# Manipulation of a Data Structure for Potential Fields on a Non-Uniform Grid on C-Space for a Robot Arm Motion Planning Problem.

Carl Van Geem
RISC-Linz
Research Institute for Symbolic Computation
Joh. Kepler University
A-4040 Linz, Austria
Internet: Carl.Van.Geem@risc.uni-linz.ac.at
URL: http://www.risc.uni-linz.ac.at/people/cvangeem

November 1996

**Abstract**

**Keywords:** Robotics, Computational Geometry, Data Structures.
Motivated by the kinematic motion planning problem for robot arms we present a solution method based on the combination of the concept of Configuration Space (C-Space), the method of potential fields, and the hierarchical manipulation of a compact representation of a $2^k$-tree (the GET-Rep). The $k$-dimensional C-Space obstacles are approximated by a non-uniform grid. The grid is organized in a $2^k$-tree. The tree is represented by two lists with binary entries: the GET-Rep. An attractive potential field is placed on the grid, represented by a list $U$. Both GET-Rep and $U$ can be constructed hierarchically. We present algorithms for building up and modifying the data structures GET-Rep and $U$. The resulting path is a good pay-off between a short path in C-Space, a safe path, and a path consisting of moves by preferred joints.

# 1 Introduction.

## 1.1 Brief outline of approach.

In this paper we consider a particular instance of the robot motion planning problem for a manipulator with many degrees of freedom (DOF). We assume that a fixed initial configuration is given, and that any configuration is to be considered a candidate for the goal configuration.

Each point in the Configuration Space (C-Space) corresponds to a configuration of the robot in its Workspace and thus the robot motion planning problem can be solved in two steps. In the first step the free space in C-Space is computed. In the second step a path is planned for a point in C-Space.

It is very time consuming to compute the C-Space of a robot with many DOFs exactly. For this reason we suggest to approximate the free space in C-Space by a non-uniform grid. Non-uniform grids already were used for motion planning, for instance in [4], [19], [11]. The robot motion planning problem we want to tackle here demands the complete space of free placements in C-Space to be modelled.

In our previous work (see [17]) we put an "attractive potential field" on the non-uniform grid on C-Space, using a wavefront expansion technique similar to the technique in [2] on a uniform grid. A path between initial and goal configuration then consists of a sequence of adjacent cells from the cell containing the initial configuration to the cell containing the goal configuration. We organized the cells of the non-uniform grid as in a $2^k$-tree (called quadtree for $k = 2$ and octree for $k = 3$), where $k$ is the dimension of C-Space. We used the compact representation for $2^k - trees$ due to K. Burger (cf. [3]) called GET-Representation (GET-Rep). For the wavefront expansion we had to find all neighbouring cells of a cell. In $k$ dimensional space two cells $C_1$ and $C_2$ in a non-uniform grid are called *neighbours* if there exist a facet $F_1$ of $C_1$ and a facet $F_2$ of $C_2$ whereby $F_1 \subset F_2$ or $F_2 \subset F_1$. We also say that $C_1$ and $C_2$ are adjacent cells. We have observed that the algorithms given in [7] can be generalized to any $2^k$-tree and can be translated into simple algorithms for manipulating the GET-Rep. We described those algorithms and reported on some experimental implementations in [17].

The GET-Rep is well-suited for a hierarchical construction of the grid. The basic ideas behind the way in which a GET-Rep can be extended in each hierarchical step, and behind the hierarchical extension of a potential field on the grid using $k$ different increments were already presented in [16]. Here these algorithms are described in detail and integrated in their natural context. We consider the model for a manipulator as described in [6] as our starting point and motivation. By the creation of a potential field along with the hierarchical construction of the grid, it is possible to build up only part of the non-uniform grid and potential and to plan a path moving only the first few joints closest to the base support. This corresponds to the in the literature well-known concept of gross motion planning (for instance, see [9]). Due to the fact that also the potential field can be constructed hierarchically, locally the grid can be extended to all dimensions of the C-Space for finer planning.

Another property of the hierarchically constructed potential is that it allows us up to some extend to assign some preference to the joints. To each cell of the grid we assign a value of the potential. This value is the value of the potential in a neighbouring cell incremented by some constant. For each dimension of the C-Space we chose a different increment. The larger the increment, the more the planner will prefer to plan the path along the corresponding direction. Therefore moving the joints corresponding to the directions with large increment will have preference over the other joints.

Furthermore we describe how the GET-Rep and the potential field can be modified when slight changes on positions and presence of C-Space obstacles occur. These modifications should allow on-line updating of the model during execution time.

New results described in this paper are the manipulation of the data structures and the new combination of known techniques.

An important part of the construction of the non-uniform grid on C-Space which we will not consider completely in this paper is the decision problem whether or not a cell is entirely in the free space. Up to some extend this problem was considered, for instance in [19]. We are studying the matter, see [15], [18]. We assume that this decision problem can be solved by the use of a collision checker. Recent papers suggest also techniques to compute C-Space obstacles using parallel computation ([8]), or other ways to avoid exact computation of the C-Space obstacles ([10]).

## 1.2 Outline of Paper.

In Section 2 we briefly define the GET-Rep for $2^k$-trees, we show why this representation can be constructed hierarchically and we give the algorithms needed for such a construction. In Section 3 we present different potential functions on the grid. We describe the way of hierarchically constructing a potential field along with the GET-Rep. We mention briefly how to find a path and discuss some properties of the resulting path. In Section 4 we address ourselves to the modification of the data structures when, after they were constructed, slight changes are needed. We present some algorithms to do so. Finally in Section 5 we sum up some considerations on the theoretical complexity of our planning scheme.

## 1.3 Hierarchical Set-up.

We presume a hierarchical construction for the non-uniform grid and for potential field $U$ on C-Space, inspired by the modelling technique for a robot arm in [6], [5]. In the $i$-th hierarchical step, the $i+1$-th dimension is added to the non-uniform grid on C-Space.

In [5] a hierarchical solid modelling technique is used to model the robot arm. A robot arm can be seen as a chain of links, each modelled by solid bodies, each of them connected to the next link by a joint. At one of the ends of this chain, the robot arm is fixed to the environment. We start counting the links and the joints from this fixed link on. That is, a robot arm consists of links $L_0$, $L_1$,..., $L_k$, where $L_{j-1}$ and $L_j$ are connected to each other by the $j$-th joint (for $j = 1, ..., k$) and $L_0$ is fixed to the environment (for instance, $L_0$ stands on the floor of the work cell).

On the $j$-th hierarchical level, the links $L_0$,...,$L_{j-1}$ are modelled by $j$ solid bodies. The other links are modelled by their swept volume with respect to the joints following link $L_{j-1}$. The cells on the free space in C-Space are computed stepwise. In the $j$-th step, the $j$-th dimension of the C-Space is added. The $j-1$-dimensional "white cells" computed in the previous step are expanded to $j$-dimensional "white cells"; additional free space is computed for the "other" $j-1$-dimensional cells. Indeed, the swept volume considered in the previous step has led to area defined as forbidden for all values of the $j$-th parameter, while usually for some values of the $j$-th parameter there is no collision with the obstacle. So, in each step we have to compute, in those areas considered as forbidden in the previous step, the intervals for the $j$-th parameter for which no collision occurs.

In the rest of this paper we assume that in the $i$-th step of the hierarchical construction, we can decide for the $i$-th joint of the robot arm, for which values the first $i$ links will (black cells) or will not (white cells) collide with an obstacle, and for which values the swept volume of the remaining links collides (grey cells).

As we will describe in detail in Section 3, the hierarchical construction of the $i+1$-th dimension of the grid when the first $i$ dimensions are already constructed can be done in two steps. In the first step, the $i$-dimensional grid is "doubled": two copies of the data structure representation for the $i$-dimensional grid are placed behind each other in the new, $i+1$ -th, direction. This is the data structure manipulation for expanding $i$-dimensional cells to $i+1$-dimensional cells. In the second step, with information from the workspace, some cells generated from a grey cell are split into smaller cells and some of them turn white, other ones may stay grey or turn black.

The decision process for the latter can be done with techniques used in collision checking (we already referred to [19]). However, since we construct a model for a large part of, or even for the whole highly-dimensional C-Space, we need to make extensive use of a fast and accurate collision checker. The Roider method collision check for superellipsoids (see [14], [13]) seems to be one method that might meet the requirements. We already initiated some research on the problem how to enclose a swept volume of a link by a superellipsoid (see [18]) and will investigate this direction further in the near future. Moreover, superellipses and superellipsoids are useful and extremely versatile primitives for constructive solid models. They are studied thoroughly as can be found in the literature (see [1], [20]).

When a path is planned after the $j$-th hierarchical step in the construction of the grid ($j < k$) grey cells are considered to be unsafe and forbidden area.

# 2 GET-Rep: Data Structure.

## 2.1 Description.

In modelling, the $2^k$-tree is a well known technique for representing objects in $k$-dimensional space (called quadtree for $k = 2$ and octree for $k = 3$). Its properties are well studied and discussed in the literature (for instance in [12]). There are several ways to represent the structure of such a tree by lists of numbers. The GET-Rep is yet another representation of that kind.

