Real-Time Path Planning for Robot Arms

Nigel William Balding

This thesis is submitted in fulfillment of the requirements
for the Degree of Doctor of Philosophy

University of Durham
Science Site
South Road
DURHAM
DH1 3LE

Department of Engineering

May 1987

ABSTRACT

This thesis presents two new methods for the automatic programming of robots, which were developed at the University of Durham. A model of the robot and its surroundings is held in the computer memory in a form which can be accessed quickly by the path generation algorithm. Information is fed to the computer which defines a task for the robot to perform. The path generation algorithm then calculates the coordinates for the movement of the robot, avoiding obstacles. Finally, the information is down-loaded into the robot control computer in its own language. An important feature of the method is the high speed of calculation and data transfer, which is designed for real-time operation.

The world model is represented as a collection of spheres, some overlapping, and the robot is represented by connected cylinders. This simplified representation is the key to the speed of calculation of the path. Different criteria are used for the optimal path selection, such as minimisation of the overall time taken, the distance travelled and the joint rotation.

Two path planning methods have been developed. The first incorporates a local method of trajectory calculation and the second uses a global method. Both methods are suitable for real-time applications, but they have different properties which can be exploited in different applications. The relative merits of the two methods are discussed.

These methods provide an on-line, real-time capability for collision free path calculation in a flexible manufacturing environment.

Acknowledgements

I wish to express my gratitude to the following people :-

To Dr. Clive Preece, my supervisor, for his help and advice towards my research and thesis.

To Delta Plc. for their financial assistance.

To my proof readers Sarah Balding, Winefred Horton and Anne Cowey.

LIST OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS

AGV	Automatically Guided Vehicle
CSG	Constructive Solid Geometry
FMS	Flexible Manufacturing System
G	Goal position of a robot path
Gm	A position close to G and clear of obstacles
MAP	Manufacturing Automation Protocol
NC	Numerically Controlled
S	Start position of a robot path
Sm	A position close to S and clear of obstacles
T1-T5	Axes one to five on the test robot

DECLARATION

The work presented in this thesis is the author's own work and has not been previously submitted for any other degree.

Nigel Balding

CHAPTER 1

INTRODUCTION

Robots are now being used for many differing tasks in industry world-wide. The robot's principle advantage over dedicated handling machinery is that it may be reprogrammed to carry out different tasks. This property has enabled new types of production methods to be developed, an example of which is the Flexible Manufacturing System, where small batches of parts are produced.

To change a robot's task requires the reprogramming of the robot path. Currently robot paths may be programmed in one of three ways:

(a) On-line or explicit programming. The user 'teaches' the motions required. Usually this is done by 'lead by the nose' or by using a teach pendant. The required path is tracked manually for the 'lead by the nose' method and the robot controller records the coordinates. When using a teach pendant, the robot is moved over the required path under manual control. Again coordinates are recorded at discrete intervals.

(b) Off-line programming. The robot and its surroundings are simulated on a computer. The user defines and modifies the robot program using a graphical display of the computer model. The computer stores the simulated robot program and downloads it to the robot controller, if required.

(c) Automatic programming. An automatic programming system also has a computer model of the robot and its surroundings. The user specifies a task such as 'load part B into machine X', and the computer automatically calculates a robot trajectory to carry out the task efficiently and safely. Automatic programming systems may be used off-line, for simulation and trajectory development purposes, or on- line, to adapt to changing processes or varying tasks.

Developing robot programs can be an expensive and tedious task. However this cost can be justified if the robot is used for a repetitive job when the cost of programming is spread over many operations.

It has been suggested that 'the future of robotics lies in truly flexible systems which can be reprogrammed at will and, crucially, without taking the robot away from production tasks'. (Irvine 1986). This capability can only be provided by off-line programming or automatic programming.

Automatic programming has the advantage that the programming cost for new paths is eliminated. Thus, the robot may be used in situations where the task is changing, with paths automatically reprogrammed between tasks.

1.1 Discussion of robot programming methods

Robot programming may be divided into four hierarchical levels of control:

(a) Joint level. Most robots have 4 to 6 movable joints so that they may change their position. The robot's joints are often concentrated in three positions, these are frequently called the shoulder, elbow and wrist after the joints of the human arm. When a robot is programmed at joint level, the displacement of each joint is specified by the program. (The joint positions are defined by joint coordinates).

(b) Manipulator level. Robots have grippers or end effectors at the ends of their arms for gripping objects or for holding tools. The positions and orientations of the gripper are recorded in coordinates relative to the robot base. (These are called world coordinates).

(c) Object level. The task is specified in terms of the positions and movements of objects within the robot installation eg. lift part C 10 mm upwards. At this level of control the computer holds a model of the shapes of objects

within the robot's reach, so that the necessary manipulator actions can be determined.

(d) Objective level. The task is specified in a general form such as 'spray car door' or 'weld part A to part B'. Objective level control implies a knowledge of the robot's surroundings and of the process. Thus the command 'spray car door' requires a knowledge of the robot, the car door and the process of spraying.

1.1.1 On-line programming

On-line programming uses either the joint or manipulator levels of control.

Most robots are taught their sequence of operations on-line. In the past the only way of seeing what a robot would do, given a certain set of commands, was to make it obey them. This is the reason why virtually all of today's industrial robots have a teach mode type of programming. But this may not be convenient because :-

(a) It may be dangerous - mistakes can be made which can damage the machine, its surroundings, or the programmer.

(b) It may be expensive if the robot is already in full-time use, and it has to be withdrawn from service while it is being taught.

(c) It may be impossible because the robot, or the machinery to which it will be linked, may not exist yet.

1.1.2 Off-line programming

In current implementations of off-line programming systems, either the object or manipulator levels of control are used. There is at present a fair amount of research effort being put into off-line programming and some working systems are evolving, such as the GRASP package at the Universities of Nottingham and

Loughborough. This system is now marketed by BYG systems and is described by Bonney et al. (1985). Other systems are available from McDonnell Douglas Information systems, Computervision and Robcad.

Programs such as GRASP may be used by a programmer off-line to create a path for the robot. The programmer tells the robot the directions and distances in which the robot must move. The robot and its workplace are displayed graphically so that reasonably efficient movements may be programmed. Collisions may be checked for by the programmer at any point on the robot's path.

It is worth noting that the development of off-line robot programming methods today is analagous to the early development of off-line programming for numerically controlled machine tools. The utilisation of expensive machine tools was greatly increased by off-line programming. Production engineers were able to write control programs well away from the machine tool, while the latter was doing productive work, and then transfer them to the machine, using punched paper tape. The sponsoring company of this research, Delta Computer Aided Engineering Ltd., provides off-line programming of NC machines as a service for a large number of engineering firms.

Off-line programming offers the possibility of reducing 'down time' for production lines, or robotic cells where the robots are used. When programming changes have to be made because of product changes, then the robot programs can be prepared in advance for a fast change over. This is also an advantage in that the programmer is removed from the robot's often hazardous environment and from the occasionally bad-tempered robot itself. The programmer no longer need remember how to program each different type of robot (robots have many different languages). The programmer may also be able to spread the workload of a major change in production over a long period, rather than working while the production line is stopped when the robots are free.

The off-line approach has advantages which are side effects of the new programming method. The computer simulation of a robot and its surroundings required for off-line programming is a useful tool for planning a robot's workspace or testing different robots for a particular application, before their purchase.

These advantages of off-line programming may be summarised as follows :

(a) Reduction of down time.

(b) Improved safety.

(c) The ability to simulate a variety of approaches to work cell design and choose the best.

(d) Spreading the operational load on the programmer. (Consider the need to change a complete production line when introducing a new model).

(e) The ability to link with a computer integrated manufacturing (CIM) approach.

(f) Programmers do not need to be familiar with many different robot controller languages.

(g) Longer and more complicated robot programs may be developed.

1.1.3 Automatic programming

Automatic programming is done at the object or objective level. Given the world model and the movements of objects that are required, the system automatically programs the robot. This is the main subject of this thesis. Two different methods of automatically planning the major movements of robot arms are described.

Automatic programming of robots was first developed in the United States for use with mobile robots involved in planetary exploration. Since then academic interest has continued and the problems of path finding and collision avoidance have been classified in the area of Artificial Intelligence.

Automatic programming may be used either on-line, where the paths generated are implemented immediately, or off-line, where the paths may be stored for use when required. There are two practical differences between off-line and on-line systems. Firstly on-line systems have to calculate robot paths quickly, in order to keep up with the demand for instructions from the robot. Secondly the programming system has to be robust and inexpensive so that it may be used in a factory.

Automatic programming systems have all the advantages of off-line programming listed above. Indeed one may envisage that packages such as GRASP may be enhanced by adding automatic programming algorithms.

The extra advantages of automatic programming systems may be listed as follows.

(a) A reduction in costs, as a programmer is not required to work out the robot path either at the manipulator level or the joint level.

(b) A reduction in errors (less chance of human error).

(c) More efficient robot movements may be calculated. This is because a cost function (see section 2.6.2) may be used to determine the best robot paths.

(d) By connecting the automatic programming system to sensory feedback from the robot installation, the system may be used on- line to re-program the robot, to take account of a changing environment or a changing task.

1.2 The requirement for a real-time system

This research was initiated from the requirement for a more intelligent robot in the flexible manufacturing environment.

In a flexible manufacturing system a wide range of products are produced using the same machine tools and handling facilities. These systems require an ability to cope with changes in the production tasks. In a highly flexible system such changes require dynamic re-scheduling, which involves fast re-calculation of robot paths. The extent to which this can be achieved will determine the re-scheduling capabilities of the flexible manufacturing system.

A real-time automatic programming system may be used in conjunction with sensors such as vision systems. Such systems will be able to cope with a new range of applications for robots, such as planetary and sea bed exploration, picking components presented at random and fruit picking.

1.3 Representation of the world model

An example of the test robot and its workspace is shown in figure 1.1. The robot used was a Smart Arms robot. This was an inexpensive robot designed for teaching and research purposes. Although it was the only robot available for this research and not necessarily an ideal design, it satisfied its requirements adequately.

The world model of a path planning computer contains a geometric representation of the robot's workspace and the robot itself. It also contains a kinematic model of the robot so that it knows how the robot can move and its joint limits. The robot's workspace is the total volume which the robot sweeps as it passes through all its possible configurations. The workspace of a robot may be generated by several methods :- (Lee and Yang 1983), (Yang and Lee 1983), (Cwiakala and Lee 1983), or (Hansen et al 1983). In practice obstacles which are likely to be in the workspace are modelled and the workspace itself does not need calculation.

The problem of finding paths for robots has, up until now, been tackled with large computers which have modelled the robot and its surroundings by using

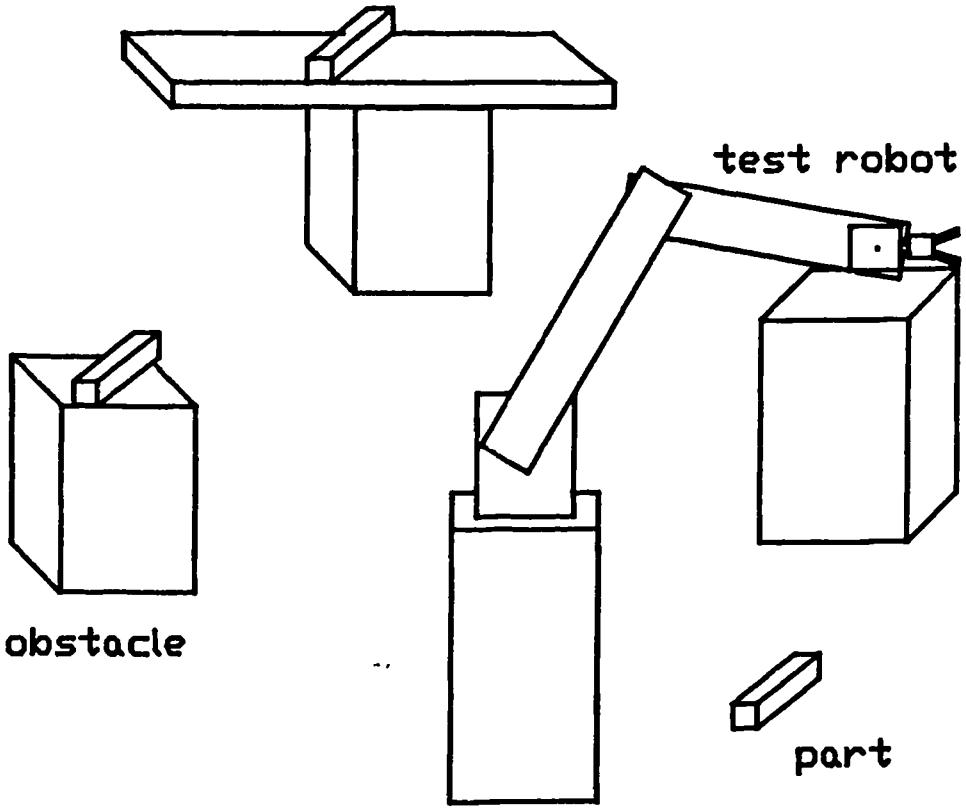


Figure 1.1 The test robot and its workspace

polyhedral representations. Examples of this approach may be found in Udupa (1977), Lozano-Perez and Wesley (1979), Lozano-Perez (1981, 1983), Brooks (1983a, 1983b), Luh and Cambell (1984) and Cameron (1982).

In order to ease the computation involved in the pathfind problem, a simple method of representing obstacles and the robot was chosen. The obstacles were modelled as sets of spheres and the robot arm was modelled by a set of cylinders and spheres.

The robot arm had three main revolute joints : two at the base which allowed the upper arm to rotate and elevate and the other, called the 'elbow joint', between the forearm and the upper arm, which allowed the angle between the forearm and the upper arm to change.

One advantage of the sphere representation was that the spheres could be enlarged to account for the thickness of the robot links. This simplified the problem further to that of finding paths for lines through sets of spheres.

1.4 Path planning

Finding a collision-free path for a manipulator through an obstacle-cluttered space is not a trivial problem. Brooks (1983a) reported that algorithms exist for solving the problem with any manipulator although their computation time makes them impractical. He described their implementation complexity as 'staggering and untried'.

Finding a path, regardless of how efficient the path is, will not be of much use as it lowers greatly the productivity of any existing robot application. Therefore a cost function for a robot path must be defined and some attempt made to reduce this to a minimum. In most of the research done so far the cost function has been defined as the distance moved by a point at the tip of the end effector. Other factors may be taken into consideration, although they may be more complicated to calculate. Some interesting work has been carried out by Gilbert and Johnson (1985), where the energy used by a two degree of freedom robot was minimised.

It is important that in an on-line application the time of calculation for a robot path should not contribute appreciably to the time of manufacture. Where possible, calculation should proceed concurrently with manufacture and programs should be ready for implementation before they are required by the robot control computer.

In order to achieve real-time operation a compromise must be made between the efficiency of the calculated path and the calculation time. For any path planning problem there is an optimum solution based on a chosen cost function.

---

1. BLEASDALE 8086 MICRO COMPUTER

Operational constraints may make a faster sub-optimal solution more acceptable in a particular application.

The method adopted here produces sub-optimal paths using a simplified world model. Any sub-optimal solution must ensure that the calculated path is a safe one, that it satisfies the 'collision free' criteria and that any divergence from the optimal path tends to produce greater, rather than smaller, clearances.

1.5 Description of work done

The computer programs developed for this research were written in Pascal and run on an Intel 8086 based micro computer. This was connected to the robot control computer by a serial link. The test robot was a Smart Arms 6R 750 robot.

In order to provide a real-time solution using a small micro-computer, it was necessary to represent the problem as simply as possible. A simple representation provided a saving in the computer memory (required to store the representation) and an increase in the speed of path planning.

The robot is represented by two connected cylinders and spheres, which contain the gripper and workpiece. The obstacles in the robot's workspace are all modelled as spheres.

Figure 1.2 shows the control software for the automatic programming system. Three separate programs are run. These are :-

(a) The world model program. This stores data about the obstacles on disk for the path planning program.

(b) The path planning program. This calculates the path trajectory for the robot control computer and down-loads the information.

