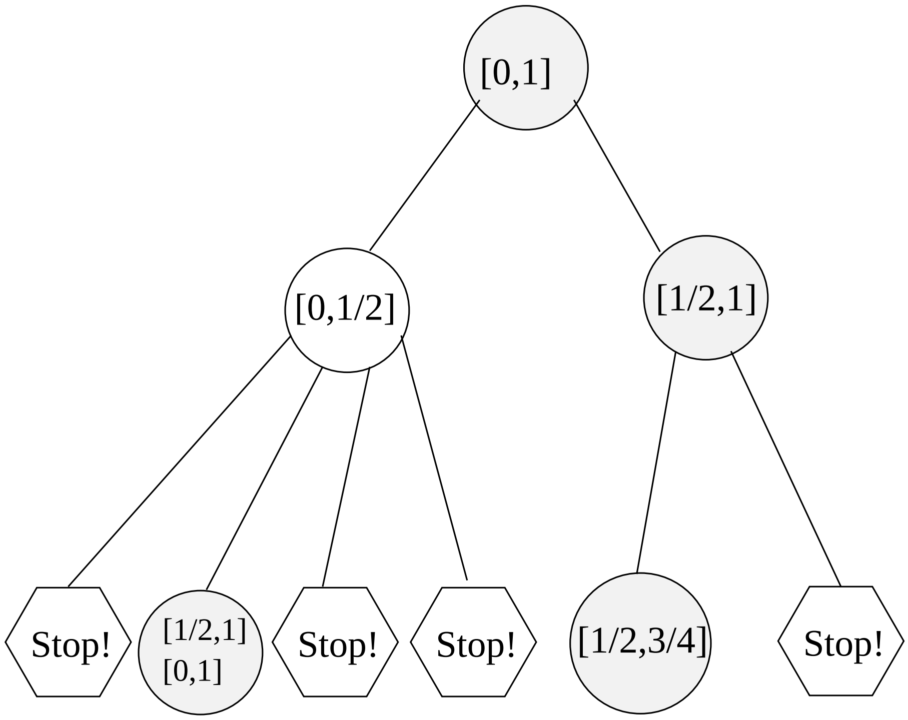

Figure 7: Tree structure after the first few stages of the free-space algorithm.

The aim here is to combine the traditional CAD-designed environment for representing real objects in the working environment with *virtual* objects representing the way in which the robot should respond to changes in the environment. An example could be the graphical representation of attractive and repulsive forces within the environment using pictures of springs (using the algorithms in [33] to compute the resultant motions). Virtual walls (appearing as translucent walls in the system) could represent no-go areas for the robot. This works towards an ideal of robot programming by specifying the minimal constraints such that the task occurs, then introducing new constraints on-line using sensors, rather than fixing a way of going a task in advance then changing it when new constraints are introduced.

There are also some simple algorithmic extensions which could be made. For example we can extend the algorithms above to carry out collision detection and path-planning in a known dynamic environment, for example to plan the coordinated motion of multiple robots. This would require the use of extrusion operators, as described in [10] to create four-dimensional space-time workspaces akin to those described in [20, 17].

Other avenues may be worthy of exploration. Some of these are computational, for example the use of parallel processing to enable these algorithms to work quickly in complex, dynamic environments. The algorithms above are amenable to parallelization at various levels, ranging from the parallel implementation of the individual small-scale algorithms such as subdivision or intersection checking, up to breaking down the problem itself by solving for different links or obstacles in parallel. Similarly it may be possible to use dedicated hardware to carry out the small-scale algorithms, as has been done for some computer graphics applications. Another direction would be to extend this modelling technique to encompass other mechanical problems, such as the design and analysis of closed-chain mechanical systems or carrying out more complex workspace analyses such as those described in [34].