A quadtree is a tree representation of a partition of an area into smaller subareas, possibly of different sizes. Assume a rectangle divided into 4 smaller rectangles as on the left in Figure 1, where the partitioning is done by cutting a rectangle into 4 smaller rectangles of equal size. We represent this partitioning by a tree, where each node corresponds to a particular rectangle. The root corresponds to the original rectangle. Each child of a node $\mathcal{N}$ corresponds to one of the 4 smaller rectangles in which the rectangle corresponding to $\mathcal{N}$ was divided. The quadtree corresponding to the partitioning on the left in Figure 1, is depicted on the right. An octree is the straightforward generalization of the quadtree to 3 dimensions. Here one starts with a cube and partitions it into 8 smaller cubes. In this case, every parent node will have 8 children. A $2^k$-tree is the straightforward generalization of the quadtree to $k$ dimensions. In the following we will refer to a $k$-dimensional cube corresponding to a parent node in the $2^k$-tree as a *cube*. We will refer to a $k$-dimensional cube corresponding to a leaf in the $2^k$-tree as a *cell*.

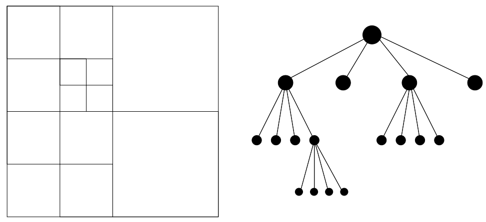

Figure 1: A 2 Dimensional Partition and Its Quadtree.

For the GET-Rep of a quadtree we assume that the children of a node are always listed from left to right according to the relative position of the corresponding partitioning. We can denote the cubes as in Figure 2, where NW, NE, SW, and SE stands for north-west, north-east, south-west, and south-east respectively, and we assume that the first child of a node in the tree always corresponds to the NW-cube, the second child to the NE-cube the third child to the SW-cube, and the forth child to the SE-cube. The tree in Figure 1 is already in this format.

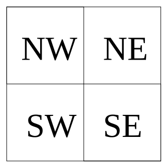


Figure 2: Relative Positions of Cubes.

The GET-Rep (cf. [3]) consists of two lists with binary entries. The first list, called *DFA*[1], is constructed by running through the tree in a breadth-first transversal order. When passing through a node, a "0" is added to the list when the node is a leaf, and a "1" when the node has children. In the case of the tree in Figure 1, we get the following DFA:

1 1010 0001 0000 0000.

Note that the whole partitioning is now represented by this one DFA-list, due to the fixed order on the children of parent nodes. The second list, called *DFB*[2], is not of any importance when we are only interested in the partitioning of an area. It describes for each leaf, that is for each

[1]Stands for "Datafield A" in [3].
[2]Stands for "Datafield B" in [3].

zero in the DFA, whether the cell is coloured black or white. For our purposes this means that the cell "contains part of a C-Space obstacle" or is "part of free space". The $m$-th entry in DFB corresponds to the $m$-th zero in the DFA, and black or white can be represented also with one digit (1 or 0).

More generally, for a grid in $k$ dimensions, we assume that the children of a node in the $2^k$-tree are always listed from left to right according to the relative position of the corresponding partitioning. Let us denote the directions in the $i$-th dimension as $A_i$ and $B_i$. That is, we have $W = A_1$, $E = B_1$, $N = A_2$, $S = B_2$, and so on in higher dimensions. A child of a node then can be represented by a $k$-tuple of directions - as in the two dimensional case for example we can write the label "NE" for the child in the north-east corner as the pair $(E, N)$. Let $X = (X_1, ..., X_k)$ and $Y = (Y_1, ..., Y_k)$ be two such $k$-tuples, where for all $i = 1, ..., k$ $X_i, Y_i \in \{A_i, B_i\}$. Let $j$ be the largest index for which $X_j \neq Y_j$, then we define $X < Y$ if and only if $X_j = A_j$. (Note that $X_j$ or $Y_j$ must be $A_j$ and the other $B_j$.) For $k = 3$ we get the following 8 tuples ordered from the smallest to the largest: $(W, N, A_3) < (E, N, A_3) < (W, S, A_3) < (E, S, A_3) < (W, N, B_3) < (E, N, B_3) < (W, S, B_3) < (E, S, B_3)$. We chose the children of a node in the $2^k$-tree to be ordered from left to right as we have ordered their corresponding $k$-tuples from the smallest to the largest. With the position of the children fixed now, we traverse the $2^k$-tree in breadth-first transversal order. When passing through a node, a "0" is added to the DFA-list when the node is a leaf, and a "1" when the node has children. Fixing the order of the children of a node allows us to have a one-to-one correspondence between the entries of the DFA-list and the nodes of the $2^k$-tree. In Figure 3 an example for $k = 3$ is given.

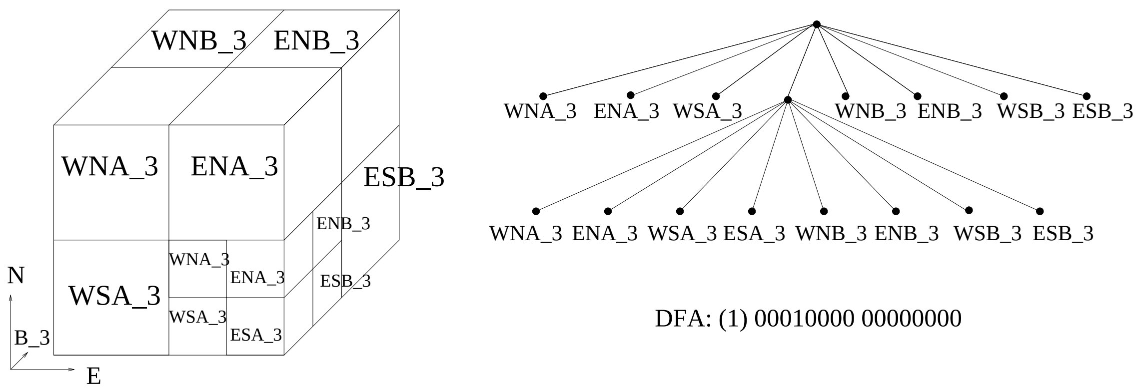


Figure 3: Example in 3 Dimensions.

Note that the first entry of the DFA-list corresponds to the root of the tree. Therefore that entry is always "1", except for the trivial case of the tree for which the root also is a leaf. In the following we will assume that the first entry of the DFA-list is a "1" and we will not longer write that entry. In the next section we will give another less technical reason for dropping that entry.

Again, the $m$-th entry in DFB corresponds to the $m$-th zero in the DFA, and black or white is represented with one digit (1 or 0).

## 2.2 More Structure in the GET-Rep.

### 2.2.1 $2^k$-tuples.

Ignore the first entry in the DFA-list corresponding to the root. Now, a way of looking at the DFA-list corresponding to a $2^k$-tree is by taking parts of length $2^k$ out of the DFA-list. The spaces in the DFA-lists of the examples above insinuate this view already.

Indeed every "1" in the DFA-list corresponds to a parent node in the $2^k$-tree and gives rise to $2^k$ children. These are represented by $2^k$ consecutive entries further in the DFA-list (each of which can be a "1" or a "0"). The DFA-list is defined constructively by a breath-first traversal through the $2^k$-tree. Therefore all the nodes on the same height in the tree (and corresponding to $k$ dimensional cubes of the same size) are represented by consecutive entries in the DFA-list.

This fact was the key observation which allowed the translation and generalization to higher dimensions of the neighbour computation algorithms of [7], as described in [17]. In the algorithm for merging two GET-Reps the $2^k$ entries in these parts of length $2^k$ are treated together. From here on we will call these parts of $2^k$ entries $2^k$*-tuples*.

### 2.2.2 Levels.

During the breath-first traversal of the $2^k$-tree the nodes are visited level by level. Therefore the entries in the DFA-list corresponding to nodes on the same level are consecutive entries. We assume for some of the algorithms described in the following that a hashing table is available which takes the level $l$ as an input and provides a pointer to the first entry in the DFA corresponding to a node on level $l$ in the $2^k$-tree. Note that if there are $z$ "1" entries on level $l$, there will be $2^k * z$ entries on level $l + 1$. Therefore we can construct such a hashing table from the DFA-list in linear time, just by one sequential pass through the list. Note that then also the height of the $2^k$-tree is known and can be used.

The entries in the DFB-list corresponding to leaves on the same level in the $2^k$-tree and thus to cells of the same size in the non-uniform grid also appear as consecutive entries. We also assume that a hashing table is available for the DFB-list which takes the level $l$ as an input and provides a pointer to the first entry in the DFB corresponding to a leaf on level $l$ in the $2^k$-tree. By counting the "0" entries in the DFA-list we can construct such a hashing table on the DFB-list from the DFA-list in linear time.

These hashing tables can be constructed along with the GET-Rep while traversing through the $2^k$-tree.

## 2.3 Merging two GET-Reps.

Assume that for the same manipulator we have a GET-Rep of the C-Space with a first C-Space obstacle $O_1$, and a GET-Rep of the C-Space with a second C-Space obstacle $O_2$. We will give an algorithm which "merges" these two GET-Reps to a new GET-Rep of the C-Space with both C-Space obstacles $O_1$ and $O_2$. We propose to compute a GET-Rep for each obstacle in parallel and to merge the resulting GET-Reps also in parallel with a divide-conquer-combine strategy. In the following subsections we will concentrate on the computation of a GET-Rep for just one obstacle. For the same reason we could restrict ourselves to convex obstacles, on the condition of being able to subdivide non-convex objects in convex components.

We describe three simple algorithms. The first one computes the position of the entry in the DFB-list corresponding to a given entry in the DFA-list. We also describe a slight variation of that algorithm which computes the position of the entry in the DFB-list corresponding to a given entry in the DFA-list on a certain level in the tree. The second algorithm computes for a DFA entry corresponding to a parent node $\mathcal{N}$ in the $2^k$-tree the first DFA entry corresponding to a child of node $\mathcal{N}$. The third one computes the "merge" of two GET-Reps.