(c) The robot program. The robot control computer receives the trajectory data and moves the robot.

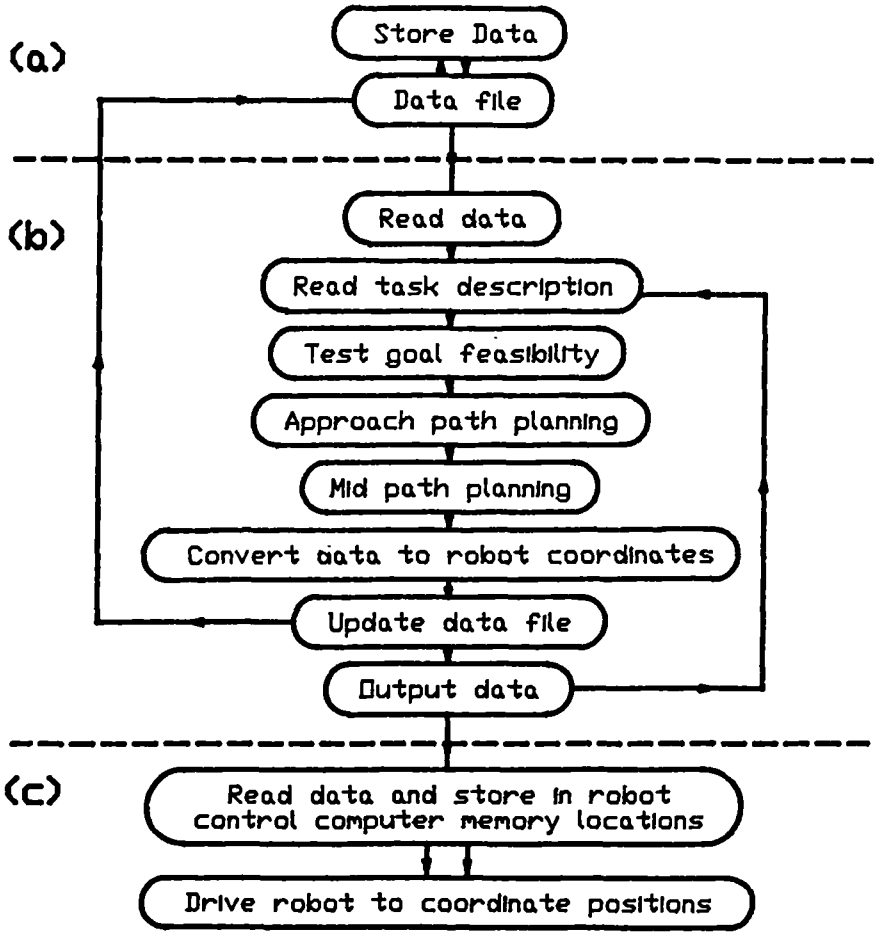


Figure 1.2 Control Software

The world model is calculated from measurements of the real robot environment. This information is then stored in a data file for the path planning program. The path planning program calculates the robot trajectory and converts it into coordinates which are acceptable for the robot control computer. When the robot is not moving, it indicates to the path planning computer that the next trajectory data is required.

The goal feasibility phase is checked to ensure that the final desired position of the robot is possible without it intersecting obstacles.

Approach paths are paths which take a robot small distances from positions clear of obstacles to positions where the robot can grip or release its payload. Because of the special nature of approach paths, which require fine movements, good local information of the part and the obstacles is necessary before they can be planned. Hence the robot approach paths used in this work were generated separately from the main robot movements.

The mid-phase planning is done automatically by the path planning computer. The mid-phase planning creates trajectories which are as efficient as possible, efficiency being defined in terms of distance moved by the robot.

The robot trajectories are converted into coordinates which can be used by the robot control computer. The information is then passed on via a serial link when the robot has finished any existing task.

1.5.1 Method 1

The first path planning method which was developed is described in chapters 6 and 7. The trajectory of the upper arm of the robot is planned first using a graph searching method. Having fixed the upper arm trajectory, the forearm motion is planned using a heuristic method. The algorithm tests for potential collisions and then avoids them by using heuristic rules which raise or lower the forearm.

1.5.2 Method 2

The second method investigated was based on a method of graph searching. The set of configurations for which the robot did not intersect obstacles is calculated. This set of configurations is structured into a graph of nodes and branches. The problem of moving from one configuration to another is transformed into that of finding a set of branches which connectsthe nodes representing the two configurations S and G.

It was found that the second method performs better generally, although a large amount of computer time is required initially to set up the graph for searching.

The path planning for both methods is divided into three parts :- goal feasibility, approach paths and mid-phase planning.

1.6 Achievements of work done

This thesis has discussed the practicalities of automatically programming robots for the first time. The need for an automatic programming system has been examined and the information and equipment required for the system itself has been described. Two methods of automatically calculating robot paths have been described and compared with other work in this field.

Automatic path planning has been achieved before for certain robots and workspaces. However, these solutions have been essentially off-line and the speeds of solutions were not matched to the speeds of robot execution. This thesis shows that the possibility of automatic programming for industrial robots outside of the research laboratory, is not far off. The system developed and operated at the University of Durham had a very low hardware cost. Hence it has been shown that the costs for future commercial automatic programming systems need not be prohibitive.

This thesis has also addressed the problem of path efficiency. The cost of a particular path may be assessed by many factors, such as time, distance travelled, energy used etc. It has been concluded that, from a path planning point of view, the distance travelled in the path planning space is the easiest cost to evaluate and that, in general, the shortest path is a good solution when the other factors, such as time and energy used, are considered. The two methods discussed in the thesis use cost functions to evaluate their solutions. They minimise their

cost functions within the limitations of their path searching methods. These cost functions could be redefined to take into account any weighting of the different types of cost which the user might desire.

The two methods of path planning described are both original. However, they may be classed with other off-line methods which have been described elsewhere.

CHAPTER 2

BACKGROUND

Automatic path planning for robot arms is an important research problem and has been tackled in various ways. These different approaches are discussed in this chapter.

2.1 Introduction

Automatic programming for robots was first investigated in the United States by Pieper (1968) and later Udupa (1977). This research was to design a robot for use in planetary exploration. Since then several other authors have added to this field of research.

The main parts of the computer programs have been the world model and the path planning algorithms. These require the following information in order to specify and solve their tasks :-

(a) A geometric and kinematic description of a robot. This is required either as an input or as an inbuilt part of the system.

(b) A geometric description of the robot's environment. This provides the information needed to produce collision free paths.

(c) The task description. This provides the objectives of the path planning problem.

The desired output is a trajectory for the robot. This is the movement of the robot which efficiently achieves the task without colliding with obstacles.

The type of world model chosen has a considerable effect on the path planning algorithms. The most popular models chosen for path planning systems have been

polyhedral models. The different types of models are discussed in sections 2.2. (The robot) and 2.3. (The obstacles).

This research requires a good understanding of path planning problems at both an abstract (mathematical) level and at a computer level. At an abstract level a good understanding is required for obtaining proofs and developing planning methods. At a computer level a good understanding is required for obtaining efficient and practical programs.

A prerequisite to many path planning methods has been to have one or more transformations of the real problem space to abstract spaces. These transformations facilitate more efficient path searching methods. Such transformations are examined in section 2.4.

Many different path planning methods have been proposed in the past. Although they are all different in certain respects but they may be grouped into broad categories. These are discussed in section 2.5.

2.2 The choice of robot model

Any robot consisting of a series of links and revolute joints may be represented by the general schematic model shown in figure 2.1. In this case there are n coordinate frames which specify the robot's configuration. Figure 2.2 shows the variables which define one joint's position in relation to the next. The line ak specifies a vector which is perpendicular to both the Zk and Zk+1 axes, lk specifies the angle between the axes and bk specifies the distance along the Zk+1 axis of the joint k+1. The values ak, bk and lk are fixed for a particular link. The variable mk specifies the joint angle relative to some origin fixed at the joint.

This theoretical model may be called the kinematic chain of the robot. This model has been widely used as the basis for modelling revolute robots (Lee and

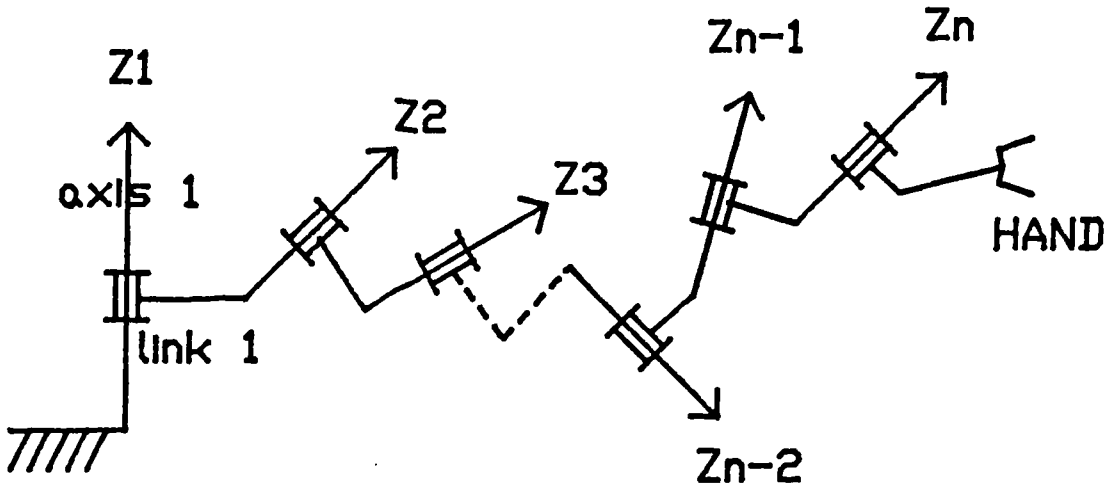


Figure 2.1 A generalised robot kinematic chain

Yang (1983), Cwiakala and Lee (1985), Hansen et al (1983)). It defines the geometric relationship between joints, onto which may be placed the flesh of the robot.

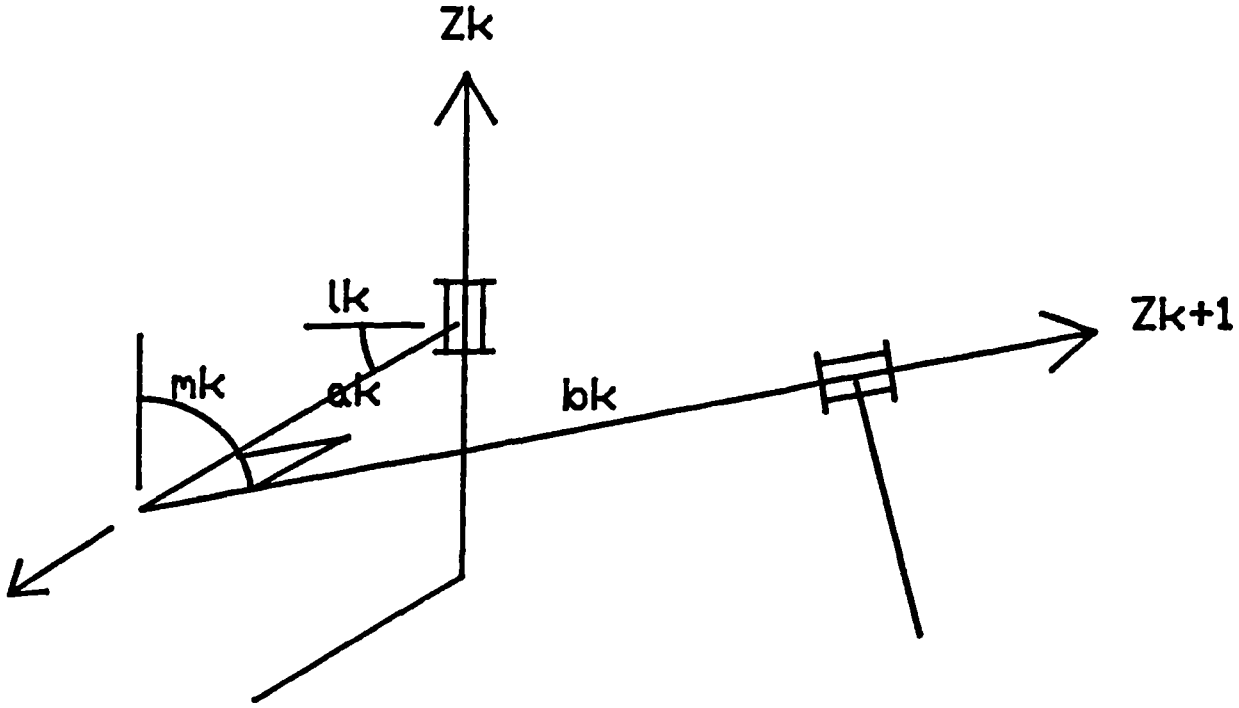


Figure 2.2 The kth joint of a robot

An interesting robot model of connected spheres was proposed by de Pennington (1983). In this work de Pennington was interested in collision avoidance

rather than in path planning. The method used a solid model of the surroundings. The robot's path was simulated and the swept volume of the robot-sphere model calculated. The robot swept volume and the obstacle volumes were compared and where intersections between volumes occurred, collisions were indicated. The reason that spheres were used was that they produced swept volumes of regularized cylinders or tori under the restricted robot trajectories considered. Thus the calculation of the swept volume became tractable.

This method was unsuitable for automatic path planning as the sweeping of spheres was restricted to translational or rotational sweeping only. This precluded the use of more than one robot motion at once and restricted the paths available for planning.

Collision detection and avoidance is simplified by using spheres, as they are the most simple three dimensional shapes. Hopcroft (1983) describes how to calculate intersections among spheres efficiently. The method of modelling awkward shapes, such as a gripper, on robot arms by spheres has been used in this research.

Udupa (1977(a)) simplified the model of the robot to two connected lines, then one line and then a point, by using obstacle transformations. Before any obstacle transformations were carried out the basic robot model was defined as two connected cylinders. The robot being simulated was the Stanford Arm.

The advantages of this representation were that path planning for a line or cylinder was much easier than the more complicated shape of the real robot.

Many robots have long slender links so the workspace lost through conservatively approximating the links by cylinders is not great. This representation allows obstacle transformations which account for the thickness of the cylinders and reduces the problem to finding a path for a line through the transformed obstacles. This is probably the most widely used representation of the robot.

Other methods used polyhedral representations of the robot eg. Brooks (1983(a)). This is a very accurate method of representing the robot although the extra computational effort for calculating collisions and planning paths is very large. Hence it is not possible to use this representation for real-time calculations at present.

2.3 Obstacle representation

Many computer representations are possible for physical objects. In the field of modelling robots and their surroundings the most popular method of representing objects is by using polyhedra.

A polyhedron is a three dimensional solid figure with many faces. The faces are planar and the edges where faces meet are linear. Many common objects have polyhedral shapes or may be closely approximated by polyhedra. An example of a program which models robots and their environments by polyhedra is GRASP (Bonney 1985). An example of a GRASP model is shown in figure 2.3.

A polyhedron may be represented by a tree structure of edges, faces and vertices. A face may be defined by specifying its edges and an edge is defined by its end points. Clearly the more complex the polyhedron the more faces, edges and vertices it has and hence the more data which are required to define it.

A prerequisite for the robot path planning problem is the interference detection problem. Interference detection among polyhedral solids was addressed by Boyse (1979). To determine whether a polyhedron A intersected a polyhedron B, all the edges of A were tested to see if they intersected any of the sides of B. For example, consider the intersection check between C1 and C2 which are two cubic obstacles. Each of the twelve edges of C1 has to be tested with each of the six faces of C2. This gives a total of seventy two edge face tests. Also a test has to be done to see if C1 is enclosed by C2 or vice versa.