Throughout all of this work the key concept is to remain within a small, closed set of representations which allow fundamental algorithms to be developed to a high level of efficiency and then applied to a wide range of problems. This has been emphasized by Farouki and Hinds in their landmark paper [35] on geometric design:

"Since the unified approach (to geometric modeling) guarantees the functional equivalence of all geometric entities of a given type, geometric operations can be performed with equal facility on simple primitives and complex sculptured geometries. Furthermore, this versatility is realized with consid-

erable conciseness of coding : A small family of geometric-function routines accepting generic geometry inputs and yielding generic geometry outputs, forms the core of the modeler."

In this paper we have described a foundation for systems which provide the context to apply the power and intuition of geometric algorithms within a new problem domain, leading towards the ideal of a single geometric design concept for both motion and shape.

References

[1] John Craig. *Introduction to Robotics*. Addison-Wesley, second edition, 1989.

[2] Nils J. Nilsson. A mobile automaton : An application of artificial intelligence techniques. In *First International Conference on Artificial Intelligence*, pages 509–520, Washington D.C., 1969.

[3] J.T. Schwartz and M. Sharir. A survey of motion planning and related geometric algorithms. *Artificial Intelligence*, 37:157–169, 1988.

[4] M. Sharir. Algorithmic motion planning in robotics. *IEEE Computer*, 22:9–20, March 1989.

[5] Y.K Hwang and N. Ahuja. Gross motion planning—a survey. *ACM Computing Surveys*, 24(3):219–291, 1992.

[6] Stephen Cameron. Obstacle avoidance and path planning. *Industrial Robot*, 21(5):9–14, 1994.

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

[8] Colin G. Johnson and Duncan Marsh. A robot programming environment based on free-form CAD modelling. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 194–199, Leuven, Belgium, 1998. IEEE Press.

[9] Les Piegl. On NURBS : A survey. *IEEE Computer Graphics and Applications*, pages 55–71, January 1991.

[10] Les Piegl and Wayne Tiller. *The $\mathcal{NURBS}$ Book*. Springer, 1995.

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

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

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

[14] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1994.

[15] J. Denavit and R.S. Hartenberg. A kinematics notation for lower-pair mechanisms based on matrices. *Journal of Applied Mechanics (Transactions of the ASME)*, pages 215–221, June 1955.

[16] Michael G Wagner. Planar rational B-spline motions. *Computer-Aided Design*, 27(2):129–137, February 1995.

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

[18] M. Erdmann and T. Lozano-Pérez. On multiple moving objects. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1419–1424, 1986.

[19] Stephen Cameron. A study of the clash-detection problem in robotics. In *IEEE International Conference on Robotics and Automation*, pages 488–493. IEEE Press, March 1985.

[20] Stephen Cameron. Collision detection by four-dimensional intersection testing. *IEEE Transactions on Robotics and Automation*, 6(3):291–302, June 1990.

[21] Z.-K. Ling and Z.-J. Hu. Use of swept volumes in the design of interference free spatial mechanisms. *Mechanism and Machine Theory*, 32(4):459–476, 1997.

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

[23] 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)*, pages 183–192. Kluwer, 1995.

[24] Q.S. Peng. An algorithm for finding the intersection lines between two B-spline surfaces. *Computer Aided Design*, 16(4):191–196 *and* C1, July 1984.

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

[26] Jonathan Yen, Susan Sprach, Mark Smith, and Ronald Pulleyblank. Parallel boxing in B-spline intersection. *IEEE Computer Graphics and Applications*, pages 72–79, January 1991.

[27] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[28] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Series in Complex Adaptive Systems. Bradford Books/MIT Press, 1996.

[29] Hanan Samet. The quadtree and related hierarchical data-structures. *ACM Computing Surveys*, 16(2):187–259, 1984.

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

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

[32] 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.

[33] A. McLean and S. Cameron. The virtual springs method—path planning and collision-avoidance for redundant manipulators. *International Journal of Robotics Research*, 15(4):300–319, 1996.

[34] K.C. Gupta. On the nature of robot workspace. *International Journal of Robotics Research*, 5(2):112–121, 1986.

[35] Rida T. Farouki and John K. Hinds. A hierarchy of geometric forms. *IEEE Computer Graphics and Applications*, pages 51–78, May 1985.