### 2.3.1 DFB entry of a DFA entry.

The following algorithm computes the position of the entry in the DFB-list corresponding to the $c$-th entry in the DFA-list. (We assume that the $c$-th entry in the DFA-list is a 0.)

```
//count ones:
ones = 0;
for (i=0;i<c;i++)
        if (DFA[i]==1) then ones = ones+1;
//return DFB-entry:
return(DFB[c-ones])
```

The following slight variation of that algorithm computes the position of the entry in the DFB-list corresponding to the $c$-th entry on level $l$ in the $2^k$-tree in the DFA-list.

```
//where does level l start in DFA:
start_DFA = DFA_hash_table(l);
//count ones:
ones = 0;
for (i=start_DFA;i<c;i++)
        if (DFA[i]==1) then ones = ones+1;
//where does level l start in DFB:
start_DFB = DFB_hash_table(l);
//return DFB-entry:
return(DFB[start_DFB + (c - start_DFA) - ones])
```

### 2.3.2 DFA entry of first child.

Let $E_{DFA}$ be the $c$-th DFA entry, corresponding to a parent node $\mathcal{N}$ in the $2^k$-tree. The following algorithm computes the first DFA entry corresponding to a child of node $\mathcal{N}$. Entry $E_{DFA}$ is an entry on level $l$.

```
//where does level l start in DFA:
start_parent_level = DFA_hash_table(l);
// count ones:
ones=0;
```

```
for(i=start_parent_level;i<c;i++)
        if (DFA[i]==1) then ones = ones+1;
// go down:
start_child_level = DFA_hash_table(l+1);
tuple_length = 2^k;
return(DFA[start_child_level + tuple_length * ones]);
```

### 2.3.3 Merge Algorithm.

We start off with two GET-Reps referred to as $GET_1$ with $(DFA_1, DFB_1)$, and $GET_2$ with $(DFA_2, DFB_2)$. These GET-Reps correspond to two non-uniform grids. We place these grids on top of each other and we ask for the GET-Rep of the induced non-uniform grid which will be denoted by $GET_R$ with $(DFA_R, DFB_R)$.

For the computation of the merge of two GET-Reps, we use the following algorithm. The merge is done level by level, starting with the first level. At each level we repeat the same operations: we "sum" the DFA entries taking into account the DFB entries on the current level, and "add" and "remove" subtrees on the following levels.

We apply the operation "sum" from left to right on all DFA entries on the current level. The "sum" operation takes an entry $E_1$ of $DFA_1$ and adds it to entry $E_2$ of $DFA_2$ corresponding to the same relative placement of the cube in the grid. The sum is defined as in the matrix:

| $E_1$ | $E_2$ | sum($E_1$,$E_2$) |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 0 | 0 |
| 0 | 1 | $V_1$ |
| 1 | 0 | $V_2$ |

where (for $i = 1, 2$) $V_i$ is 0 when the DFB entry corresponding to $E_i$ is 1 ("black"), and $V_i$ is 1 otherwise. When we "sum" a 0 with a 1 we call one of the other two operations.

**First case:**
$E_1 = 1$, $E_2 = 0$. When $V_1 = 0$ we call the procedure "remove_subtree" with 1 as the value for its parameter. When $V_1$ is 1, we call the procedure "add_subtree" with 1 as the value for its parameter.

**Second case:**
$E_2 = 1$, $E_1 = 0$. When $V_2 = 0$ we call the procedure "remove_subtree" with 2 as the value for its parameter. When $V_2$ is 1, we call the procedure "add_subtree" with 2 as the value for its parameter.

The operation "remove_subtree($i$)" is called when an entry in $DFA_i$ corresponding to a black cell is summed with an entry in the other DFA-list corresponding to a subdivided cube $Cu$ in the other grid. The subdivision corresponding to $Cu$ is not longer needed for the "sum" operation on the next levels, so "remove_subtree($i$)" removes all entries in DFA and DFB lists which define all subcubes of $Cu$.

The operation "add_subtree($i$)" is called when an entry in $DFA_i$ corresponding to a white cell $Ce$ is summed with an entry in the other DFA list corresponding to a subdivided cube $Cu$ in the other grid. The subdivision of $Cu$ has to be preserved identically in $DFA_R$. We copy all entries corresponding to the subdivision of $Cu$ and insert them on the appropriate position in $DFA_i$ and $DFB_i$ as if cell $Ce$ would have been a subdivided cube, subdivided in exactly the same way as cube $Cu$.

We assume the existence of a hashing table on the $DFA$ and $DFB$ lists, which take a level as input and return a pointer to the first entry in the list on that level. The hashing table allows us to maintain not more than the entries at levels to be processed in future since we can use the pointers from the hashing table instead of the information in the already processed part of the $DFA$ lists. This is especially useful in adding and removing a subtree from a $DFB$-list.

All together this yields following merge algorithm, in which we denote $k$ for the dimension of the space on which the grid lies.

**main algorithm:**

```
for(level=1;level<=height;level++)
        {
            h_1 = DFA_1_hash_table(level);
            h_2 = DFA_2_hash_table(level);
            n = number_of_DFA_entries_on_level;
            for(i=0;i<n;i++)
                sum(DFA_1(h_1+i), DFA_2(h_2+i), level);
        }
```

Parameter "level" is passed on to be used in the subalgorithms add_subtree and remove_subtree.

**sum**(DFA-E_1,DFA-E_2,current_level):

```
if((value(DFA-E_1)==0) AND (value(DFA-E_2)==0)) then
        {
            value(DFA-E_R) = 0;
            if ((value(DFB-E_1)==1) OR (value(DFB-E_2)==1) ) then
                    value(DFB-E_R) = 1
                else
                    value(DFB-E_R) = 0;
        }
if((value(DFA-E_1)==1) AND (value(DFA-E_2)==1)) then value(DFA-E_R) = 1
if((value(DFA-E_1)==0) AND (value(DFA-E_2)==1)) then
        {
            DFB-E_1 = DFB entry corresponding to DFA-E_1
            if(value(DFB-E_1) == 1) then
                {
                    value(DFA-E_R) = 0 // = V_1
                    value(DFB-E_R) = 1
                    remove_subtree(2,current_level,DFA-E_2)
                }
            else
```

```
                    {
                         value(DFA-E_R) = 1 // = V_1
                         add_subtree(1,current_level,DFA-E_2)
                    }
          }
if((value(DFA-E_1)==1) AND (value(DFA-E_2)==0)) then
          {
               DFB-E_2 = DFB entry corresponding to DFA-E_2
               if(value(DFB-E_2) == 1) then
                    {
                         value(DFA-E_R) = 0 // = V_2
                         value(DFB-E_R) = 1
                         remove_subtree(1,current_level,DFA-E_1)
                    }
               else
                    {
                         value(DFA-E_R) = 1 // = V_2
                         add_subtree(2,current_level,DFA-E_1)
                    }
          }
```

**add_subtree**(i,level,DFA-E_j): // add subtree of DFA-E_j to DFA_i

```
if (i==1) then j=2;
if (i==2) then j=1;
next_level = level + 1;

h_DFA_j = DFA_j_hash_table(level);
next_h_DFA_j = DFA_j_hash_table(next_level);

n = 0;
for (c=h_DFA_j;c<DFA-E_j;c++)
          if (value(DFA_j(c)) == 1) n = n+1;
// n = number of ones appearing before DFA-E_i on level
DFA_j_copy_pointer = n*2^k positions further than next_h_DFA_j;

h_DFA_i = DFA_i_hash_table(level);
next_h_DFA_i = DFA_i_hash_table(next_level);

n = 0;
for (c=h_DFA_i;c<DFA-E_i;c++)
          if (value(DFA_i(c)) == 1) n = n+1;
// n = number of ones appearing before DFA-E_i on level
DFA_i_insert_pointer = n*2^k positions further than next_h_DFA_i;
```

```
DFA_pointer_i = DFA_i_insert_pointer;
DFA_pointer_j = DFA_j_copy_pointer;
for (c=0;c< 2^k;c++)
        {
            insert value(DFA_pointer_j) after position of DFA_pointer_i;
            if (value(DFA_pointer_j)==0) then
                {
                    DFB_pointer_i = DFB entry corresponding to DFA_pointer_i;
                    DFB_pointer_j = DFB entry corresponding to DFA_pointer_j;
                    insert value(DFB_pointer_j) on position of DFB_pointer_i;
                }
            shift DFA_pointer_i by one position;
            shift DFA_pointer_j by one position;
        }

pointer_j = DFA_j_copy_pointer;
for (c=0;c< 2^k;c++)
        {
            if (value(pointer_j) == 1) then add_subtree(i,next_level,pointer_j)
            shift pointer_j by one position
        }

remove_subtree(i,level,DFA-E_i): // remove subtree of DFA-E_i from DFA_i
if (i==1) then j==2;
if (i==2) then j==1;
next_level = level + 1;
h_DFA_i = DFA_i_hash_table(level);
next_h_DFA_i = DFA_i_hash_table(next_level);
n = 0;
for (c=h_DFA_i;c<DFA-E_i;c++)
        if (value(DFA_i(c)) == 1) n = n+1;
// n = number of ones appearing before DFA-E_i on level
DFA_removal_pointer = (n+1)*2^k positions further than next_h_DFA_i;
DFB_removal_pointer = DFB entry corresponding to DFA_removal_pointer;
for(c=2^k;c==0;c--)
        {
            if (value(DFA_i(DFA_removal_pointer)) == 1) then
                    remove_subtree(i,next_level,DFA_i(DFA_removal_pointer));
            remove the entry DFA_i(DFA_removal_pointer) from DFA_i;
            remove the entry DFB_i(DFB_removal_pointer) from DFB_i;
            DFA_removal_pointer = pointer to previous entry in DFA_i;
            DFB_removal_pointer = DFB entry corresponding to DFA_removal_pointer;
        }
```