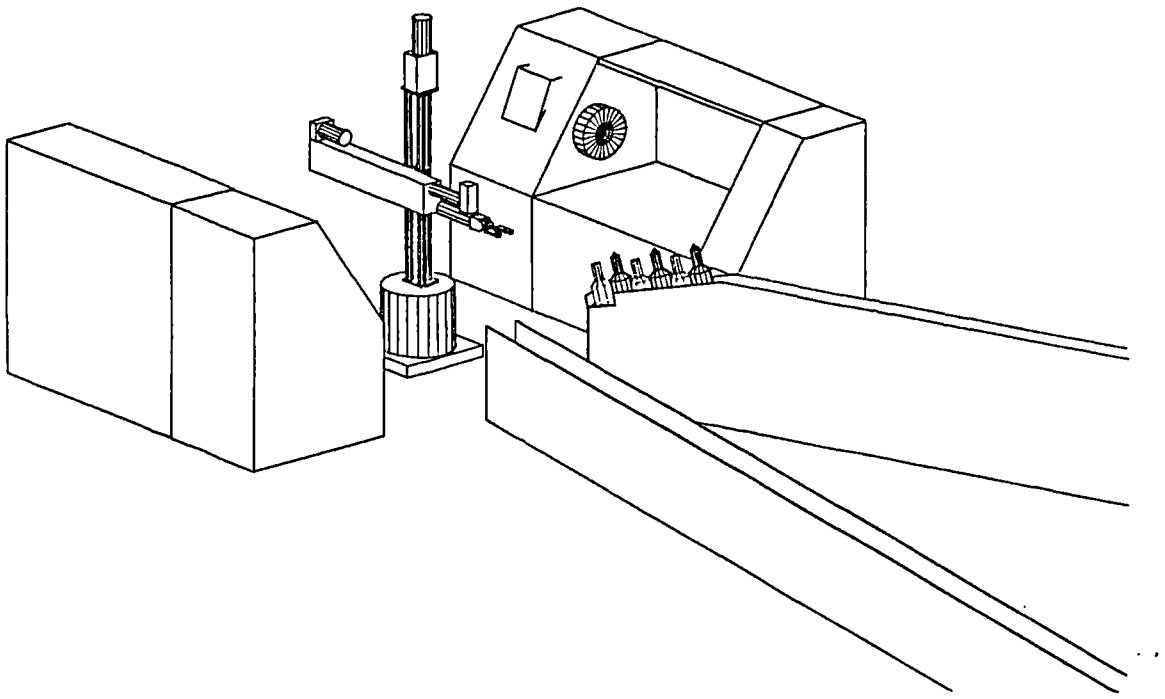


Figure 2.3 GRASP model (taken from Bonney (1985))

Solid modelling has been tried for representing the robot workspace (de Pennington 1983). De Pennington used Constructive Solid Geometry (CSG) provided by the the computer program NONAME. CSG models use simple shapes, called primitives, to produce complex and accurate representations of the robot environment. The primitives satisfy certain mathematical properties (Requicha 1980) so that operations such as volume calculations and intersection checking can be carried out easily.

Spatial occupancy enumeration is a subset of solid modelling. Space is divided into a matrix of spatial cells. Each cell is defined either as containing an obstacle or free space. Ahuja (1980) has shown that this method can be used to represent the path planning problem. A tree structure was used to represent three dimensional space. Space was represented as a solid cubic block. This was subdivided into eight blocks. Each block was tested and given a 'colour flag'. A block was designated black if it was completely within an object, white if it was free space and grey if it contained object and space. Each grey block was then

subdivided into another eight blocks. Recursive subdivision continued until a minimum sized block was reached. At this point any minimum sized grey blocks were designated as black.

To solve the collision detection problem using spatial occupancy enumeration the obstacle sets are calculated for the moving object or robot and its surroundings. To detect collisions the obstacle sets are compared and where two or more equivalent cells in the models are black then collisions are indicated.

The representation by a matrix of spatial cells has the advantage that it is convenient for computer storage. However the computing time required to generate the representations of the moving robot may be large.

For one of the earliest attempts at robot path planning Pieper (1968) used a world model consisting of two simple solid primitives, cylinders and spheres. He also represented surfaces such as table tops and the floor as planes. Cylinders could be joined to form composite obstacles. Spheres were assumed to be supported by planes. This simplified the intersection calculations as compared to the polyhedral and CSG representations.

Khatib (1986) produced a unique method of representing obstacles by mathematical functions. For a point on the robot, such as the centre of the end effector, the obstacle function (FIRAS) was evaluated to provide a value related to the distance away from the obstacle. The function tended to infinity as the point approached the surface and was zero beyond a certain distance from the obstacle.

This representation had the advantage that the task of calculating the distance between the robot and the obstacle was replaced by the task of evaluating the FIRAS functions which, compared to solid geometry or polyhedral methods, was relatively fast.

The main disadvantage was that only a limited number of obstacle shapes were available. As Katib (1986) stated 'this potential is difficult to use for asymmetric obstacles where the separation between an obstacle's surface and equipotential surfaces can vary widely'.

2.4 Obstacle transformations

Several authors have made a once only transformation of the manipulator and its surroundings into an abstract space. The purpose of creating an abstract space is so that the problem may be reduced to finding a path for a single point through a set of obstacles.

Udupa (1977(a)) enlarged obstacles by the width of the manipulator links to produce a 'primary map'. A transformation called Survey was applied which permitted the upper arm to be viewed as a point. The transformed space was called a primary chart. The primary chart was a map of all the positions of the end of the upper arm for which the upper arm was collision free.

A secondary map was produced by enlarging the obstacles by the radius of the forearm. The transformation Survey was applied to produce the secondary chart. The entire manipulator was now represented as a point on this secondary chart.

The advantages of these transformations were that the path planning of a point or single line segment was much easier in these transformed spaces.

Lozano-Perez (1983) developed a method for the calculation of paths for rigid polyhedral objects moving through space littered with other polyhedral objects. The method involved transforming obstacles into Cspace.

An example of how this method is used may be found in Red (1985). The configuration space for a PUMA robot with two degrees of freedom is calculated

by a VAX mini computer. The configuration space is displayed graphically and the operator can plan a path for a point through this space. The path is then converted back into robot coordinates for execution of the task.

The configuration of a three-dimensional object may be specified by a six dimensional vector. The six dimensional space of configurations for an object A is denoted by Cspace A. This contains all the information necessary to solve the findpath problem for A.

Lozano-Perez reported that when A was a three dimensional solid which was allowed to rotate, then a simple object B in real space became a complicated curved object in six dimensional Cspace A. So he did not calculate such objects, instead he approximated objects by a series of two dimensional slices containing polyhedral shapes.

Brooks (1983(b)) transformed the space between obstacles into freeways for the upper arm and payload of the robot. The two freeway spaces were searched concurrently with the constraint that the upper arm and payload were a fixed distance apart, due to the forearm.

Brooks reduced the degree of freedom of the payload in order to simplify the problem. He justified this by saying that for many operations the only reorientation of the part required is a rotation about the vertical axis. This may be achieved by wrist motions alone.

The algorithm generated prisms of freespace between obstacles. The obstacles were effectively only two and a half dimensional in that they had a two dimensional shape and a height. Thus a cube could be represented accurately but a tetrahedron could not.

Certain designs of robots cause difficulties when it comes to planning their trajectories. An example of this is the Stanford Arm whose boom is likely to

collide with obstacles both behind and in front of the robot. Luh and Campbell (1984) transformed their polyhedral obstacles in such a way that if any part of the arm would cause a collision in real space then the tip of the arm would also have a collision in the transformed space.

2.5 Path planning

The primary aim of a path planning method is to find a series of trajectories for a robot which will take it safely from one specified configuration to the next. Further than this, the path planning method should produce as efficient a path as possible, so that the robot does not waste time and energy in its movement and the computer calculation time should be as short as possible.

Several different methods of path planning have been tried in order to solve the problem. The approachs taken in each case have been governed by the obstacle and robot representations. Each path planning method may only be used with its own particular world model.

Gouzanes (1984) divided path planning methods into two categories, local methods and global methods. Although not all methods fit strictly into these categories it is useful to discuss them separately.

Figure 2.4 shows some examples of the work done in the field of path planning for robots. The polyhedral method of representing obstacles has been most popular. The path planning methods have been evenly spread between the local and global methods.

2.5.1 Local methods

Local methods proceed by proposing new configurations in the direction of certain strategies starting from an initial safe configuration S. The algorithm finds a path by repetitively moving the robot's configuration small distances towards

Author	Date	Path planning method	Obstacle representation
Pieper	1968	Local	cylinders, spheres
Udupa	1977	Local	Polyhedral
Ahuja	1980	Local	Polyhedral
Lozano-Perez	1981	Global	Polyhedral
Brookes	1983	Global	Polyhedral
Balla	1984	Local	CSG
Gouzanes	1984	Global	——
Kattb	1986	Local	Mathematical functions

Figure 2.4 The evolution of path planning

the goal. When obstacles are encountered alternative strategies are tried, such as 'move above' or 'move below' the obstacle.

For local methods the problem is treated as that of finding a series of closely spaced intermediate positions connecting the initial and final states.

Often if more than one obstacle is present, a move that appears good to avoid one obstacle will cause a collision with another. This may lead the manipulator to oscillate between objects. It is also possible for some joints to be at their physical limits so that the avoidance routine does not find an acceptable move. Finally, the avoidance routine itself may come up with a non-productive move. It is therefore necessary to ascertain continually whether or not progress is being made towards the goal. If no progress is being made it is then necessary to decide whether a slight change in strategy is in order.

Udupa (1977(a)) also used a local method for path finding. Udupa planned trajectories for the upper arm and forearm of the Scheinman Arm separately.

Firstly a trajectory was proposed for the upper arm directly between S and G configurations. Where collisions were detected, sub- goals were introduced which were intended to direct the path around the obstacles. For example, if a path

between A and B was tested and a collision occurred then a subgoal C between A and B was proposed. The paths between A and C and between C and B were then tested and so on, until either a clear path was found, or a time limit on the calculation was reached.

Having found the upper arm path, the forearm path was planned for positions where the forearm could collide with obstacles. Udupa described the method as 'juggling the forearm back and forth so that it avoids any collisions, as one end of it travels along the boom tip locus'.

Khatib (1986) used a very simple local path planning method. The manipulator moved in a field of forces. The obstacles were represented as repulsive surfaces and the goal as an attractive pole. The path planning method was to allow the summing of forces at each configuration to guide the robot to the goal.

This simple but effective method has allowed obstacle avoidance to be carried out in real-time. For this implementation two PDP 11 computers were used.

However the method could become trapped, failing to find the goal if a local point of minimum force was reached. This occurred if the robot was drawn between two obstacles where either no possible path existed or the robot had to pass close to the obstacles.

2.5.2 Global methods

Global methods may be applied only after the path find problem has been reduced to that of finding a path for a point through space. Lozano-Perez (1979) called this space, configuration space. Gouzanes (1984) reported that the actual path planning takes place in the subset of configuration space through which the point may pass. He called this 'empty space' (it has also been termed 'free space' by Brooks (1983)).

There are two approaches for finding empty space.

(a) Calculate the space occupied by obstacles and subtract this from the configuration space; examples of this are Lozano-Perez (1981) and Udupa (1977(a)).

(b) Calculate the empty space directly; examples of this are Brooks (1983), Gouzanes (1984), Chien (1984).

The choice of which approach to take depends upon the type of representation used, and whether space is expected to be cluttered or uncluttered with obstacles. It may be seen that the fewer the obstacles, the more efficient is method (a) and the smaller the empty space, the more efficient is method (b).

Lozano-Perez (1981) calculated the space occupied by obstacles by using a slice projection technique. Projections of the obstacles onto horizontal planes were calculated for a range of Z values. These obstacles were then transformed into configuration space by taking into account the size and range of orientations of the moving object. Cells of empty space in the projections were defined and a graph containing these cells was defined by considering the connectivity of the cells. Finally the path planning problem was solved by the graph searching method of Hart (1968).

The algorithm worked for cartesian manipulators only. Obstacles were polyhedral prisms whose axes were perpendicular to the horizontal.

Brooks (1983) modelled empty space as 'freeways' along which the manipulator could move. He separated the planning of the upper arm, forearm and workpiece. The upper arm planning was done in joint space. Figure 2.5 shows how joint space was divided into freeways along which the upper arm moved. Similar freeways were defined for the workpiece in real space. Figure 2.6 shows how a three dimensional freeway was defined between the prismatic obstacles.

A path was then found by firstly considering the path for the upper arm, secondly seeing which workpiece freeways could be used with the upper arm path

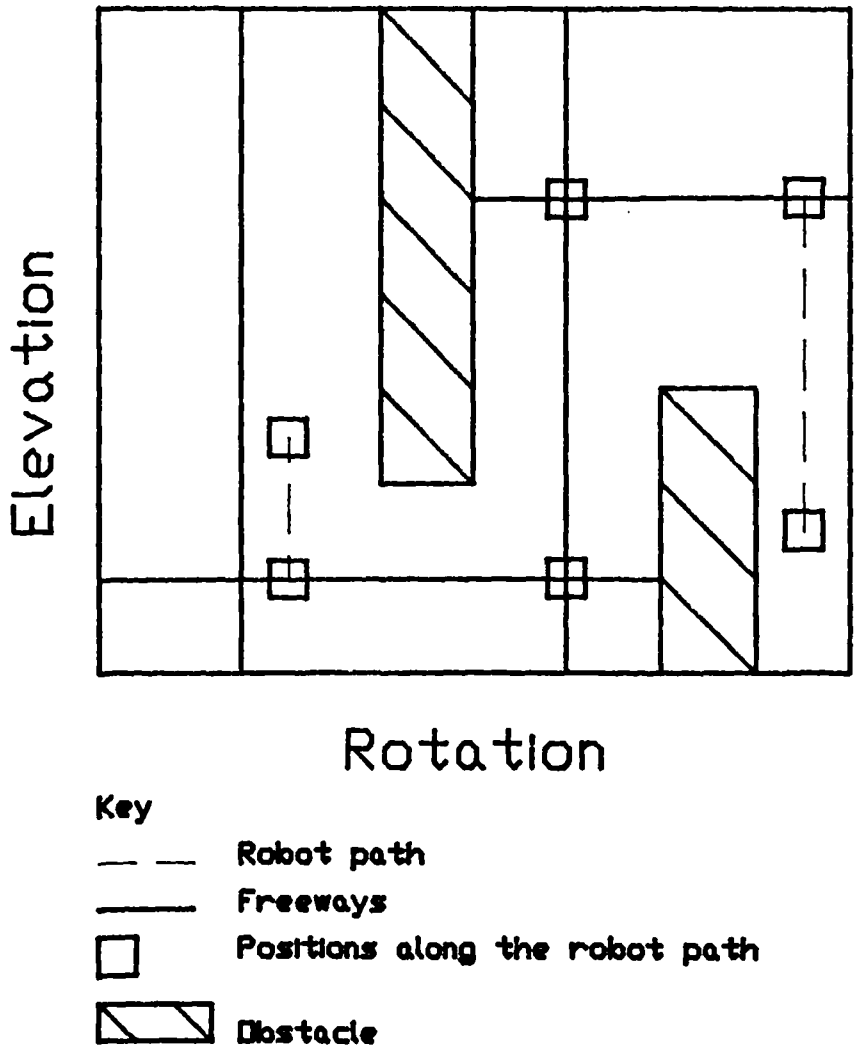


Figure 2.5 Free-space for upper arm planning (taken from Brooks (1983(b)))

and thirdly rejecting those paths which would cause a collision for the forearm.

This type of path planning produces paths which generally have good clearances from obstacles. This is advantageous from a safety point of view but disadvantageous for producing short paths, as the shortest paths generally pass close to obstacles. The method also greatly restricts the possible solutions because of the constraints which have to be applied to the movements whilst concurrently planning the upper arm and the workpiece in different representational spaces.

Chien et al (1984) used the concept of a rotation mapping graph (RMG) to plan paths for a rod, and then they extended the idea to cover the Stanford Arm.