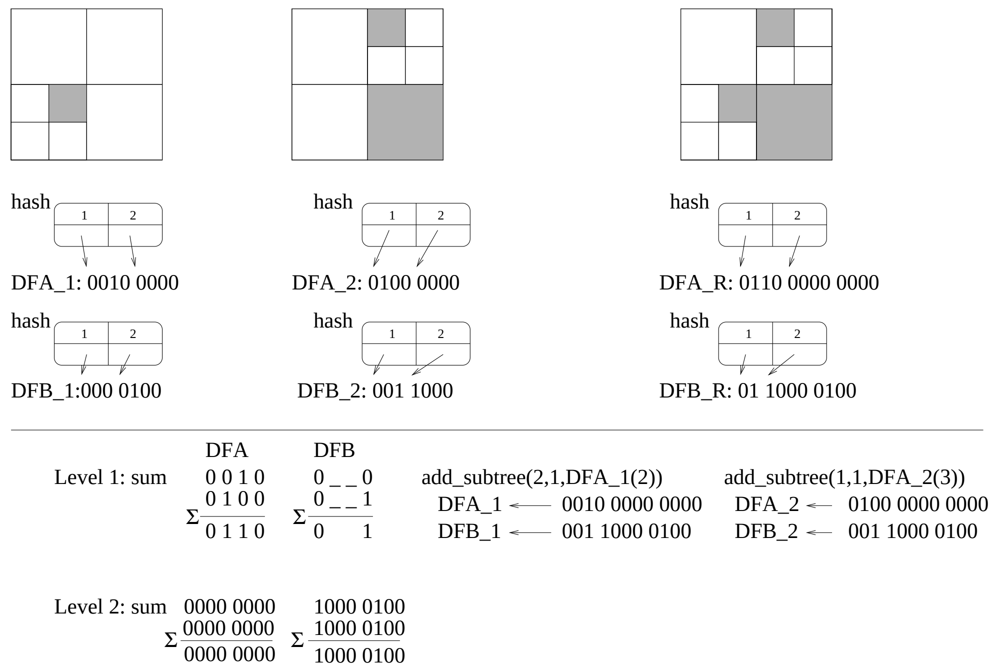


Figure 4: First Example on Merge Algorithm.

The removal of entries is done backwards on each level, and depth-first in the tree, because for determining where the subtree of a node $\mathcal{N}$ is we need the information preceeding the entry of node $\mathcal{N}$ in the DFA. We delete the subtree of node $\mathcal{N}$ before the node itself. Also we do not delete preceeding siblings of $\mathcal{N}$ before deletion of $\mathcal{N}$ itself.

As a purely technical remark, note that when a subtree is added or removed at the beginning of next_level, one must make sure that the pointer in the hashing table from next_level onwards still points to the beginning of the level.

The "sum" operation is applied to each entry of the $DFA$ lists. If $m_i$ is the number of entries in $DFA_i$, then the "sum" operation is called $\max(m_1, m_2)$ times. For each subtree that has to be added, at each level the $DFA$ entries are visited twice, the $DFB$ entries once. For each subtree that has to be removed, one additional linear passing through the $DFA$ and $DFB$ lists is necessary. From the "sum" operation an added subtree will never give rise to a call of "add_subtree" or "remove_subtree" on a next level. The removal of a subtree obviously implies that the entries removed from one of the GET-Reps will not induce any further operations (also not the "sum" operation).

We now demonstrate this algorithm on two simple examples for $k = 2$. In these examples we write $DFA_u(v)$ as a shorthand for the $v$-th entry in the DFA-list $DFA_u$ for $u = 1, 2$.

In the first example let $DFA_1$ be 0010 0000, $DFB_1$ 000 0100, $DFA_2$ 0100 0000 and $DFB_2$

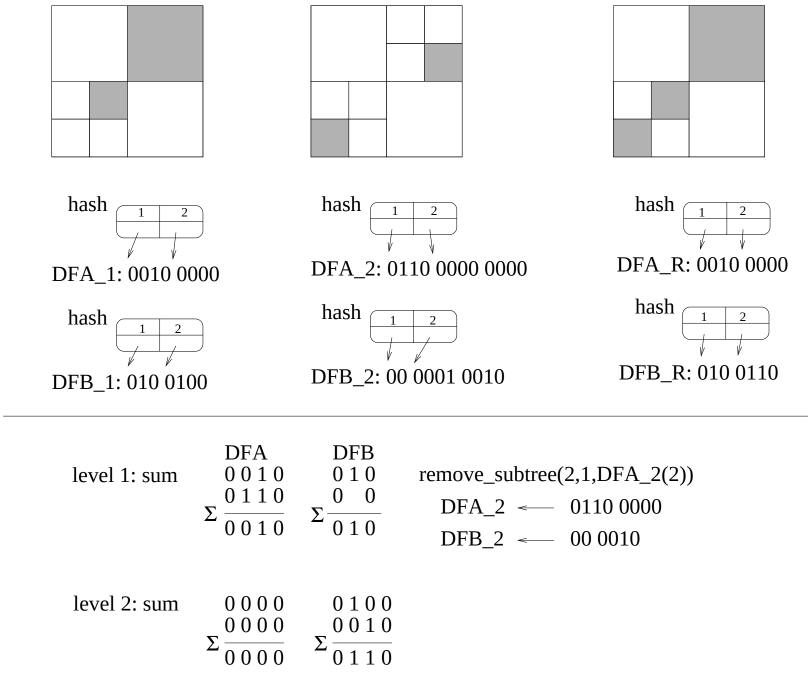


Figure 5: Second Example on Merge Algorithm.

000 1000. On level 1 we get the $DFA_R$ entries 0110 and the $DFB_R$ entries 01. The operation add_subtree(2,1,$DFA_1(2)$) is called, whereby $DFA_1$ is transformed in the list 0010 0000 0000, and $DFB_1$ into 000 1000 0100. Also the operation add_subtree(1,1,$DFA_2(3)$) is called, whereby $DFA_2$ is transformed in 0100 0000 0000, and $DFB_2$ into 000 1000 0100. On level 2 only the operation sum is called. This example is illustrated in Figure 4.

In the second example let $DFA_1$ be 0010 0000, $DFB_1$ 010 0100, $DFA_2$ 0110 0000 0000, and $DFB_2$ 0 0001 0010. On the first level the sum of 0010 and 0110 results in the first level of $DFA_R$: 0010, and in the first level of $DFB_R$: 010. The operation remove_subtree(2,1,$DFA_2(2)$) is called, resulting in the transformed $DFA_2$ list 0110 0000, and in the transformed $DFB_2$ list 00 0010. On the second level only the operation sum is called. This example is illustrated in Figure 5.

## 2.4 Hierarchical Construction.

We now present another observation on the structure of the GET-Rep, needed to allow a hierarchical construction of the GET-Rep as we will show in the following.

### 2.4.1 Useful Property of GET-Rep.

Consider a parent node $\mathcal{N}$ in a $2^k$-tree and the corresponding cube $Cu$. Node $\mathcal{N}$ has $2^k$ children corresponding to the $2^k$ subcubes of $Cu$. Half of the subcubes of $Cu$ have a relative position tuple $(X_1, ..., X_k)$ - using the notation introduced in the previous section - with $X_k = A_k$, the other $2^{k-1}$ children have a relative position tuple with $X_k = B_k$. Since the order on the relative position of these subcubes is fixed and all tuples $(X_1, ..., X_{k-1}, A_k)$ come before $(X_1, ..., X_{k-1}, B_k)$ in the order on the tuples, we can see the part of $2^k$ entries in the DFA-list corresponding to children of a particular parent node as 2 parts of $2^{k-1}$ entries: the first part of $2^{k-1}$ entries corresponds to the tuples with $X_k = A_k$, the second part of $2^{k-1}$ entries corresponds to the tuples with $X_k = B_k$. This observation allows a hierarchical construction: such a part of $2^{k-1}$ entries in the DFA is as $2^{k-1}$ consecutive entries in a DFA-list of a $2^{k-1}$-tree considering only the $k-1$ first dimensions. This observation is illustrated in Figure 6. The part of the octree at parent node $\mathcal{N}$ falls apart in two identical parts of a quadtree. One part of a quadtree corresponds to the subcubes in direction $A_3$, the other part of a quadtree corresponds to the subcubes in direction $B_3$.

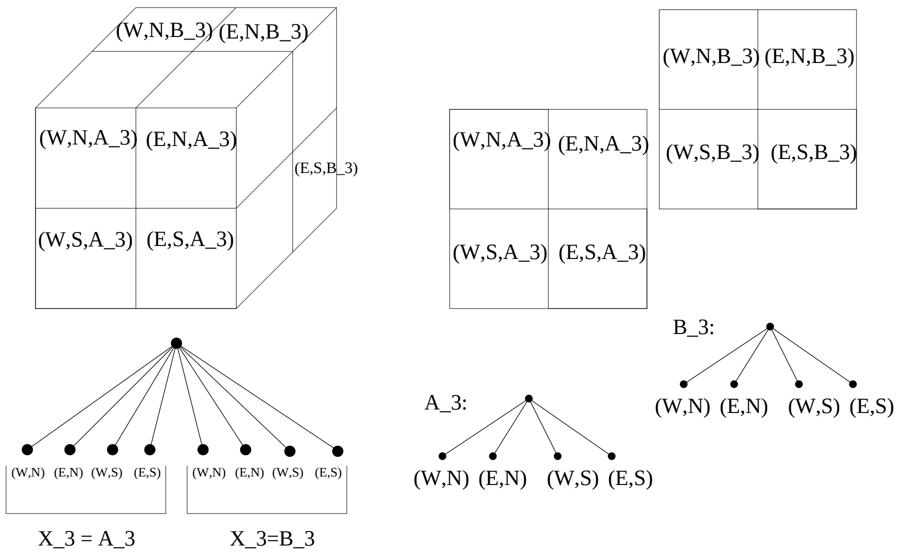


Figure 6: Illustration of the Doubling Property.

So when we want to construct the GET-Rep in a hierarchical way, using the hierarchical solid model of a robot arm ([5]) as we described in Section 1, the first step of expanding the $j-1$-dimensional cells to $j$-dimensional cells consists of "doubling" the parts of $2^{j-1}$ entries for children in positions $(X_1, ..., X_{j-1})$ in the DFA-list for the $2^{j-1}$-tree to parts of $2^j$ entries for children in positions $(X_1, ..., X_{j-1}, A_j)$ and $(X_1, ..., X_{j-1}, B_j)$. In the next subsection we present the algorithms for the hierarchical construction of the GET-Rep and clarify them on some examples.

### 2.4.2 Construction Algorithms.

In order to construct the GET-Rep in a hierarchical way, we need two routines. The first one adds an extra dimension to the C-Space, the second one processes the information from the computation of new black and white cells in the cells under investigation. (Only the cells for which no collision occurs for the first $i$ links or the cells known to be grey in the previous hierarchical step have to be investigated).

Suppose we start with the GET-Rep of an $i$-dimensional C-Space. Since we want to keep the cell construction symmetric (we want the cells to be $i+1$ dimensional cubes), we first have to "double" the GET-Rep on each level of the corresponding $2^i$-tree. Since we have to "double" all children of a "doubled" "1" entry in the DFA of the GET-Rep, we best "double" the GET-Rep level by level. In pseudo-code the algorithm that doubles the GET-Rep of a $2^i$-tree looks like this.

```
for (level = 1 ; level ≤ height_of_tree ; level++)
    {
        for each 2^i-tuple T do
            {
                insert a copy of T right behind T in the DFA-list;
                t = number of entries in T equal to "1";
                if (t≠0) then double_children(level+1,t);
            }
    }
double_children(next_level,t)
{
    denote T_list for the first t 2^i-tuples on next_level
            from Pointer onwards;
    insert a copy of T_list right behind T_list in the DFA-list;
    t_in_list = number of entries in T_list equal to "1";
    if (t_in_list≠0) then
        {
            move Pointer to first entry which is not yet copied;
            double_children(next_level+1,t_in_list);
        }
}
```

The algorithm terminates because on the last level all entries are "0". The pointer Pointer initially points to the first entry on level 2 in the DFA. For the DFB we can proceed analogously. We could add to the algorithm above a search for the tuple in the DFB corresponding to a tuple in the DFA whenever it is copied, and then copy that DFB-tuple too. Note that the length of such a DFB-tuple is equal to the number of "0"-entries in the corresponding DFA-entry.

We illustrate the first routine on two examples. Starting from a 1 dimensional C-Space with DFA

$$10,\ 00$$

we double the first level to

$$1010,\ 00\ 00$$

and then the second level to

$$1010,\ 0000\ 0000.$$

"Doubling" a level, consists of two actions. First we transform a $2^i$-tuple $T$ into a $2^{i+1}$-tuple by concatenating two copies of the original $2^i$-tuple. And second, we insert on each level a copy of the $2^i$-tuples corresponding to the subtrees of the "1" entries in tuple $T$ on the appropriate place. When this is done for all $2^i$-tuples on level $j$, we repeat the process for all $2^i$-tuples on level $j+1$. As a second example consider the GET-Rep of a 2 dimensional grid to be doubled to a 3 dimensional one. Say we start with

$$1010,\ 0000\ 1000,\ 0000.$$

Then doubling on the first level results in

$$10101010,\ 0000\ 1000\ 0000\ 1000,\ 0000\ 0000.$$

Doubling on the second level results in

$$10101010,\ 00000000\ 10001000\ 00000000\ 10001000,\ 0000\ 0000\ 0000\ 0000.$$

Finally we also double the third level, resulting in

$$10101010,\ 00000000\ 10001000\ 00000000\ 10001000,$$

$$00000000\ 00000000\ 00000000\ 00000000.$$

This process has to be carried out analogously for the DFB of the GET-Rep.

The second routine consists of splitting up some $i+1$ dimensional cells in smaller ones. Here we have to turn some "0" entries in the DFA into "1" entries, corresponding to the extreme values for the $i+1$-th parameter computed during the hierarchical construction of the $i+1$-th dimension of C-Space. In the DFB some cells will be marked white and others black.

# 3 The Attractive Potentials.

In this section we present two different functions which play the role of an attractive potential field towards the cell containing the initial configuration. The first one is similar to the Manhattan distance on a uniform grid, is constructed with a wavefront approach as in [2], and was described already in [17]. The second distance function can be constructed hierarchically, along with the GET-Rep.

A path between goal and initial configuration exists when the potential has a finite value at the cell containing the goal configuration. A path which can be constructed from this potential field on the non-uniform grid is a list of neighbouring cells following the steepest descent of the distance. We will go into more detail in the following subsections. We call the value of the distance function at cell $Ce$ the *U-value* of $Ce$. The cell containing the initial configuration will be denoted by $I$, the cell containing the goal configuration will be denoted by $G$.

Since we assign $U$-values only to cells, we can store the potential field in a single list (the $U$*-list*) with natural numbers as its entries. The list is organized in exactly the same way as the DFB list: the $j$-th entry in the $U$-list contains the $U$-value which corresponds to the $j$-th cell appearing in the DFA.

## 3.1 Non-Hierarchical Wave.

The first distance function is defined by its construction algorithm. In the first step we assign $U$-value equal to 0 to cell $I$. In the second step we assign $U$-value 1 to all neighbouring white cells of cell $I$. In the $j$-th step we assign $U$-value $j-1$ to all neighbouring white cells of cells with $U$-value $j-2$. In the case of a uniform grid the $U$-value of cell $Ce$ is the Manhattan distance to cell $I$. The algorithm for the construction of the $U$-values spreads like a wave, as in [2] on a uniform grid. An example is shown in Figure 7 of a wavefront expansion construction on a non-uniform grid in 2 dimensional space.

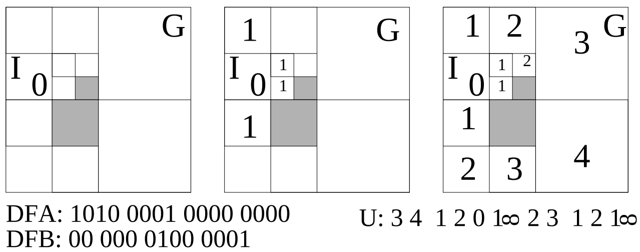


Figure 7: Non-Hierarchical Wave Expansion in 2D on Non-Uniform Grid.

Let $C_1$ and $C_2$ be some arbitrary white cells. For the non-hierarchical potential field a path $P$ from $C_1$ to $C_2$ is a list of white cells $(S_0, ..., S_q)$, where $S_0 = C_1$, $S_q = C_2$, and $S_j$ is adjacent to $S_{j+1}$ for all $j = 0, ..., q-1$. The length $l_u(P)$ of path $P$ is defined equal to $q$.

Also the $U$-value as constructed here on the non-uniform grid corresponds to a distance, namely to the distance defined by the formula:

$$D(C_1, C_2) = \min_{P \in \mathcal{P}} \{l_u(P)\},$$

where $\mathcal{P}$ is the set of all paths from $C_1$ to $C_2$. In case the set $\mathcal{P}$ is empty, we set $D(C_1, C_2) = \infty$. The function $D(.,.)$ is defined on the set of white cells.

If the $U$-value of cell $G$ is finite a path to cell $I$ exists. The shortest path with respect to the corresponding distance function between $I$ and $G$ can be constructed by the following algorithm: $S_q = G$, $S_{q-1}$ is a neighbouring cell of $G$ with minimal $U$-value amongst all neighbouring cells of $G$, $S_j$ is a neighbouring cell of $S_{j+1}$ with minimal $U$-value amongst all neighbouring cells of $S_{j+1}$. Look for these neighbouring cells with locally minimal $U$-value until $S_0 = I$ is reached.

Some experimental results were reported in [17].

## 3.2 A Hierarchically Built Potential.

Here we show that another potential field, corresponding to another distance function, can be constructed on the non-uniform grid, along with the here above presented hierarchical construction of the GET-Rep. As in the previous we consider a hierarchical model for a manipulator. In each hierarchical step $j$ of the construction of the GET-Rep, another dimension is added to the C-Space and to the non-uniform grid on it. As soon as a $j$-dimensional cell contains the initial configuration for the $j$ first parameters and the swept area of the other $k-j$ links does not collide with any obstacle, the $U$-value 0 is assigned to that $j$-dimensional cell. A wave as in the hierarchical approach assigns $U$-values to all reachable cells in the $j$-dimensional grid constructed so far. However, the increment depends on the direction in which $U$-values are inherited during the wave, as we will describe in more detail further. When a next hierarchical step $d$ $(j < d \leq k)$ is made for the construction of the GET-Rep, we first determine in which $d$-dimensional cell the initial configuration lies. Then some new waves assign $U$-values. For some cells only neighbours in a particular direction have to be found, for other cells neighbours in all directions have to be computed. In the new direction a new increment is used for the inheritance of $U$-values.