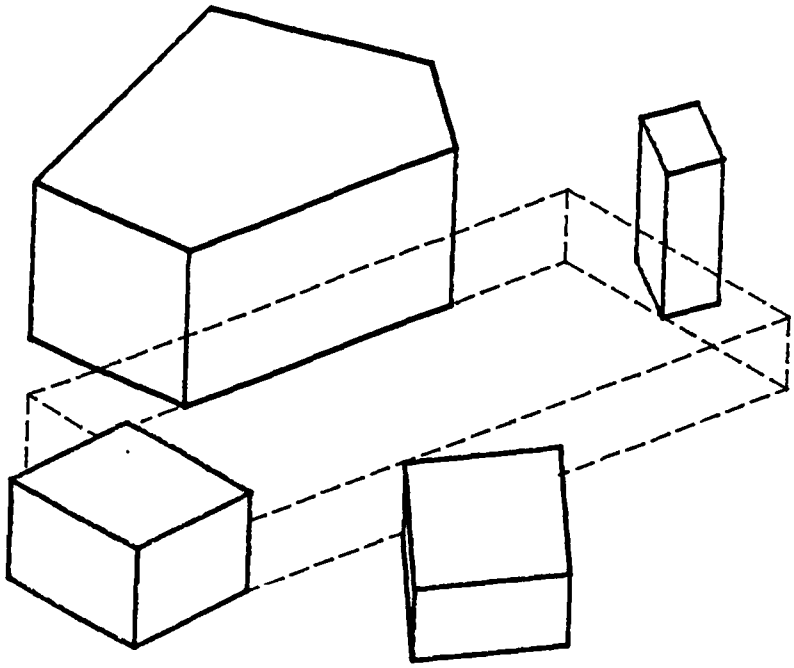


Figure 2.6 Free-space for payload (taken from Brooks (1983(b)))

They modelled empty space as regions of collision free motion for the forearm of the Stanford manipulator. These regions were limited to those which implied collision free motion for the upper arm. These regions were then converted into a graph for searching by using a connectivity algorithm. However, Chien did not comment on the implementation of the algorithm.

Luh (1984) calculated the configuration space for the boom (upper arm) of the Stanford Arm. The path planning for the boom consisted of planning a path for a point among a polyhedral representation of the configuration space. The shortest path for a point through this space is in straight lines between the edges of these obstacles. Luh presented an algorithm which, given an ordered set of edges, produced the minimum distance path. However, how to find which set of edges to use for the best path was not discussed.

2.6 The contribution of this thesis

With the exception of Khatib (1986) all of the robot path planning work

done so far has required greater computation time than the robot takes to carry out the trajectories. The importance of producing real time solutions can be classified into two main reasons.

(a) By attempting real-time solutions, the researcher is forced to develop efficient algorithms to solve the 'find path' problem. Efficient algorithms are vital for computing more complex problems which may be tackled 'off-line'.

(b) With the increasing sophistication of sensors, such as vision systems, it will soon be possible for robots to carry out more general tasks such as picking parts from bins, or fruit picking, for example. These tasks will require the automatic recalculation of paths in real- time.

The two methods discussed in this thesis provide a useful comparison of the two main classes of path finding algorithm, the local method and the global method.

The local path planning method produces real-time solutions for a wide range of problems. It requires no time to preprocess data before searching for a path and thus it can be used in situations where the environment changes frequently.

The global path planning method is based on a more rigorous mathematical treatment of the path finding problem. The method produces optimum solutions within the restrictions of its world model. It requires a small amount of time to preprocess the world model but then it produces real-time solutions to a wide range of path planning problems, where solutions are possible.

CHAPTER 3

THE DEVELOPMENT OF A ROBOT TEST RIG

3.1 Description of the robot test rig

The essential components of any automatic planning system are the robot, the robot control computer (RCC) and the path planning computer (PPC). A schematic diagram of the test rig is shown in figure 3.1.

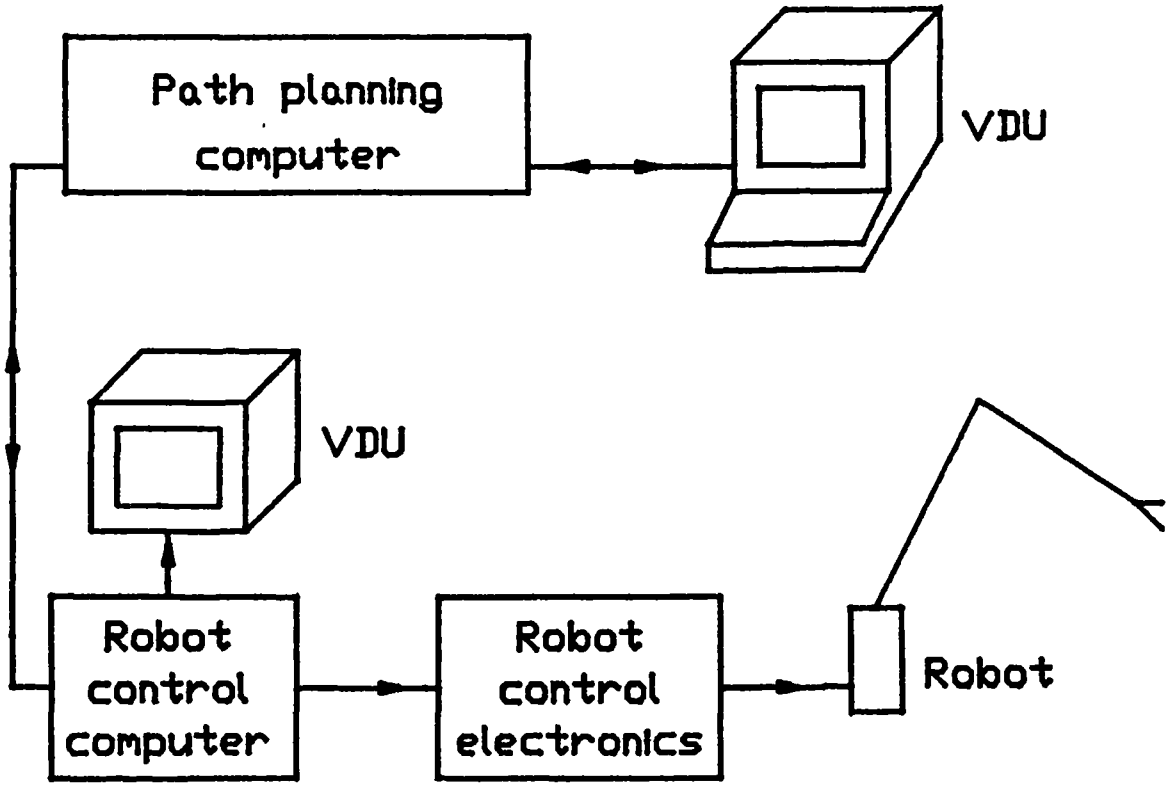


Figure 3.1 Schematic Diagram of System Hardware

The rig evolved out of equipment which was available for this research and hence the equipment used was not necessarily ideal for the purpose. The robot in particular had many disadvantages. However, it did have one great advantage which was that a copy of the robot software source code was available and the RCC facilitated the alteration of the software.

The robot was placed on a wooden base. A datum position of the robot base was recorded to enable accurate repositioning of the robot in case it was moved. The robot was not bolted down so that when programming mistakes occurred and the robot hit obstacles, the robot moved across the base without damage to itself or the obstacles. For early development work the gripper was removed when it was not required for tests, as this was the most delicate part of the robot.

Test obstacles were constructed out of thick cardboard. Although this was viewed as a 'sealing-wax and string' approach it was very successful, as different models could be constructed quickly and at no cost. A grid of lines was marked on the wooden base so that obstacles could be accurately positioned on the grid. This also aided the measurement of robot configurations.

3.2 Form of robot data

The robot software was designed such that the robot path data consisted of a number of blocks of data. Figure 3.2 shows a block of robot data as it was displayed on the RCC.

STEP 1)

Rate	Mode	Input	Output	Wait	Jump
5	2	0	0	0	0
500	250	50	500	500	0
rotate	shoulder	arm	wrist	hand	gripper

Figure 3.2 A block of robot data

The block of data contained twelve pieces of information.

(1) Rate :- The rate at which the RCC indexed the position of the robot joints when moving.

(2) Mode :- Different modes were available for special activities such as 'search for a part'.

(3) Input :- The robot had several electrical inputs and it could be programmed to wait for an input to be activated.

(4) Output :- The robot could activate its own outputs.

(5) Wait :- Wait for a number of seconds.

(6) Jump :- Continue executing the path at a specified line number.

(7-12) Coordinates of robot axes.

The operator taught the robot trajectory programs by moving the robot physically through different positions using keypad control. Each position was recorded as coordinates in joint space in a block of data. Other information such as the rate of movement, jump statements etc. was recorded in each block as well.

3.3 Transfer of data

The transfer of data was achieved by using a serial link. This was chosen because it was robust and easily produced. A procedure in the PPC converted the path from a series of positions in joint space to a series of equally spaced robot coordinates. These numbers were sent in ASCII characters to the RCC. A procedure in the RCC decoded the ASCII characters and stored the numbers in memory.

Although both the calculated path and the robot coordinates in the RCC were joint space coordinates they used different scales and origins for their co-

ordinates. Thus a coordinate transformation in the form of equation 3.1 was used.

$$Bi = l.Ai + k \tag{3.1}$$

where

Bi	is the RCC coordinate for joint i.
Ai	is the PPC coordinate for joint i.
l and k	are constants.

3.4 Modification of robot software

The robot control program was the result of at least two man years of development work by the robot manufacturers and it involved many different features. The program was so long that when it was loaded it occupied all free memory on the computer. The organisation of the program routines was closely tied to the method of programming the robot which was done using a teach pendant.

To enable the PPC to program the RCC, the software in the RCC was considerably altered. The program was changed so that data could be loaded from the serial link and the robot path executed without the need for operator intervention. The program was first modified by erasing several procedures such as 'Tutor text' in order to make greater space for the program code.

The manufacturer's program required the robot software to go through an initialisation process which included driving the robot to a park position. The program then went into an edit mode which allowed the programmer to teach the robot a new path.

The initialisation procedure was retained but the park position software was deleted. After initialisation of variables the program read in the trajectory data

from the PPC via the serial link. This procedure was implemented in assembly language so that data transfer was as fast as possible. Having read in the trajectory data, the program mode was set to run and the step counter was set to the start of the trajectory data. When the trajectory was completed the program returned to the 'input data' procedure.

3.5 Investigation of robot properties

The properties of the test robot were investigated for two reasons, these were:

(a) In order to assess the usefulness of automatic programming.
(b) To give a guide as to what minimum clearance the robot should give to an obstacle when manoeuvring around it.

Three robot properties were investigated.

(i) The ability to repeat a previous position when approaching from the same direction and with the same speed.
(ii) The ability to repeat a previous position when approaching on a new path.
(iii) The ability to move to a specified coordinate position.

The first two are generally called the repeatability of the robot and the third is called the accuracy of the robot.

For automatic or off-line programming it is the accuracy of the robot which is important, whereas for normal teach-pendant type programming it is only the repeatability (first type) which is important. Hence if the robot's accuracy is not as good as its repeatability then there will be tasks which an operator can program the robot to carry out but which the automatic programming system cannot.

For the first experiment a light source was attached to the end of the robot forearm. This produced a beam of light which was focussed on a piece of graph paper a small distance away from the robot (see figure 3.3). The robot was programmed to move to another position and then return again. Having returned, the position of the light beam was recorded. This procedure was repeated ten times.

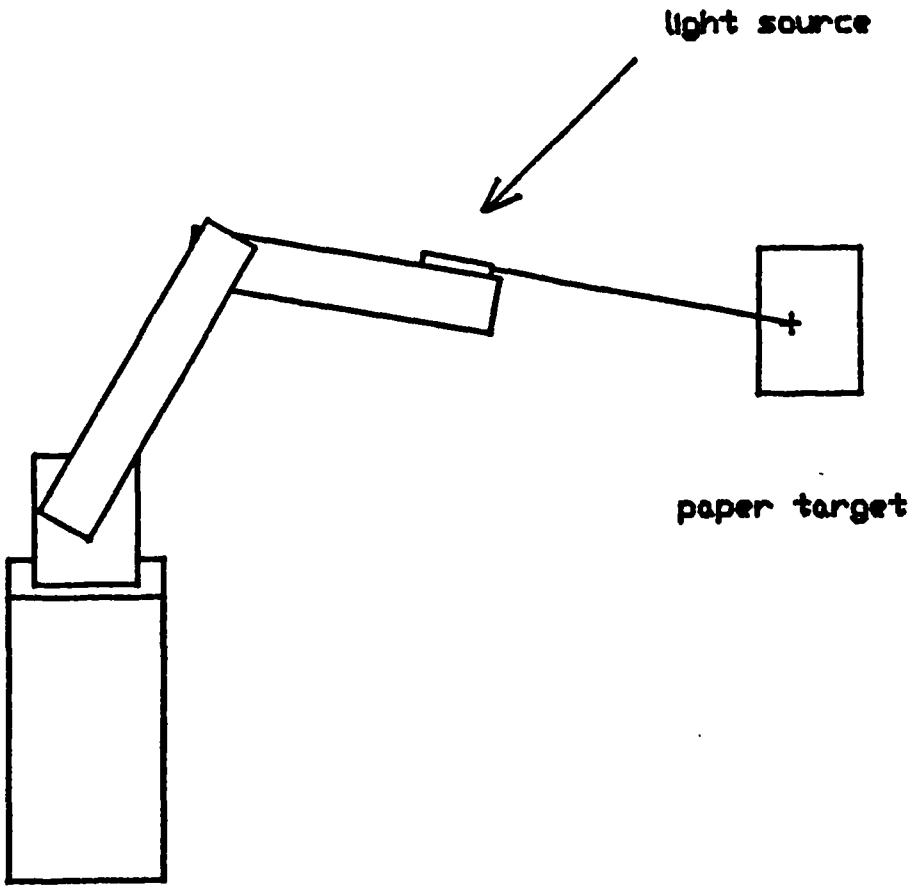


Figure 3.3 Repeatability experiment

The second experiment was similar to the first experiment, except that the robot moved to a different position away from the test position for each test.

For the third experiment the tip of the robot was moved to ten different coordinate positions. The position was measured by suspending a pendulum from the tip of the robot. The x and y coordinates were measured at the end of

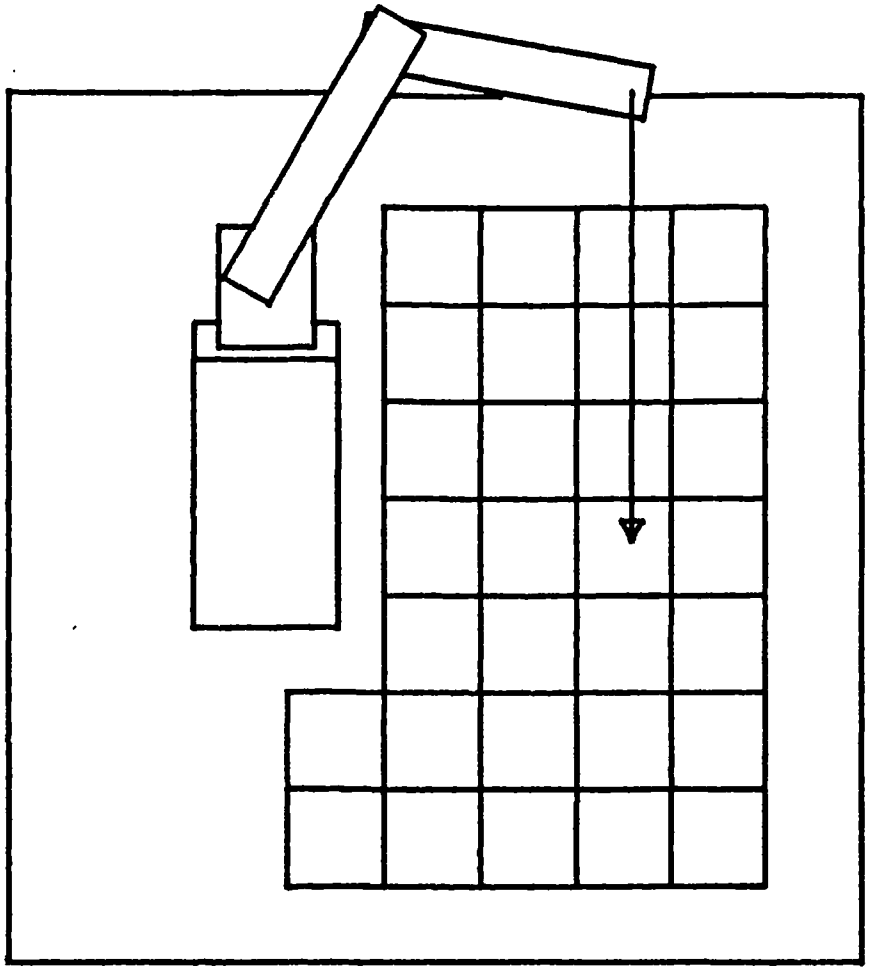


Figure 3.4 Measurement of robot accuracy

the pendulum and the z coordinate was given by the length of the pendulum (see figure 3.4).