### 3.2.1 Main Algorithm.

In a grid there are three kinds of cells: white, grey and black cells. We only assign a $U$-value to the white $d$-dimensional cells $(d \leq k)$ when they are reachable from the $d$-dimensional cell $I_d$ containing the projection of the initial configuration onto the subspace just considering the first $d$ coordinates.

We can start computing a potential field as soon as the initial configuration of the robot is in a white cell. We then can give a $U$-value to each white cell that can be connected with the cell where the initial configuration is by a path through white cells. Some white cells may not be reached: they might lie in another connectivity component of the free space in C-Space or they might currently be unreachable because some grey cell is lying in the way.

When a new dimension is added to the C-Space and grey cells turned (partially) into white, grey or black (only grey cells turn into grey cells), we can determine (for the coordinate in the new dimension) in which cell the initial configuration lies.

### 3.2.2 Algorithm for Extending the Potential Field.

As in the non-hierarchical approach, the potential field is stored in a list like the DFB list. Here the entries are the integer number of the potential field assigned to the corresponding cell in the grid. During the operations for the extension of the $d$ dimensional grid to the $d+1$-th dimension, this list changes in the same way as the DFB in the grid extension presented in the previous section.

There are two kinds of waves that have to be performed for the $U$-value assignment. Firstly, during the doubling of white cells that were reachable from the initial configuration, they all get the value of the $d$-dimensional cell as a temporary value. When the $d+1$-dimensional cell containing the initial configuration is found, we perform a wave starting from that cell but only expanding in the new direction. In all other white areas generated from reachable white cells in the $d$-dimensional grid, we perform a wave in the new direction only, visiting only cells generated from the same $d$-dimensional cell, starting from the cell with same coordinate in the new dimension as the cell containing the initial configuration.

Secondly, we perform waves for the assignment of a $U$-value to all newly reachable white cells, generated from a previously unreachable white cell or from a grey cell. During whitening a cell in a grey area, we look for its white reachable neighbours in the old directions. We assign to this new white cell the smallest $U$-value of these neighbours, added to the appropriate increment. These new white cells are stored as candidate sources, ordered in sets of cells with the same $U$-value.

From the candidate sources on we perform a wavefront expansion in any direction, through all previously grey and all previously unreachable white areas. We start with a wave from those candidate sources with smallest $U$-value. Each cell that inherits a $U$-value through this wave is removed as candidate source. When the wave stops and some candidate sources are left, we repeat the process.

The increment depends on the direction of inheritance. For each direction we use another value for the increment. We use all values in the set $\{1, K, K^2, ..., K^{k-1}\}$, where $K$ is a constant which in our examples will be equal to 2. When the increment in the $d$-th direction equals to $K^{k-d}$ ($d = 1, ..., k$) the moves of joints closer to the base support will have a higher preference during the path planning. In general the order in which the increments are chosen determines which directions will be preferably taken in the path, and that results in preferences on which joints will be moved most.

After these computations, all reachable white cells in the $d+1$-dimensional grid will have a $U$-value, which finally allows us, to use the same path planning technique as in the non-hierarchical approach, following the descent of the potential field.

Note that we can start computing a potential field in the $d+1$-dimensional subspace of C-Space if the projection of the initial configuration of the robot onto the $d+1$ first dimensions is in a $d+1$-dimensional white cell. We then can give a $U$-value to each white cell that can be connected with the cell where the initial configuration is by a path through white cells. When in the $d+1$-dimensional grid the initial configuration is in a white cell while it was not yet in a white cell in the $d$-dimensional grid, then the first kind of waves is not performed and in the second kind of waves the cell containing the initial configuration is the only candidate source with potential value equal to "0".

Another advantage of a hierarchical way of constructing both the grid and the attractive potential field is that we can start looking for a path before the grid is constructed in all dimensions. As soon as the initial white $d$-dimensional cell $I_d$ is found, a potential can be computed and a path can be searched for. When such a path exists, its execution will only consist of moves of the first $d$ joints nearest to the base support. This corresponds to gross motion planning whereby moves of the joints in the wrist of a robot arm are not considered. When no path exists in the $d$-dimensional grid, we can add another dimension to the grid and to the attractive potential field and try to find a path in the $d+1$-dimensional space. For the extension of grid and potential to a higher dimensional space we do not need to recompute anything already computed in the previous steps. The only extra cost of an attempt to find a path in a lower dimensional space is determining the $d$-dimensional cell containing the projection of the goal configuration, deciding whether the cell is white or not and whether this cell has a finite $U$-value.

## 3.3 Corresponding Distance Function.

Corresponding to the potential we use in our hierarchical approach we define

$$D_w(C_1, C_2) = \min_{P \in \mathcal{P}} \{l_w(P)\},$$

where $l_w(P)$ is the weighted length of path $P$ from $C_1$ to $C_2$. The weighted length of path $P = (S_0, ..., S_q)$ can be written as

$$l_w(P) = \sum_{i=1}^{q} w_i,$$

where $w_i$ is the weight of the move from $S_{i-1}$ to $S_i$. In our examples weights will be of the form $K^j$, where the constant $K$ will be equal to 2. Notice that $w_i > 0$. In case the set $\mathcal{P}$ is empty, we set $D_w(C_1, C_2) = \infty$. Thus when the weights are of the form $K^j$ the length of a path can also be written as

$$l_w(P) = \sum_{j=0}^{k-1} m_j K^j,$$

where $m_j$ is the number of times a move in the direction to which weight $K^j$ is assigned is to be executed along path $P$. The function $D_w(.,.)$ is defined on the set of white cells. It is easy to show that $D_w(.,.)$ is a distance.

The value for cell $Ce$ of the hierarchically constructed potential field on the non-uniform grid can thus be written as $D_w(Ce, I)$, where $I$ is the cell containing the initial configuration.

The second formula for $l_w(P)$ shows that the direction and the size of cells both play a role in the length of the path and thus in the value of the potential field at cell $Ce$.

## 3.4 The Path, Planning and Properties.

When a non-uniform grid and an attractive potential field is constructed, a path is constructed by following the steepest descent of the potential field starting from the goal cell $G$ towards the initial cell $I$. Since the attractive potential field only depends on the initial cell, from any reachable goal cell we can construct a path.

The first cell of the path is cell $G$ containing the goal configuration. As long as the cell added latest to the path is not yet cell $I$ containing the initial configuration we repeat the following process: In the DFA-list we look for all neighbours of cell $Ce$ latest added to the path, find their corresponding $U$-values in the $U$-list, and add a cell with minimal $U$-value amongst the neighbours of $Ce$ to the path.

Note that unless $Ce = I$, there must be a neighbour of $Ce$ with a $U$-value strictly smaller than the $U$-value of $Ce$. This property proves that the algorithm for path construction will always terminate in finitely many steps.

Note also that during the path computation we only look for neighbours of cells in the path. The number of neighbours to be computed is much less than during the construction of the potential field. Path planning thus goes much faster than the construction of the $U$-values on the whole grid.

Due to the combination of a non-uniform grid, fine when close to C-Space obstacles and coarse in large free areas in C-Space, with the attractive potential fields the resulting path tends to go through large cells which are far away from the C-Space obstacles. For the non-hierarchically constructed attractive potential this was discussed already in [17]. For the hierarchically constructed attractive potential the property is still true in each dimension, but the different increments in different directions is an additional factor for determining which cell is going to be the next in a path. In that sense a path computed from our potential fields on a non-uniform grid is a pay-off between a safe and a short path with preferences on which joints to move more than the other joints.

# 4 Updating GET-Rep and U.

In this section we assume that already a GET-Rep and potential field on it was constructed, and that due to some slight changes in the Workspace of the robot some slight changes have to be made to the C-Space. A few operations on the GET-Rep and the potential field suffice to update these data structures when slight changes occur in C-Space.

We can classify the updates for the potential field $U$ in two classes. The first one involves only local changes to the grid and to $U$. The second one involves the impact of the change on long distance from the place where the event took place. When we only apply the first one, we still get a potential field without local minima and with the certainty that a path will be found if there exists one. However, we loose the nice property that the path will be a pay-off between safe and short path. The second kind of update of $U$ will repair this property by recomputing over a long distance the values of the potential function. We will call the "wavefront" computation for the first kind of update the *local wave* and for the second kind of update the *update wave*. We now present the four operations: delete an obstacle, insert an obstacle, model more accurately, and moving and obstacle slightly.

## 4.1 Delete an Obstacle.

**GET-Rep Update.** When an obstacle has to be deleted, some black cells turn into white ones. These new white cells form a new white area. In the GET-Rep of the grid we only have to change some entries in the DFB-list.

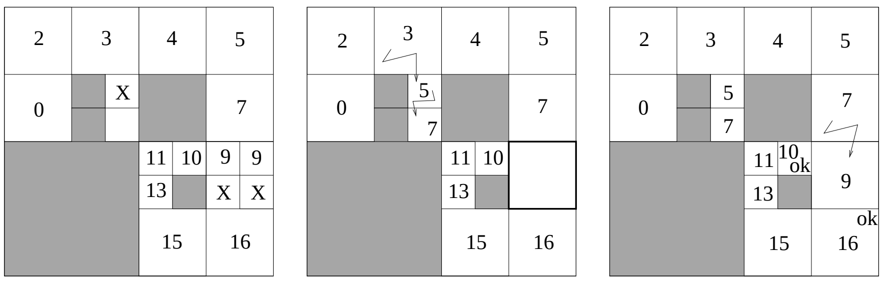


Figure 8: Local wave.

Sometimes deleting an obstacle also implies that $2^k$ cells can be merged to one larger cell, and this can happen subsequently on several levels. For each new white cell we look for its neighbours.

If all neighbours of a new white cell are white they can be merged. In that case we can apply the procedure "remove_subtree" - already used in the algorithm to merge two GET-Reps, as described in the previous section - to parent cell $C_p$ of the new white cell. We then have to check all neighbours of $C_p$. In case these are all white cells we repeat the merge process.

**Local Wave.** Locally the potential field can be "repaired", in the sense that we can assign a $U$-value to the new white cells just by looking at its direct neighbours. This is just a local wave. The local wave works in the following way. Find among all neighbours of the area of new white cells, the one with the lowest value and start up a wave in the new white cells from there. The values assigned to the new white cells have to lie between the lowest and the highest $U$-value among the neighbouring cells of the new white area. The increment has to be small enough in order to make sure that the $U$-values for the new white cells do not go over the higher bound. This increment can be determined by considering the amount of new white cells and their size.

The increment can be smaller than 1, that is a non-integer. Therefore either the $U$-list in the data structure has to allow rational numbers, or the original increments should be large enough so that the increment for the local wave can be an integer as well. At any time all original $U$-values can be multiplied with a constant factor without influencing the properties of the potential field.

If the whole new white area corresponds to exactly one new cell (after merging together several smaller cells), the $U$-value from the local wave will simply be the minimum value of all values on the merged smaller sized cells.

**Update Wave.** To perform the update wave, we start from the cell(s) in the new white area with a neighbour with lowest $U$-value of all neighbours of the new white region and proceed in the usual way. The update wave does not update those cells (outside the new white area) that have already a $U$-value equal or smaller than the value the wave would assign to them. The wave does not go on from such cells.

It is irrelevant of how many connected components the new white area consists since the update wave disperses first from all cells with the current $U$-value and only after that step from the cells with lowest $U$-value larger than the current one. In Figures 8 and 9 we show on an example such

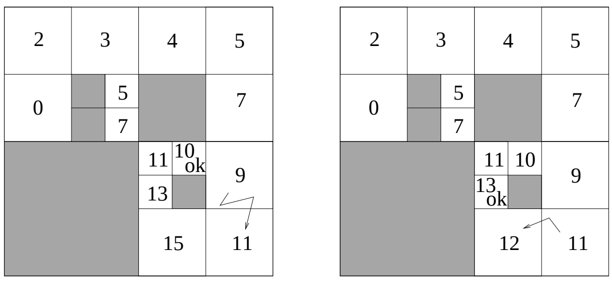


Figure 9: Update wave.

local and update waves. In Figure 8 the first adaptation of the potential field is done for three cells that came to be free (the ones labeled with "X"). First four smaller cells are merged together, then the new white cells get a suitable $U$-value. In Figure 9 the potential field is restored completely as if the cells that were freed had never been blocked.

### 4.2 Insert an Obstacle.

**GET-Rep Update.** When a new obstacle has been observed, its insertion into C-Space will result in new black cells. White cells will have to be replaced completely by black cells or subdivided in smaller cells some of which will be black and others stay white. In the first case just some entry in the DFB-list has to be changed. In the latter case, the white cell has to be split up. We can use the procedure "add_subtree" - already used in the algorithm to merge two GET-Reps, as described in the previous section - for doing so.

**Local Wave.** In some cases the new black cells will block a passage in C-Space, in other cases no such dramatic change will occur. These two possibilities have to be treated separately. When a passage is blocked, we cannot expect that a local wave is able to adapt $U$, since $U$ has to be changed drastically. This can only be done by an update wave. When no passage is blocked, a very easy and very local change will be sufficient for fixing up the potential field in order to find "a" path.

Deciding on whether or not we can use a local wave can be done simply by looking at the

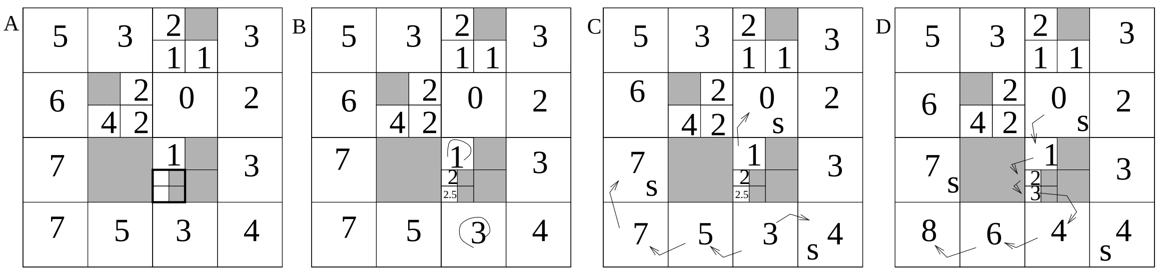


Figure 10: A: new black cells, original cell with bold face edges had $U$-value 2, B: fixing, C: wave in opposite direction, D: update wave from sources (s)

neighbours of the new black area. If they all can inherit their $U$-value from one of their neighbouring cells that were unaffected by the insertion of the obstacle, then we can use a local wave. The white cells resulting from a subdivision of a white cell that turned black partially, are considered in this test as if they had the $U$-value of the large cell from which they originate. This search is the first step of the treatment for the update wave anyway, as we will describe further, so if a local wave does not suffice the test is not a wasted overhead.

Only the new white cells which result from a previously white cell subdivided and partially turned black need to get a $U$-value. We treat each of these cells as if it were the only cell that was subdivided by the insertion of the obstacle. We look for the largest and the smallest $U$-value of all previously white (and partially or completely white still) neighbouring cells of the previously white cell under treatment. All subcells of the previously white cell under treatment will get a $U$-value between these largest and smallest $U$-values. A wave is started up from the cell(s) where this smallest $U$-value was (were) found. Only the new white subcells get a $U$-value. The increments used in this local wave depends on the size of the new subcells. Again these increments can be rational numbers, or the original increments during the construction of the potential field were large enough to allow integer increments for the local wave, or we can multiply all $U$-values by the same constant. An example is sketched in Figure 10, A and B. A cell (bold face edges in Figure 10 A) with $U$-value equal to 2 was split, two of the children turned black.

**Update Wave.** Some neighbours of the new black area inherited their $U$-value from one of the white cells that turned (partially) black and cannot inherit the same value from any of their other neighbours. Some new cells appeared and have to get an appropriate $U$-value. These cells have to be treated with the update wave as we will describe now.

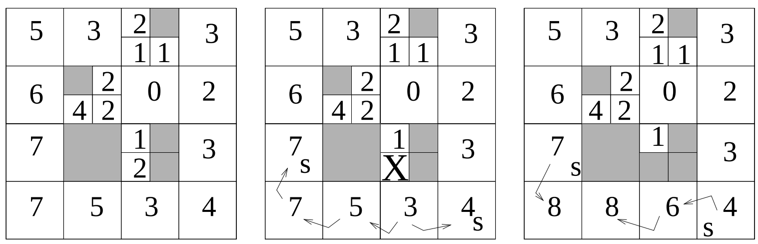


Figure 11: Passage Blocked: Wave in Opposite Direction, Update Wave. (s: candidate sources)

First we search for all the neighbours of the new black area that have to be treated with the update wave. Then we start looking for cells (potential sources for the update wave) from where these cells can inherit a new value. Sometimes such a potential source does not exist. In that case we have an area in C-Space that cannot be reached from the initial position of the robot (we have another connected component, created by the inserted obstacle). In order to decide whether such a source cell exists and, if there exists one, to find sufficiently many of them, we start up a *wave in the opposite direction* (away from the new black area). We start from the neighbouring cells of the new black area with lowest $U$-values. We call this process also a "wave" because we have to consider all the neighbours of these cells first before we move to the cells with larger $U$-values neighbouring to the new black area or neighbouring to cells previously treated by the wave in opposite direction.

We only treat cells that cannot get their $U$-value from any other cell than from one of those treated before or from a white cell now (partially) turned into a black one. For this reason, the wave has to stop, either at white cells which are potential sources or at black cells. When it stopped, we may have found potential sources for the update wave in those cells where the "wave in the opposite direction" died out. We now let the update wave start at the potential sources, adapting the $U$-values. A potential source is activated only when the inheritance of the $U$-values reaches the value of $U$ at that potential source.

Figure 11 illustrates this for the case in which a passage is blocked by some new black area. An example in case no passage is blocked can be found in Figure 10.

### 4.3 Model More Accurately.

Starting from an obstacle and a construction algorithm for C-Space, assume we get a more accurate subdivision of some black cells. That is, some black cells are subdivided into smaller cells and some of those smaller cells become white. We can assume that no complete black cell turns into a white cell, since we assume that the information we had in the previous grid was correct (i.e. in the black cells some configurations are forbidden, but maybe not all of them). The adjustment of $U$ can now be done exactly like in the approach for deleting an obstacle, described above.