The first two experiments were carried out several times for different test configurations. It was found that the repeatability of the first experiment was +/- 1.5 mm. However for the second experiment it was +/- 10mm. The difference in these figures was probably due to the back-lash in the robot joints. This would always be in the same position for the first experiment but not in the second.

For the third experiment the accuracy was also found to be +/- 10mm.

3.6 Discussion

Many practical problems occurred with the equipment. These may be expected to recur when other off-line programming systems are developed. In particular, the robot software and the robot mechanical design created problems which were overcome.

The robot mechanical design was difficult to model for the path finding problem as it had overslung links. This produced the possibilities of left handed and right handed configurations and also provided more difficult coordinate transforms. These are described in section 4.5.

The robot software had no facility for off-line programming. This meant that separate routines had to be written for data transfer and entry into the RCC program and the automatic operation of the robot as soon as data had been received. This proved to be a time consuming task as the robot control program was necessarily complex.

For this development work no standard data protocol was available which could have been used to transfer data. However with the increase in popularity of Manufacturing Automation Protocol (MAP), (European MAP users group secretariat (1986)) it should be possible to obtain robots which use this protocol and link into them much more easily.

It was found that the robot properties had to be closely investigated before automatic programming could be implemented. This was done in order to set safety margins on the minimum distance allowable from the robot to obstacles and to assess the feasibility of some operations.

The types of operation which may be programmed off-line or automatically for a particular robot are affected by the accuracy of the robot. For example the test robot accuracy of +/- 10mm would be unacceptable for many operations such

as loading a part into a vice. However the repeatability of +/- 1.5mm enables a human operator to program this task.

Other inaccuracies also affect automatic and off-line programming. These are the positional accuracy of the robot and the objects in the robot's workspace, relative to the computer world model. For example, consider the task of putting a part into a fixture. The possible errors come from the positioning of the robot on the workcell base, the positioning of the part in the robot gripper and the positioning of the fixture in the robot workspace. These errors have to be added together in order to assess whether the robot can achieve the task. For instance if an operation requires an accuracy of +/- 0.5mm, the positioning accuracy of the robot on the base is +/- 0.1mm, the positioning accuracy of the fixture is +/- 0.1mm and the accuracy of the position of the part in the gripper is +/- 0.2mm, then the accuracy of the robot must be 0.1mm or less.

CHAPTER 4

REPRESENTATION OF THE ROBOT AND ITS SURROUNDINGS - THE WORLD MODEL

4.1 Introduction

The following list gives the requirements which are important in methods used to represent the world model of a robot and its surroundings.

(a) Fast intersection calculations.

(b) Easy to use with path planning algorithms.

(c) Easy to generate a model.

(d) Low memory storage requirements.

(f) Efficiency in terms of the workspace volume occupied at critical points.

The world model may be divided into two parts, the robot which incorporates all moving objects and the surroundings which incorporate all stationary obstacles. Moving parts have different modelling requirements from stationary objects so the two parts of the world model are dealt with separately in this chapter.

4.2 The surroundings

Several different methods of modelling surroundings were considered. These were polyhedral models, constructive solid geometry models, surface models and models consisting of spheres. For each representation the five requirements listed above were assessed.

Most published computer models of robot surroundings are in the form of polyhedral obstacles. This geometry is chosen because most obstacles tend to

have flat surfaces and straight edges. However these model forms can be difficult to deal with in path finding calculations. Figure 4.1 shows a situation, where a robot has to find paths avoiding two box shaped obstacles. In this case the robot and the obstacles are all polyhedral objects and so a polyhedral model would imply a high degree of accuracy.

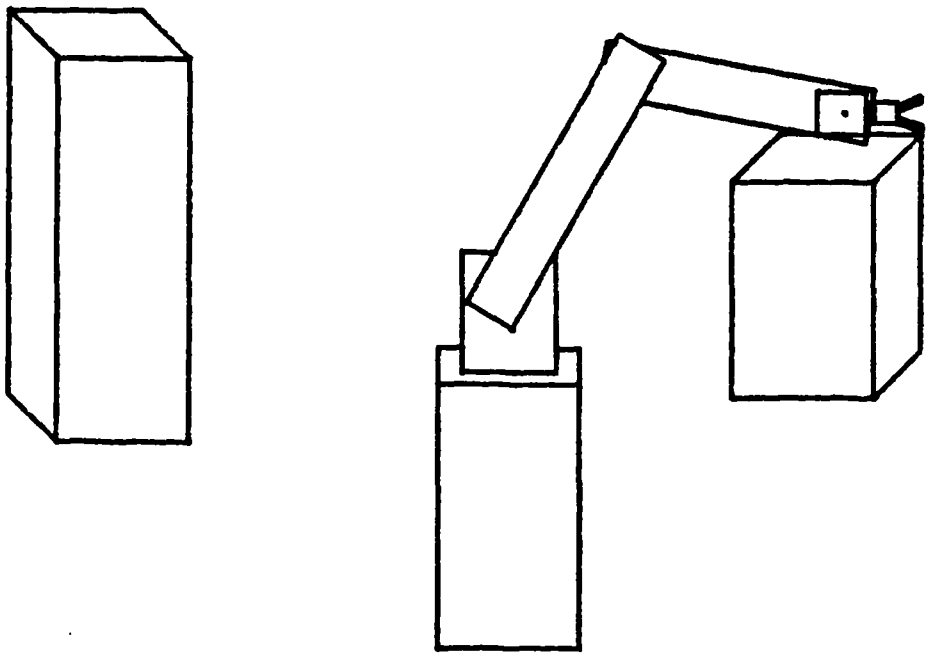


Figure 4.1 An example of a robot workspace

Constructive Solid Geometry (CSG) representations are (ordered) binary trees. Figure 4.2 shows an example of a CSG tree which represents a "U" shape. Non-terminal nodes represent operators, which may be rigid motions, regularised union, intersection, or difference. In the example nonterminal nodes are a union (U*) and a rigid motion (translate). Terminal nodes are either primitive leaves which represent solid primitive shapes, or transformation leaves which contain the defining arguments of rigid motions. In the example, P1 and P2 are solid primitive shapes and there is a translation WX of P2. More information about this may be found in papers by Requicha (1977), Braid (1973) and Braid (1975).

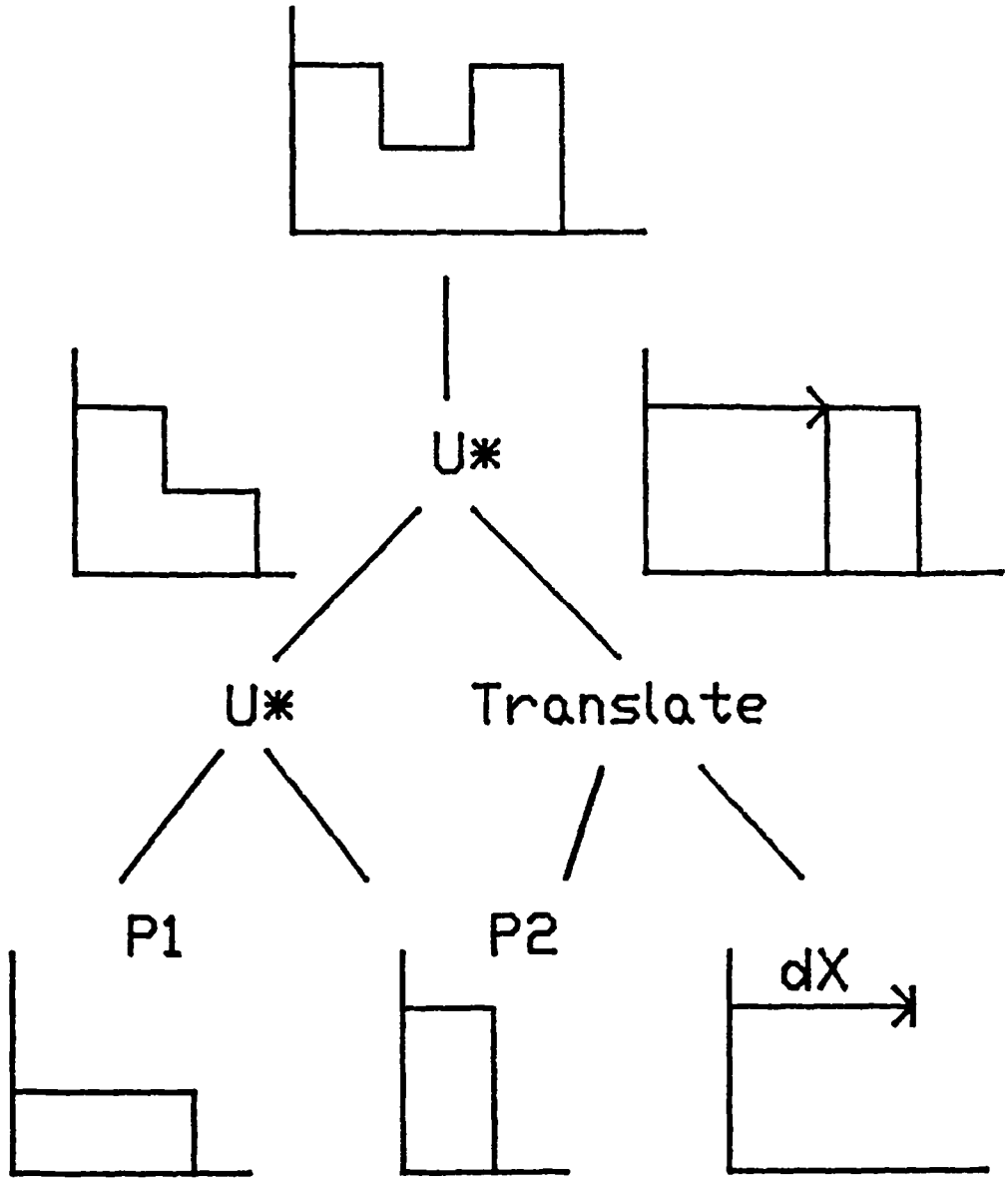


Figure 4.2 Diagram of a CSG tree

Surface modelling methods may be used to model environments in which robots operate. These programs may be used to model complex surfaces in great detail, and are widely used in the design of complex shapes such as telephone hand sets and plastic bottles.

An introduction to surface modelling is given by Ball (1983). Surface modellers use complex parametric functions such as Bezier equations to represent the

detail of surfaces. These representations are difficult to use for intersection checking. As surfaces only are represented and not the volume beneath the surface, it is difficult to determine whether a particular point in space is inside an obstacle or not. Deciding whether two surfaces intersect is also difficult because of the representation by parametric functions.

The method chosen in this work was to model the surroundings as collections of spheres. This method fulfilled requirements 1 to 4 above better than the other methods and it was felt that requirement 5 would be adequately met.

4.3 The robot

The requirements for the robot model are similar to those for the surroundings. The most important factors are efficiency in terms of workspace used, ease of use for intersection calculations and low memory storage requirements.

Shapes of robot vary a great deal; some robots are bulky and capable of limited motion, others are slender and capable of moving through a wide range of configurations. The robot model chosen must be conservative in that the whole volume of the robot must be contained by the model, but the model may be larger than the real robot.

The robot arm was modelled as two connected cylinders with hemi spherical ends. The advantages of this representation were that the cylinders modelled the robot links efficiently and the intersection calculations between the robot arm and obstacle spheres were very fast. The calculations consisted of finding the distance between the centre of the sphere and the closest point on the arm centre line. From this distance was subtracted the radius of the arm, to give the distance between the arm surface and the sphere surface.

A large number of robots have a similar design to that of the test robot. These robots have two major links, the upper arm and the forearm, which have slim profiles so that they can operate in restricted environments. Thus the model chosen for the test robot may be applied to a large number of different robots.

An accurate model of the robot was obtained by measuring it. This would be necessary for each individual robot as even robots of the same design vary slightly from each other. The dimensions were then incorporated into the model. The physical limits of the robot's different axes of motion were also incorporated into the model to prevent the path planning algorithm from producing impossible paths.

The robot contained some features which were difficult to model. In particular the lateral property of the robot made path planning more complex and the protruding cables were very difficult to model. It would have been possible to redesign and rebuild the robot to remove these features, however in the case of the lateral property it was decided that the problem should be tackled, as it applied to many robots. The fact that the robot had protruding cables was ignored as most commercial robots did not have protruding cables and those on the test robot could have been eliminated by minor design alterations.

4.4 The gripper and the workpiece

The gripper and the gripper motors were modelled as a sphere. From figure 4.3 it may be seen that the sphere centre was displaced from the axis of the upper arm. The sphere radius was just sufficient to enclose the main part of the gripper, the gripper motors and the attached cables.

The gripper fingers, if empty, were modelled by a smaller sphere. The workpiece was modelled as a series of spheres, depending on its size. It was found that

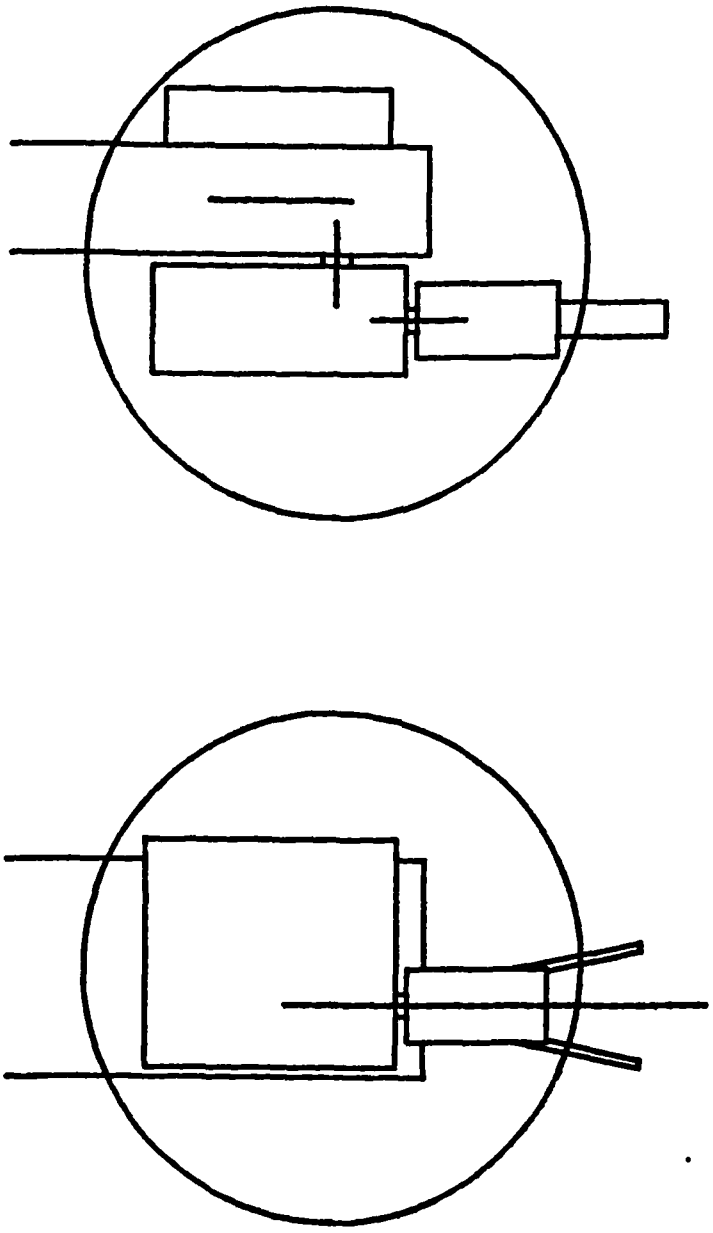


Figure 4.3 The robot gripper

many small workpieces could be modelled as a single sphere and that this also enclosed the gripper fingers.

As the path planning algorithm was not designed to take account of reorientation of the gripper to avoid obstacles, the position of the gripper was fixed for the mid-phase planning. The gripper position relative to the forearm was defined such that the gripper axis and the forearm axis were parallel.

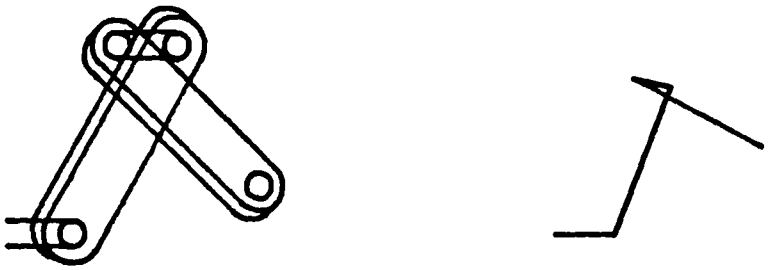
4.5 The lateral property

The links of the test robot did not lie in the same plane. This is illustrated by the kinematic model in figure 4.4. This property is called the lateral property and robots may be defined as left handed or right hand dedending on their current configuration. Figures 4.4(a) and (b) show the kinematic chain for the test robot. However, (a) is the right handed configuration and (b) is the left handed configuration.

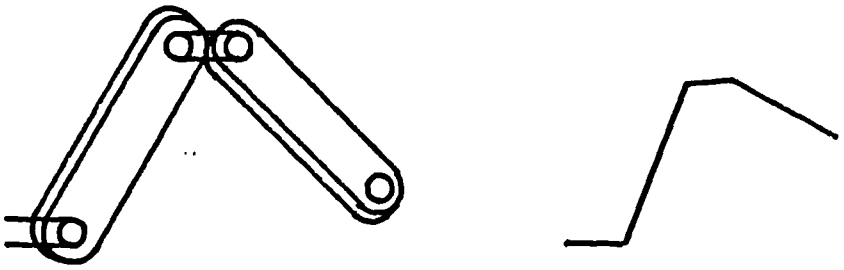
Figure 4.4(c) shows the model kinematic chain used to represent the test robot for path planning purposes. In order to use this model the obstacles were transformed such that a collision in the real space also caused a collision in the transformed space.

In order to take account of the elbow of the robot, extra spheres were proposed in the upper arm model. These extra spheres ensured that the path planned for the upper arm was also free for the whole of the elbow joint.

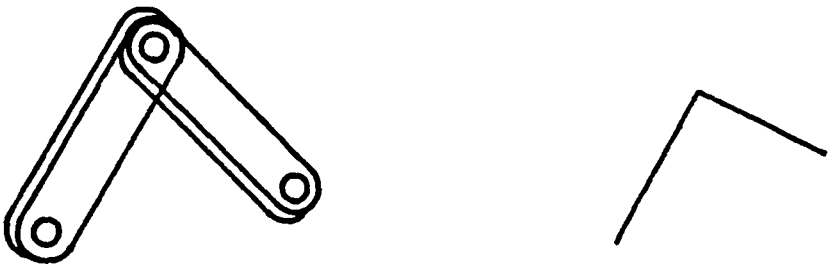
Figure 4.5. shows a flow chart of the program for creating these 'new spheres', called Expobs. Each of the existing spheres in the model was considered in turn. The distance to the centre of the sphere was calculated. This was compared with the maximum and minimum distances to the elbow and the radius of the sphere, to determine whether a collision was physically possible. If a collision was possible a new sphere was proposed, such that the upper arm intersected the new sphere for all configurations where the elbow intersected the previous sphere.



(a) Right hand configuration



(b) Left hand configuration



(c) Model kinematic chain

Figure 4.4 Kinematic robot models

The new sphere was then put into the list of obstacle spheres and the next one tested.

In order to take account of the lateral property of the manipulator the obstacle spheres were transformed onto new spaces. Firstly the configuration at which the axis of the robot arm passed through the centre of the sphere was found. The

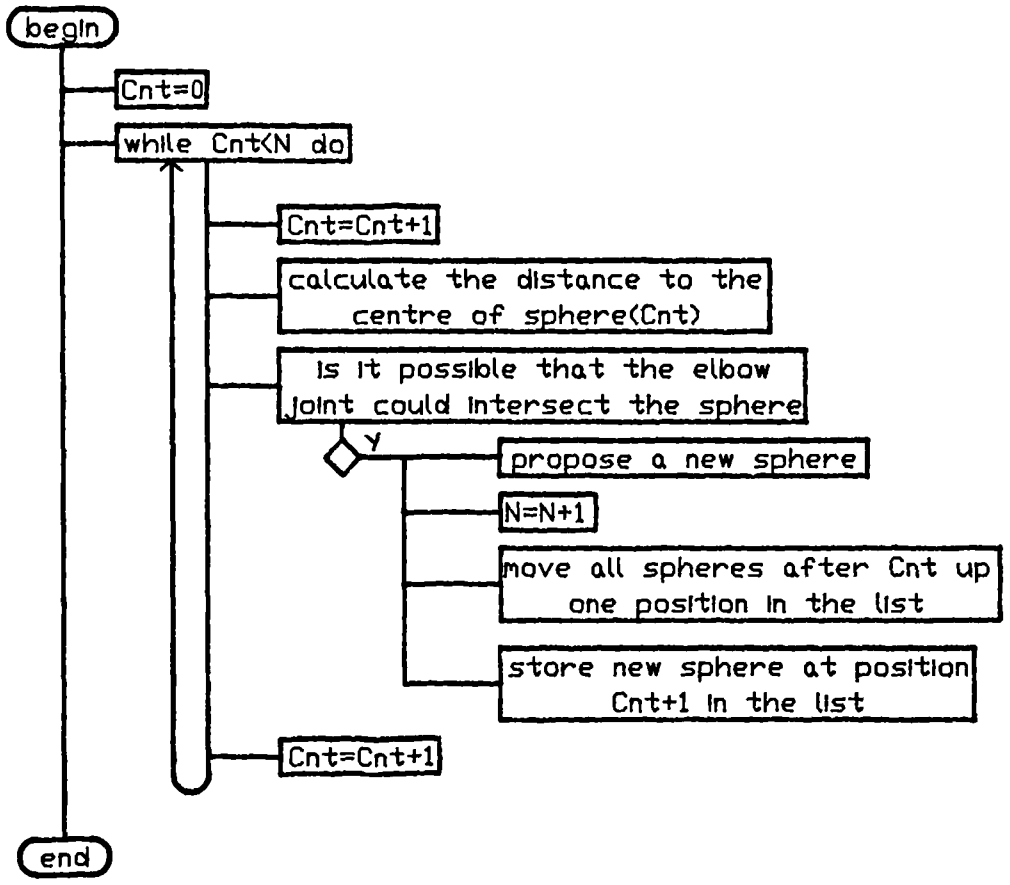


Figure 4.5 Flowchart of Expobs

sphere was then translated by the vector equivalent to the offset of robot link from the origin. The sphere was then enlarged slightly to take account of the reduced distance from the origin.

Although the transformation is a mathematical approximation of the real situation it was found to be satisfactory for practical path planning.

4.6 Efficiency of sphere models

In order to quantify the difference between the model of spheres and the real objects, the volumes of the spheres and real objects were compared.

The volume of the real objects must be completely contained by the spheres for safety reasons. Therefore the model will always have a larger volume than real objects.

Any shape may be modelled by spheres and to any accuracy, however, the greater the accuracy required, the larger the number of spheres needed and the larger the number of spheres, the longer the computation time. Thus the accuracy of a model is limited by the computer memory available and the computation time permitted for the path finding algorithm.

As the real environment for a robot becomes more complex so more spheres are needed for the model. It is important to know how increasing the number of spheres might increase the accuracy of the model.

In order to find out how increasing the number of spheres affects the accuracy of the model, the case of modelling a unit cube was investigated. A cube was represented by a cubic number of spheres ie. 1, 8, 27, 64 etc. The spheres formed a regular pattern and were equal in size. The volumes were deduced from calculations which have been given in appendix A.

Figure 4.6 shows a graph of the volume of the sphere model vs the number of spheres used. From the graph it may be seen that an infinite number of spheres is required to model the cube completely.

Modelling objects using the same sized spheres is inefficient. For instance in modelling a cube by 64 spheres of the same size 8 of the spheres are totally enclosed within the cube and might easily be replaced by a single larger sphere with no increase in model volume.

In general, deciding on the best sizes and positions of spheres to model real obstacles is difficult and no rules have been developed to do this automatically (to the author's knowledge).

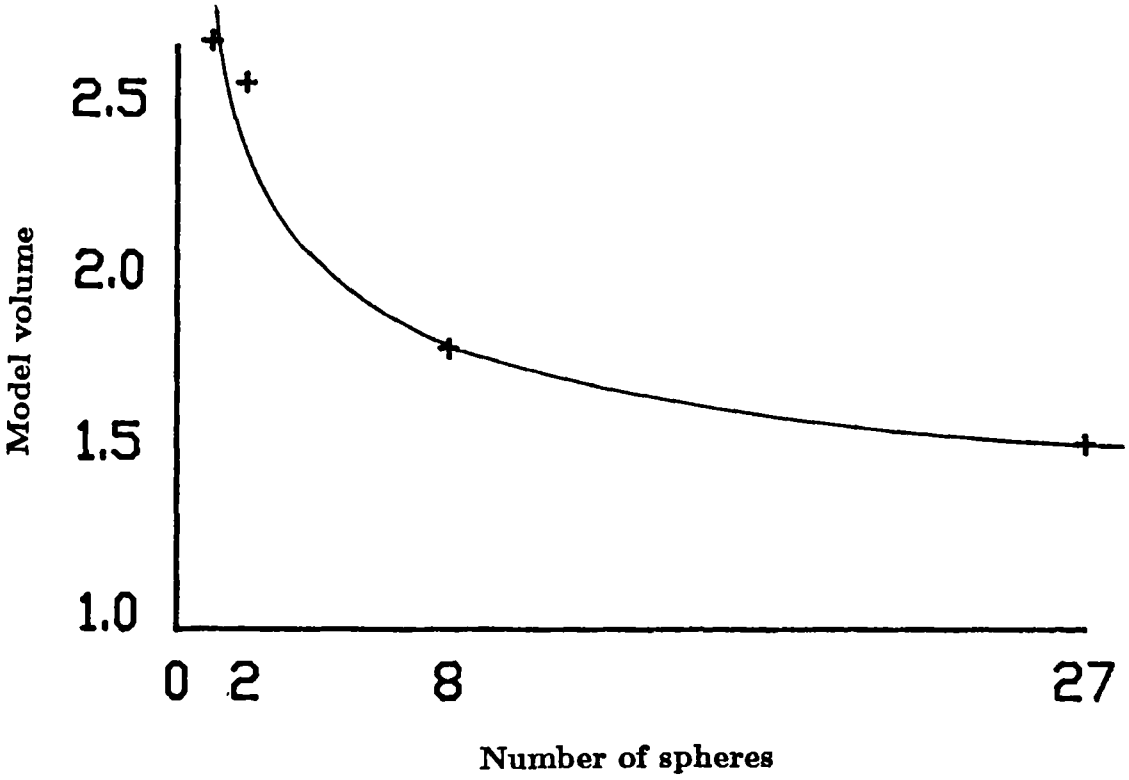


Figure 4.6 Volume of spheres vs number of spheres

In practice typical numbers of spheres used to model an obstacle were between 1 and 8. This made the model simple and speeded up path calculation. It also required little computer storage space and produced efficient robot paths.

4.7 Conclusions

Polyhedral, CSG or surface modelling methods may be used to represent the world model accurately. However they are complex models requiring complex intersection calculations to determine whether the robot can move to any particular position. The models also provide difficult problems for heuristic algorithms and the generation of these models is time consuming.

The sphere model on the other hand ensured the fastest possible intersection calculations. The calculation was reduced to finding the distance from the robot

to a point and subtracting the radius of the sphere to give the distance to the surface of the sphere.

Heuristic algorithms were made simpler by the use of spheres, as the distance and direction of the robot to the nearest obstacle was easily calculated. Thus directions could be modified heuristically to avoid collisions.

It was found that the sphere model used was more difficult to visualise than the other types of model, however models could be quickly built up and entered into the computer. Figure 4.7 shows one possible model of the real environment of figure 4.1. The storage requirements for complex models were low as each sphere required only four items of data, the three cartesian coordinates and the radius.

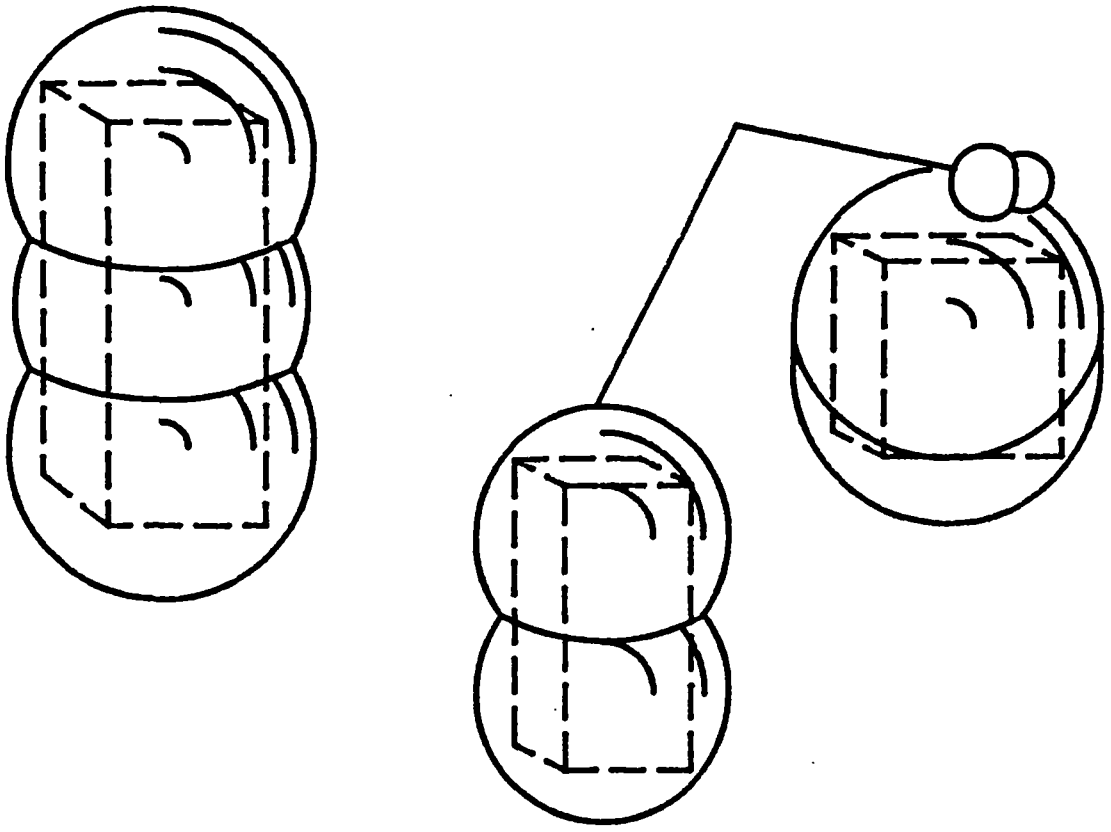


Figure 4.7 Model of robot and obstacles

The detail of the modelling of an obstacle can be tailored to its importance, for instance a cubic obstacle which is away from likely paths can be modelled as a single sphere. Items which are more critical can be modelled as greater numbers of spheres. Thus although the workspace volume is reduced, by this method, the critical workspace for pathfinding is not significantly affected.

CHAPTER 5

PLANNING PRELIMINARIES

5.1 Planning in a flexible manufacturing environment

In flexible manufacturing systems the control of a robot may be divided into hierarchical levels. This is important in order to integrate a robot into a system of machines. Further information about integrating robots into automated systems may be found in the following references, Albus (1982 and 1983), Arai (1982), Smith (1983), Gaspart (1982), Fussell (1983) and Cassinis (1983).