### 4.4 Move an Obstacle Slightly.

Assume that a non-uniform grid on the C-Space of a robot arm in an environment with several obstacles is given. Assume that an obstacle is observed not to be exactly on the position for which the grid was computed. The slight move of this obstacle results in minor changes in C-Space: some larger white cells might be subdivided and turn partially black, some black cells might turn partially white and sometimes get merged with some neighbouring white cells to result in a larger white cell. The adjustment of $U$ can be done as before, by consecutively adding and deleting black cells with the operations described above.

## 5 Considerations on Complexity.

Essential in the wavefront expansion algorithm on a non-uniform grid in our approach is the computation of the neighbours of a cell.

There are neighbours in every direction, that is for a grid on a $k$-dimensional space we will look for neighbours in $2k$ directions. Since the grid is non-uniform the number of neighbouring cells in a particular direction depends on the size of the neighbours. If a neighbour of cell $Ce$ is a cell at the same level as $Ce$, there is only one neighbour in that direction. If a neighbour of cell $Ce$ is $l$ levels deeper in the $2^k$-tree, there will be at most $(2^{k-1})^l$ neighbours in that direction. However, neighbours of $Ce$ in the same direction can differ in size too.

For the construction of a potential field all neighbours of all cells have to be computed. When a potential field exists and a path has to be planned, only the neighbours of cells in the path will be computed. In the case of the non-hierarchically constructed potential function the number of cells in the path is equal to the $U$-value of the cell containing the goal configuration. For the hierarchically constructed potential function the $U$-value of the cell containing the goal configuration is an upper bound for the number of cells in the path unless rational values were introduced during an update procedure with a local wave (see previous section).

The complexity of the operation to determine the cell in which a configuration $\mathcal{C}(\mathcal{C}_x, \mathcal{C}_y, ...)$ lies (an operation which has to be performed for initial and goal configurations) depends on dimension $k$ of the C-Space and on the size of the cell to be found (or the height in the $2^k$-tree, with upper bound the height of the tree). Indeed, when a cube is determined in which $\mathcal{C}$ lies, with $\mathcal{C}_x$ we determine whether $\mathcal{C}$ lies in the Western half or in the Eastern half of the cube, with $\mathcal{C}_y$ we determine whether $\mathcal{C}$ lies in the Northern half or in the Southern half of the cube, and so on. This determines which child of the cube has to be considered in the $DFA$-list. If that $DFA$-entry is a "0" we have found the cell. Otherwise we call recursively the operation on the cube, now for the child we determined in the previous call.

When grid and potential are given the decision whether a cell $G$ can be reached from the cell containing the initial configuration is done by finding the entry in the $U$-list corresponding to cell $G$ and by determining whether the $U$-value is finite or not. When we know the $DFA$ entry for $G$, we can apply the algorithm of Section 2.3.1.

In the following we denote $M$ for the number of entries in the DFA-list, that is for the number of cubes in the grid. We write height_of_tree for the height of the $2^k$-tree, that is for the number of levels in the GET-Rep. We denote $Z$ for the maximum number of cubes on a level. For a uniform grid $Z$ is equal to the number of cells. Due to the uniformity and the representation in a $2^k$-tree $M = \sum_{i=0}^{w} (2^k)^i$ and $Z = (2^k)^w$ holds for a uniform grid (where $w+1$ is the height of the $2^k$-tree).

The complexity of the computation of the DFB entry from a DFA entry as presented in Section 2.3.1 is O($M$) without use of hashing tables and O($Z$) when we do use hashing tables.

The complexity of the computation of the DFA-entry of the first child as presented in Section 2.3.2 is O($Z$). The worst case appears when on the level of the parent all cubes are parent nodes in the $2^k$-tree.

Both the operations add_subtree(level) and remove_subtree(level) as presented in Section 2.3.3 contain a recursive call of depth H = height_of_tree - level + 1. In both operations the recursive call is made O($2^k$) times on each level in the tree and every call takes O($2^k.Z$) time. So both operations are of complexity O($Z$.H$^2.2^{2k}$). Here we assume that adding to and removing from the DFA or DFB of a particular entry can be performed in constant time.

In the sum operation as described in Section 2.3.3 the use of operations add_subtree(level) and remove_subtree(level) is necessary because of the non-uniformity of the grids. The real performance of the sum algorithm will depend heavily on the environment, on the Workspace. The operation add_subtree is called when a subdivision in one of the to be merged GET-Reps is added to one big cell. The operation remove_subtree is called when a large black cell is added to a subdivision in the other GET-Rep. As mentioned in Section 2.3.3 an added subtree will not give rise to a call of operations add_subtree or remove_subtree. If $m_i$ is the number of entries in $DFA_i$, then the "sum" operation is called max($m_1, m_2$) times.

The complexity of the algorithm for doubling the DFA as it is presented in Section 2.4.2 is O(height_of_tree.$M$). Doubling the DFB and the $U$-list only has complexity O($Z.M$). Here we assume that reading and inserting an entry in the GET-Rep is a constant time operation. This we can assume when we use hashing tables and pointers like Pointer in the doubling algorithm.

The use of a hashing table on the GET-Rep and the $U$-list reduces shifting around the lists. Hashing tables take space but improve dramatically the accessibility to the data in the lists.

## 6 Conclusions.

We presented algorithms for the hierarchical construction of a data structure (the GET-Rep) representing a non-uniform grid on the C-Space of a robot arm amidst obstacles. We also presented algorithms for the computation of a data structure for a potential field ($U$) on that grid. And we described methods to update both GET-Rep and $U$ without having to recompute these representations completely.

However, since we want to construct a model for at least a large part of the highly-dimensional C-Space, if not for the whole space, it is important to solve the decision problem whether or not a cell is entirely in the free space accurately and fast. The Roider method collision check for superellipsoids (see [14], [13]) seems to be one method that might meet the requirements. We already initiated some research on the problem how to enclose a swept volume of a link by a superellipsoid (see [18]) and will investigate this direction in the near future.

## References


[1] A. H. Barr. Superquadrics and angle-preserving transformations. *IEEE Computer Graphics and Applications*, 3:11–23, January 1981.

[2] J. Barraquand and J. Latombe. Robot motion planning: A distributed representation approach. *The International Journal of Robotics Research*, 10(6):628–649, December 1991.

[3] K. C. Burger. A kernel-based construction system (KBC). Technical Report diploma thesis, RISC-Linz, Joh. Kepler University Linz, Austria, Juli 1992.

[4] G. Conte, S. Longhi, and R. Zulli. CAD-based generation of collision-free paths for robotic manipulators. (preprint, to be published in Control Engineering Practice), December 1993.

[5] B. Faverjon and P. Tournassoud. Motion planning for manipulators in complex environments. In *Lecture Notes in Computer Science 391, Geometry and Robotics*, pages 87–115, 1988.

[6] Hierarchical object models for efficient anti-collision algorithms. In B. Faverjon, editor, *IEEE International Conference on Robotics and Automation, Scottsdale, AZ*, 1989.

[7] I. Gargantini. Linear octtrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, 20:365–374, 1982.

[8] J. S. González and D. I. Jones. Parallel computation of configuration space. *Robotica*, 14:205–212, 1996.

[9] Y. K. Hwang and N. Ahuya. Gross motion planning - a survey. *ACM Computing Surveys*, 24(3):219–292, September 1992.

[10] M. Kohler and M. Spreng. Fast computation of the C-Space of convex 2D algebraic objects. *The International Journal of Robotics Research*, 14(6):590–608, December 1995.

[11] J. Reif and H. Wang. Non-uniform discretization approximations for kinodynamic motion planning and its applications. In J.-P. Laumond and M. Overmars, editors, *Preprints of the 2nd Workshop on Algorithmic Foundations of Robotics (WAFR'96)*, Toulouse, France, 3-5 July 1996.

[12] H. Samet. *Applications of Spacial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[13] S. Stifter. Collision detection in the robot simulation system SMART. *International Journal of Advanced Manufacturing Technology*, 7:277–284, 1992.

[14] S. Stifter. The Roider method: A method for static and dynamic collision detection. *Issues in Robotics and Non-Linear Geometry*, 1992.

[15] C. Van Geem. Computation of a non-uniform grid on the configuration space of a robot arm amidst obstacles: Intersection problem. In *11th European Workshop on Computational Geometry*, RISC-Linz, Johannes Kepler University Linz, Austria, March 23-24 1995.

[16] C. Van Geem. Hierarchical construction of a potential field on a non-uniform grid on the configuration space of a robot arm. In *Proceedings of the 4th International Workshop on Robotics in Alpe-Adria Region (RAA'95)*, Poertschach, Austria, 6-8 July 1995.

[17] C. Van Geem. On using a manhattan distance-like function for robot motion planning on a non-uniform grid in configuration space. *International Journal of Flexible Automation and Integrated Manufacturing (IJFAIM)*, 3(2):165–177, 1995.

[18] C. Van Geem. Enclosing robot links by superellipses and superellipsoids. Technical Report 96-01, RISC-Linz, Joh. Kepler University Linz, Austria, 1996.

[19] B. J. H. Verwer. A multiresolution work space, multiresolution configuration space approach to solve the path planning problem. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 2107–2112, Cincinnati, Ohio, May 1990.

[20] M. Y. Zarrugh. Display and inertia parameters of superellipsoids as generalized constructive solid geometry primitives. In *Proceedings of the 1985 ASME International Computers in Engineering Conference and Exhibition, Boston, MA*, volume 1, pages 317–328, August 4-8 1985.

# List of Figures