For this research the control of the robot was divided into four levels. These were :-

(i) Task description,

(ii) Trajectory description,

(iii) Robot coordinates,

(iv) Power to robot.

A task description contains general information about what the robot must do. An example would be 'pick up part X from position W, put the part into machine Y. Wait until the machine has finished. Remove the part and place it at Z'.

A trajectory description is a mathematical representation of the robot's path which will fulfill the requirements of task description. The robot coordinates are generated from the trajectory description; these coordinates generally refer to the robot joint angles. These coordinates may be transferred directly to the robot controller. The robot controller then converts the robot coordinates into the motive force (electrical, hydraulic, etc) which carries out the required movement.

For this work the task description was provided by a human operator who typed the required start and finish coordinates of a part into the path planning computer. This first level of control was then converted into the second level, the trajectory description, by the path planning algorithms. The trajectory description was then converted into the robot coordinates which were the third level of control. These in turn were converted into robot movements by the robot controller.

5.2 Introduction to planning

The planning problem was divided into three stages following the method of Udupa (1977(a)). These were path feasibility, approach path planning and mid-phase planning. The most difficult of these was the mid-phase planning, which is addressed in chapter 6.

The configurations of the robot at goal positions along the path were checked for feasibility. Positions which were out of the robot's workspace, or which would cause collisions with obstacles, were clearly unacceptable.

Approach paths are paths which move from positions with good clearance from local obstacles to end positions such as the start position (S) and the goal position (G). A path which moves to S, from a position clear of obstacles Sm, may also be followed in reverse when moving from S, to Sm. The positions which are clear of obstacles are generally close to the end positions so that the approach paths are short.

When a trajectory was planned, a series of intermediate configurations were produced between goal configurations in order to take the robot around obstacles. The way in which the robot moved between these configurations was called the path definition. The path definition was critical to the trajectory locus and the path efficiency. It had to be defined before any path planning was done as it

affected whether the direct path between two configurations hit an obstacle or not.

The path planning algorithm attempted to minimise the cost of the robot path. There were many different criteria which may be considered when estimating the cost of a path for a robot. Five of the most important criteria which may be considered are listed below.

(i) The distance travelled by the robot.
(ii) The time taken by the robot.
(iii) The energy used by the robot.
(iv) The wear on the robot.
(v) The safety of the path ie. how close does the robot come to obstacles.

The weighting given to each criterion in assessing the cost of a path must be decided before the planning can take place, so that where choices of different paths exist the most efficient path can be chosen.

5.3 Path Feasibility

The task description was divided into a series of configurations through which the robot moved in order to carry out the task. When moving from one configuration to the next, only the latest configuration was checked for feasibility, as the robot had already reached the previous configuration.

If this check had not been made in the planning program then valuable calculation time would have been wasted in attempting to plan impossible tasks. It was found that the small amount of extra calculation time and the few lines of program code, which were required to carry out the feasibility check, were worth while as a programmer may easily make the mistake of asking the robot to move to an impossible position.

5.4 Approach path planning

Approach path planning requires special knowledge of the environment and details of the chucks and grippers of the machines. As an example, consider the manipulation of a component into a lathe. The part must be lined up with the central axis of the chuck and must move along this axis until it is at the required position in the lathe chuck jaws.

Approach paths may be considered to be related to machine geometry, and calculated for the specific machine configuration. This part of the path planning was therefore separated from the mid-phase path planning and is treated differently. The user defines an approach path by giving the following information:

(a) The orientation of the part for the approach path.
(b) A vector defining the direction, length and position of the approach path.

This information is machine and part specific. Special programs were written to calculate the approach paths from the above information.

5.5 Interpolation between configurations

A path was defined as a series of robot configurations. The robot trajectory was derived by interpolating the robot coordinates between these configurations. The way in which the interpolation was carried out affected the way that the robot moved, whether it hit obstacles or not, and factors such as the speed of movement.

The section below shows an example of two types of interpolation used for the test robot which had different properties.

5.5.1 An example of two types of interpolation

For this example the robot upper arm is considered. It may be idealised for simplicity as a line segment attached at one end to the origin. Figure 5.1 shows how the robot has two degrees of freedom, T1 and T2.

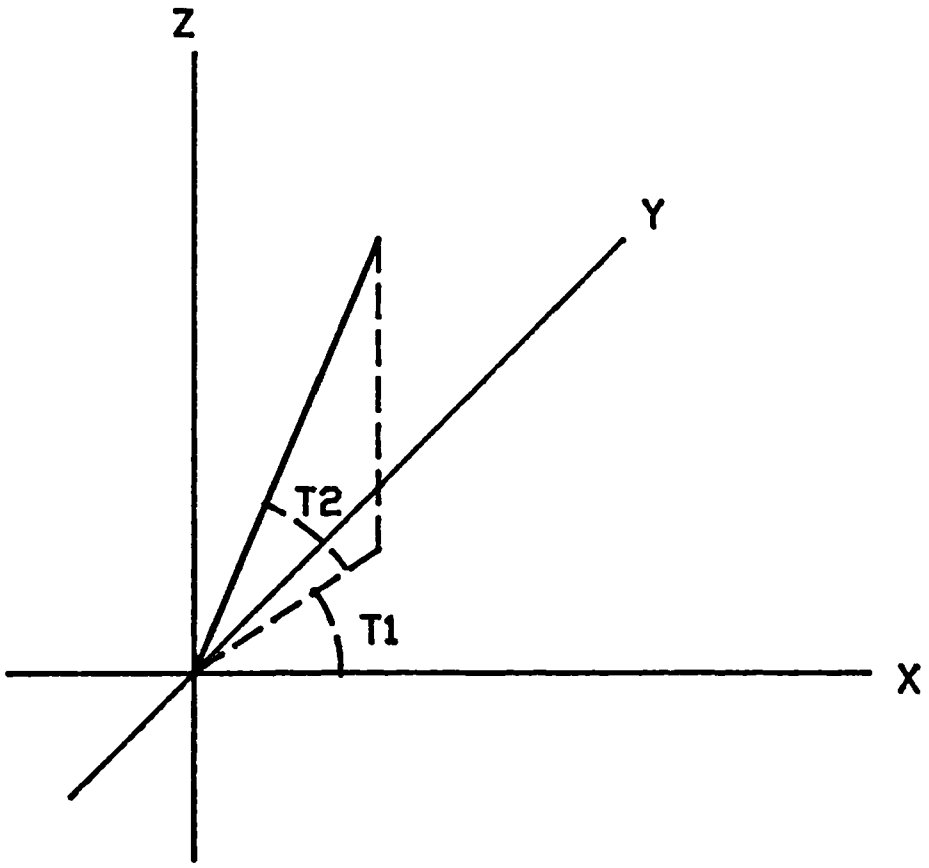


Figure 5.1 Idealised robot upper arm

Figure 5.2 shows how two different types of interpolation produce two different paths between configurations A and B. The figure represents a sphere with lines of longitude and latitude marked on it. A line from the centre of the sphere to its surface defines the configuration of the idealised robot upper arm. Configurations A and B have the same latitude. For a path where longitude and latitude are interpolated path 1 is followed. If the path is defined so that a line from the surface to the centre of the sphere moves in a plane then path 2 is followed.

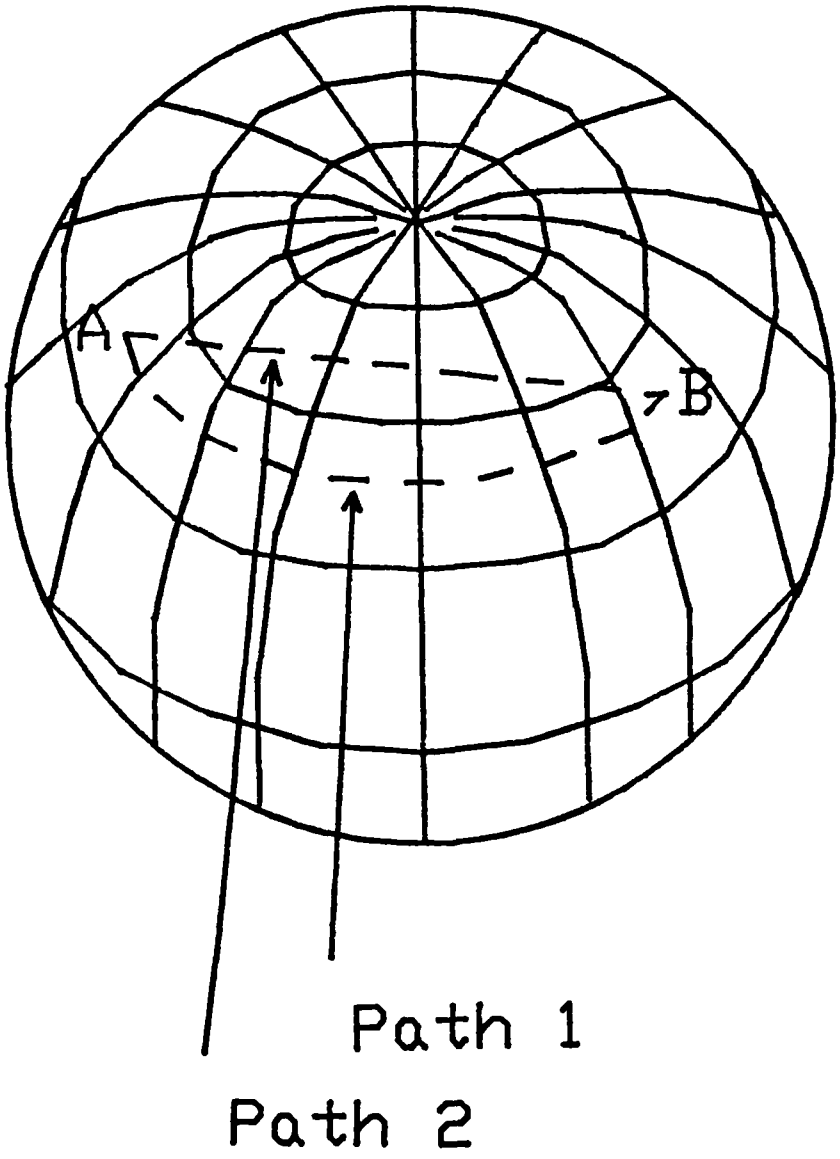


Figure 5.2 Different types of interpolation

Both methods of interpolation have advantages. Path 1 has the advantage that it is easy to calculate the interpolated position at any time between when the robot leaves A and when it arrives at B. Path 2 is the shortest path for any given point on the robot. It would also be the most energy efficient path, and it simplifies the calculation of intersections between the path and obstacles.

It was found that the robot control software was designed to use the first kind of interpolation which was used in the above example, ie the linear interpolation

of the joint axes with time.

For the path finding method described in the following chapter it was decided that the first method would be used for axes T3, T4 and T5 on the robot and the second method was used for axes T1 and T2.

The reasons for adopting method 2 for the upper arm axes were that it simplified the intersection calculations and provided more efficient paths. For the forearm and the end effector axes no simplification of trajectory was found by different interpolation methods so the interpolation method of the robot control computer was used.

5.6 Efficient paths

The efficiency of a path may be determined using many different criteria. Different applications require different emphases to be placed on the different criteria. Five of the most important criteria are listed in section 5.1. For instance, in order to optimise a process, which has a cycle time dependent upon the robot cycle time, a high emphasis must be placed upon a fast robot path. If a process does not depend on the speed of the robot, then a higher emphasis may be placed on minimising the energy consumed.

Some criteria may oppose each other, for instance, as the time taken for a robot path decreases so the energy consumed increases. The optimum path is the compromise between these criteria that produces the best possible path.

The following sections discuss the factors affecting the five criteria of section 5.1 above, and their effect on each other.

5.6.1 The distance travelled by the robot

The distance travelled by a robot may be defined, either as the distance travelled by a point defined somewhere on the robot (usually the hand), or by the total amount of movement which the robot has made.

The total amount of movement which a robot has made is the sum of the movements of each robot axis. If there is a mixture of linear and rotary movements then the rotary movements may be converted to linear ones by defining their linear distances as,

$$Ld = T.L$$

where

T = the rotary movement,

L = the length of the link,

Ld = the linear distance.

Lozano-Perez (1981) and Udupa (1977(a)) used the total distance moved by a robot criterion, to calculate minimum distance paths.

When a robot is programmed to move between two positions there are many different ways it may move. Most robots may move in one or more of three different ways.

(a) Point to point linear interpolation.

A point is defined on the robot, and the robot moves such that the point travels in a straight line from the startpoint to the goalpoint.

Some robots, such as the PUMA may be programmed in so called 'tool space'. In this case a point at the end of the robot forearm moves linearly from one position to another and the hand moves such that its orientation to the xyz coordinates stays the same.

(b) Interpolation of robot axes.

The robot's axes are interpolated such that they all have the same function of time. For instance, if one of the robot's axes has initial value A1 and final value A2 then,

$$A(t) = A1 + f(t)(A2 - A1)$$

and f(t) is the same for all other axes.

For this type of interpolation all points on the robot arm describe complex curves in three dimensional space.

(c) Independent movement of axes.

The robot's axes move independently from their starting positions to their finishing positions. This type of movement requires the minimum of computer control and so in some cases it is the fastest movement possible. However, it is very difficult to model because the different axes reach their final positions at different times.

5.6.2 The time taken by the robot

The time for a robot to move from one position to another depends on the following.

(a) The distance of the path. The greater the distance of a path the longer will be the minimum time taken for that path. The minimum possible time for a path is the time taken at top robot speed. However the speed of the robot's path is in turn affected by the complexity of the path.

(b) The complexity of the path. For a complex path a larger amount of time is spent in accelerating and decelerating the robot arm so the average velocity is reduced.

(c) The type of path. Paths which require large amounts of computing time to calculate, such as point to point linear interpolation, take a longer time to execute than paths calculated by, for instance, the interpolation of axes.

A general method for the planning of minimum time trajectories for robot arms has recently been described by Sahar (1986). Sahar reports that 'optimal paths tend to be nearly straight lines in joint space'

5.6.3 The energy used by the robot

Luh (1985) and Vukobratovic (1982) used the criterion of energy used by the robot motors to optimise the robot path. They found that the factors affecting the energy used were the following.

(a) The time taken by the path. As the robot's path time decreases so the accelerations and decelerations for the robot increase. This then increases the energy used.

(b) The shape of the path. Smooth paths require least energy because the accelerations and decelerations involved are less. Figure 5.3a shows a path of minimum energy consumed for a point moving around a rectangular obstacle. Figure 5.3b shows the minimum distance path.

(c) The distance travelled by a robot. Energy is also dissipated in friction when the robot moves, so the further the robot moves the greater the energy required.

5.6.4 The wear on the robot

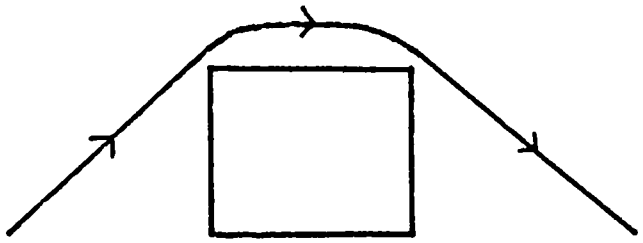


Figure 5.3 Optimum paths - a) minimum energy path

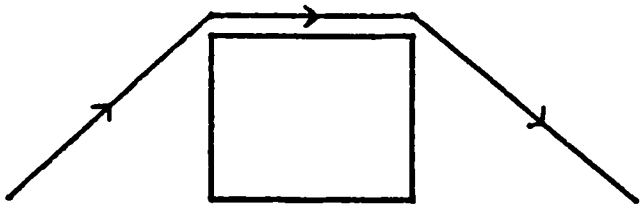


Figure 5.3 Optimum paths - b) minimum distance path

The wear on a robot may be an important factor in determining the cost of a particular path. As wear affects the mean time between failures and the time between services for a robot, then reducing wear will reduce costs of operation and increase productivity.

Wear on robots is affected by the same things that affect the energy used by a path.

5.6.5 The safety of the path

Bonney (1984) described how the safety of a path may be viewed from three different standpoints.

(a) The robot.

A robot may collide with obstacles if it is programmed to move too close to them. It has been found from experience that the path along which a robot is programmed to move may be significantly different to that which it actually takes. One particular problem is the rounding off of corners.

Figure 5.4 shows a programmed robot path consisting of two straight lines meeting at a corner. Most robots will follow a path similar to the dotted line in figure 5.4 which cuts the corner, unless they are programmed to wait for a certain length of time at the corner.

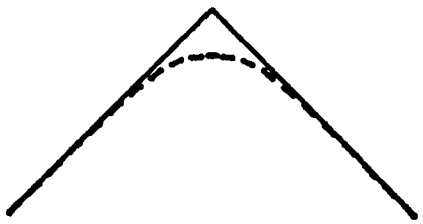


Figure 5.4 Robot paths cut corners

To reduce the danger of a robot hitting obstacles, the nominal size of the obstacles may be increased by some safety margin. This ensures that if a robot does cut corners it will still miss obstacles. However any safety margin may have to be reduced to zero at the startpoints and goalpoints of a path.

(b) The workpiece.

If the robot is moving quickly the forces on the workpiece will increase. This may cause the workpiece to move in the gripper or be dislodged from it.

(c) Humans.

As the speed of the robot increases so the danger to human operators is increased. This means that additional safety precautions may have to be made.

CHAPTER 6

MID-PHASE PLANNING

6.1 Introduction

Mid-phase planning calculates a collision free path between the robot configurations Sm and Gm, where Sm and Gm are configurations with good clearance from obstacles and close to S and G respectively.

The idea of mid-phase planning was first introduced by Udupa (1977). Since then many different methods have been investigated. Mid-phase planning is inherently difficult and all methods tend to require large amounts of computer calculation time in order to produce efficient paths. Schwartz and Sharir (1983) developed an algorithm to solve the find path problem for any particular manipulator. However the algorithms require considerable amounts of computation for simple problems and thus are of no practical use.

Methods may be divided into two categories, local methods and global methods. Local methods start by proposing a path and then going through a process of testing and modifing paths until a collision free path is found to the goal point. Global methods use mathematical graph searching techniques which test large numbers of small movements between many different configurations in an attempt to find a series of movements which will take the robot to its goal.

The method described in this chapter uses a combination of Local and Global planning. It was found that a Global method of planning was suited to the planning of the upper arm movements and that this could be supplemented by a local method to plan the forearm movements.

6.1.1 Local methods

Pieper (1968) was the first to tackle the find path problem of robots and he used a local method. Local methods proceed by moving from one safe configuration to another close by configuration, in a certain direction. Various heuristics are applied to avoid obstacles as they are met. Alternative stratagies may be used if the first strategy is not successful.

6.1.2 Global methods

Global methods solve the pathfind problem in two steps.

(a) The set of configurations where the robot is free from collisions is determined. This is composed of subsets (S1, S2, ... Sr) of configurations. A graph is built from these subsets using an adjacency relation.

(b) Find a path through the graph.

Global methods have the advantage that they do not have to use strategies which may or may not be successful. Thus they are more reliable. But the construction of a graph is a time consuming process.

This method is dealt with more fully in chapters 8 and 9.

6.2 Mid-phase planning for the upper arm

The upper arm was modelled as a line segment fixed at one end to the origin. The obstacles were represented as collections of spheres. The aim of the calculation was to produce the shortest path between configurations Sm and Gm whilst avoiding the obstacles.

The upper arm has only two degrees of freedom, elevation and rotation. This means that the problem is equivalent to a two dimensional one even though the

arm is moving in three dimensions. A projection of the upper arm from the origin gives a point on a background of circles. The problem is transformed into that shown in figure 6.1.

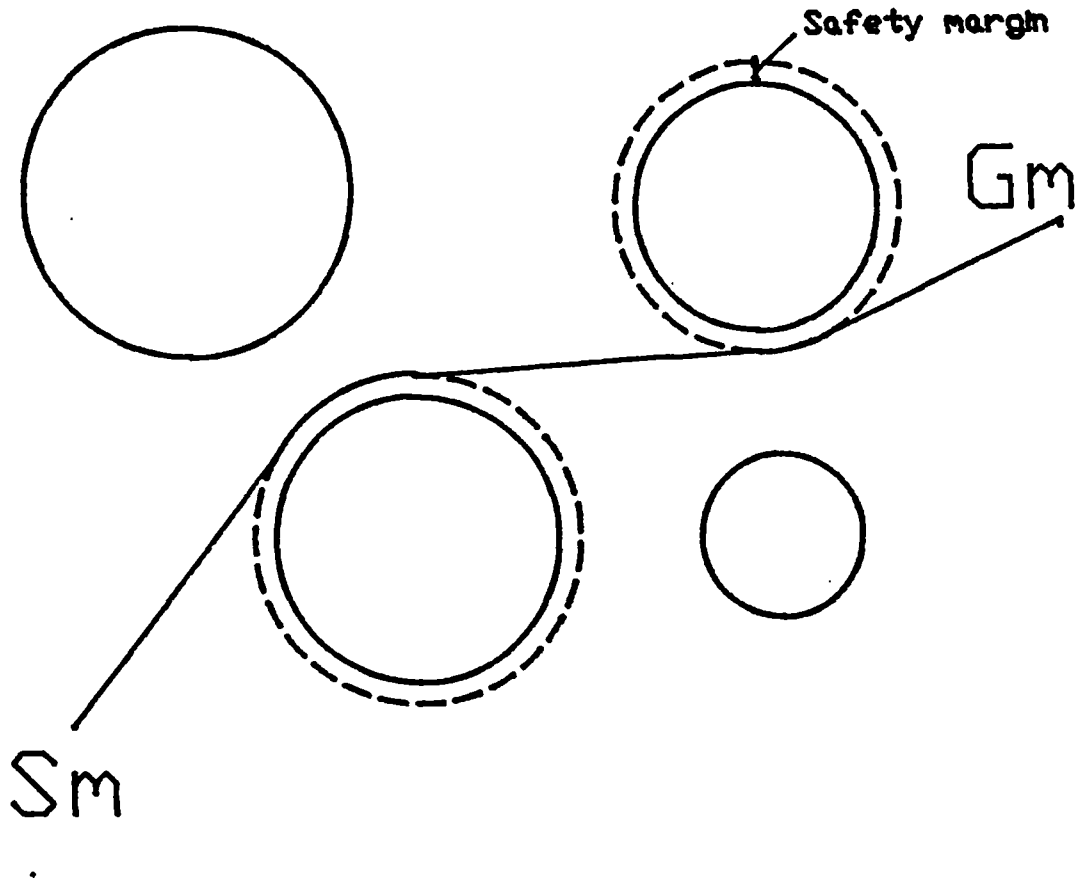


Figure 6.1 Path through circles

The aim of the mid-phase path planning is to produce the shortest collision free path between end points. The term 'shortest path' is defined as the distance a point on the end of the upper arm travels in space. If the path of the point is represented by a vector function r(t) such that

$$r(t) = x(t)i + y(t)j + z(t)k \qquad (a <= t <= b) \tag{6.1}$$

then it can be shown that the path length is given by equation 6.2 (Kreyszig 1980).

$$l = \int_a^b r'.r' \, dt \qquad where \qquad r' = \frac{dr}{dt} \tag{6.2}$$

Factors such as the energy consumed by the robot, the wear, and the time of movement may modify the optimal cost. These may be calculated for specific robots and incorporated in the algorithm if required. It is contended that by calculating the shortest path the other factors are close to their minimum values and so a very good approximation to the optimum path, for any chosen set of criteria, is obtained.

With the cost function equivalent to equation 6.2 it may be seen that the required path between S and G consisted of straight lines between obstacles and circular arcs when traversing the circumferences of obstacles.

One method of finding the shortest path is to calculate all the possible paths of this type. If there are n circles and the path passes around all of them once, then the maximum number of paths is given by equation 6.3.

$$2^n n! \tag{6.3}$$

Thus the maximum possible number of paths m is

$$m = 1 + \sum_{i=1}^{i=n} 2^i i! \binom{n}{i} \tag{6.4}$$

where $\binom{n}{i}$ is the binomial coefficient i of n.

This grows large very quickly, for n=3, m=79; for n=4, m=633; and for n=5, m=6451. In practice the number of possible paths is generally less than this, because some paths between obstacles will be blocked by other obstacles, but even so an unacceptable number of options may remain.

6.3 A heuristic approach

To determine the shortest distance path length a heuristic method of graph searching was found to be quicker than the method of calculating all possible paths. The following method was based on that of Hart (1968).

A graph is defined which consists of a set of elements, called nodes, and a set of paths between nodes called branches. Each branch between nodes has a cost associated with it.

The sub-graph Gn is the set of nodes accessible from a particular node n. The sub-graph is calculated by a successor operator T.

A path from a start node to a goal node is an ordered set of nodes (n1, n2, ... , nk) with each ni+1 a successor of ni. Every path has a cost which is obtained by adding the individual costs of each branch, Ci, i+1, in the path. The optimum path from ni to nj is a path having the smallest cost over the set of all paths from ni to nj.

Starting with the start node S, the subgraph Gs is generated by the successor operator T. During the course of the algorithm, if the subgraph Gn of a node is generated, then the node is said to be expanded. The minimum cost of each node encountered is calculated, and a pointer to the predecessor of each node along that path is stored. The unexpanded node with the minimum cost is always expanded next in the algorithm. Finally, the algorithm is terminated when the goal node G is reached.

6.4 Application to spherical obstacles

A path around a sphere, as seen by an observer, may be in either a clockwise or an anticlockwise direction. Thus for every sphere there are two nodes in the graph.

In the two dimensional case the shortest path consists of straight lines between circles and arcs around the circles. In the real situation this corresponds to the movement of the robot arm in planes between spheres. The robot moves in planes, only changing direction when traversing around spheres. Planes which pass close to spheres are tangential to their surfaces. The robot arm strictly follows the planes without departing to follow the sphere surfaces, direction changes occur at the intersections of consecutive planes.

In order to calculate paths between one sphere and another the generalised procedure Findplan was developed. Each sphere has two nodes associated with it, one for each possible direction of path, clockwise or anticlockwise. When the subgraph of a node is calculated two paths are generated to each sphere. However because of the symmetry of the problem it was decided that both nodes of a sphere could be expanded at the same time without a large increase in calculation time. Thus the procedure Findplan calculates the four possible paths between two spheres.

The mathematics for finding paths between spheres are given in appendix B. The procedure Findplan may be found in the program listing of Mainrpf which is given in appendix C.

Two cost functions were considered. Firstly the cost of a branch was defined as the value of its length. The lowest cost from S to G became the path of the shortest length.

The second cost function was defined as follows. The cost of a branch between node 1 and node 2 is the extra distance the robot has to travel along that branch and from there directly to the goalpoint, compared with a branch which goes directly from node 1 to the goal point. To calculate this cost the following equation was used:

Cf = distance along branch from node 1 to node 2
+ direct distance from node 2 to the goal point
- direct distance from node 1 to the goal point. (6.5)

A comparison of the two methods is shown in figure 6.2. For ease of representation figures 6.2(a) and 6.2(b) show graphs of independent nodes. The double nodes of circles are not considered. Figure 6.2(a) shows a graph of the first cost function. In this case all the branches were searched. In this example the method took ten steps. Figure 6.2(b) shows that the second cost function calculation caused the search to be completed after only five steps. The second cost function was preferred, as the time saved by computing fewer steps was greater than the extra time taken to calculate the more complicated cost function.

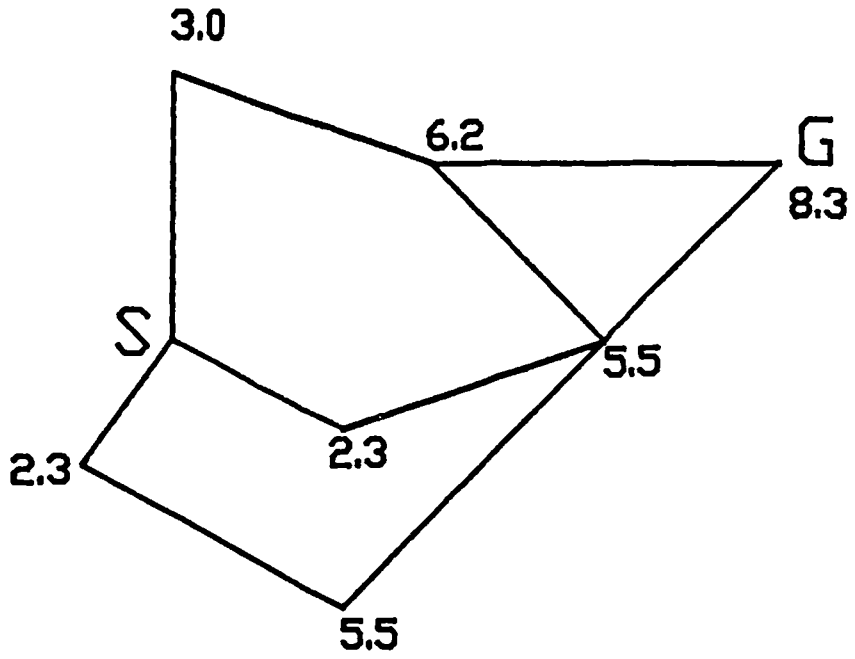


Figure 6.2(a) First search strategy

The successor operator T generates subgraphs of a node n as follows. Paths to all the other nodes are proposed. The node which represents the same circle as n but the opposite direction is rejected. Of the candidate paths those which leave the circle with angles A of less than 180 degrees are rejected because the optimum path must be tangential (see figure 6.3). The remaining paths are then checked for intersections with other circles, those that are clear, form the subgraph Gn.

6.5 Forearm path planning

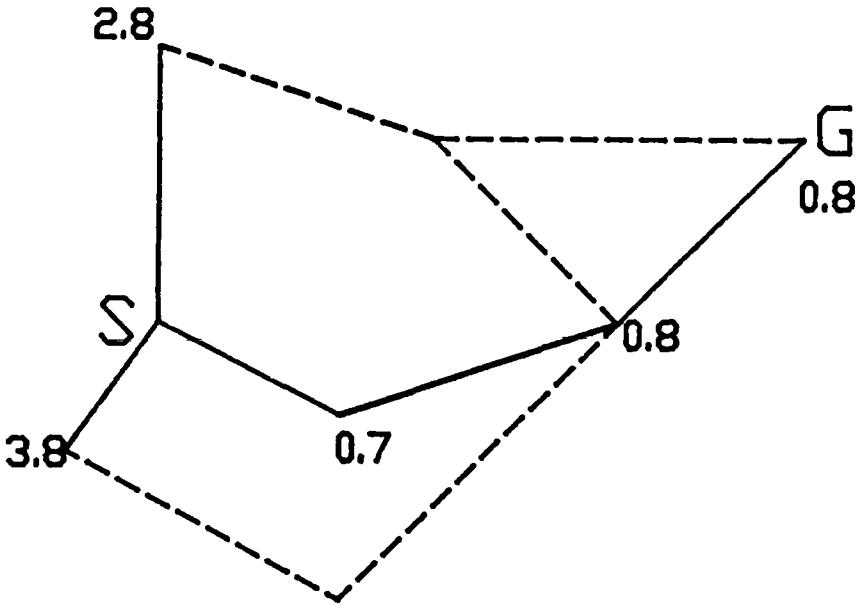


Figure 6.2(b) Second search strategy

Having fixed the trajectory of the upper arm the locus of the elbow was established, and thus the problem of finding a path for the forearm became a 2D problem instead of a 3D problem.

The forearm path is constrained by the locus of the elbow. A graph of distance along the locus of the elbow can be plotted against the elbow angle. Viewed from the locus of the elbow, a graph of the two dimensional working space of the forearm can be calculated showing the positions of obstacles. The pathfinding task is now to search this graph. However the shapes of the obstacles are not circles as they were in the case of the upper arm, but complex shapes due to the transformation into the new two dimensional space. These shapes have to be calculated for each upper arm path. An example of these shapes is shown in figure 6.4.

While the transformation gave the true location of the obstacles in the two dimensional space, for the purposes of this investigation a further simplification was adopted to define the forearm path.

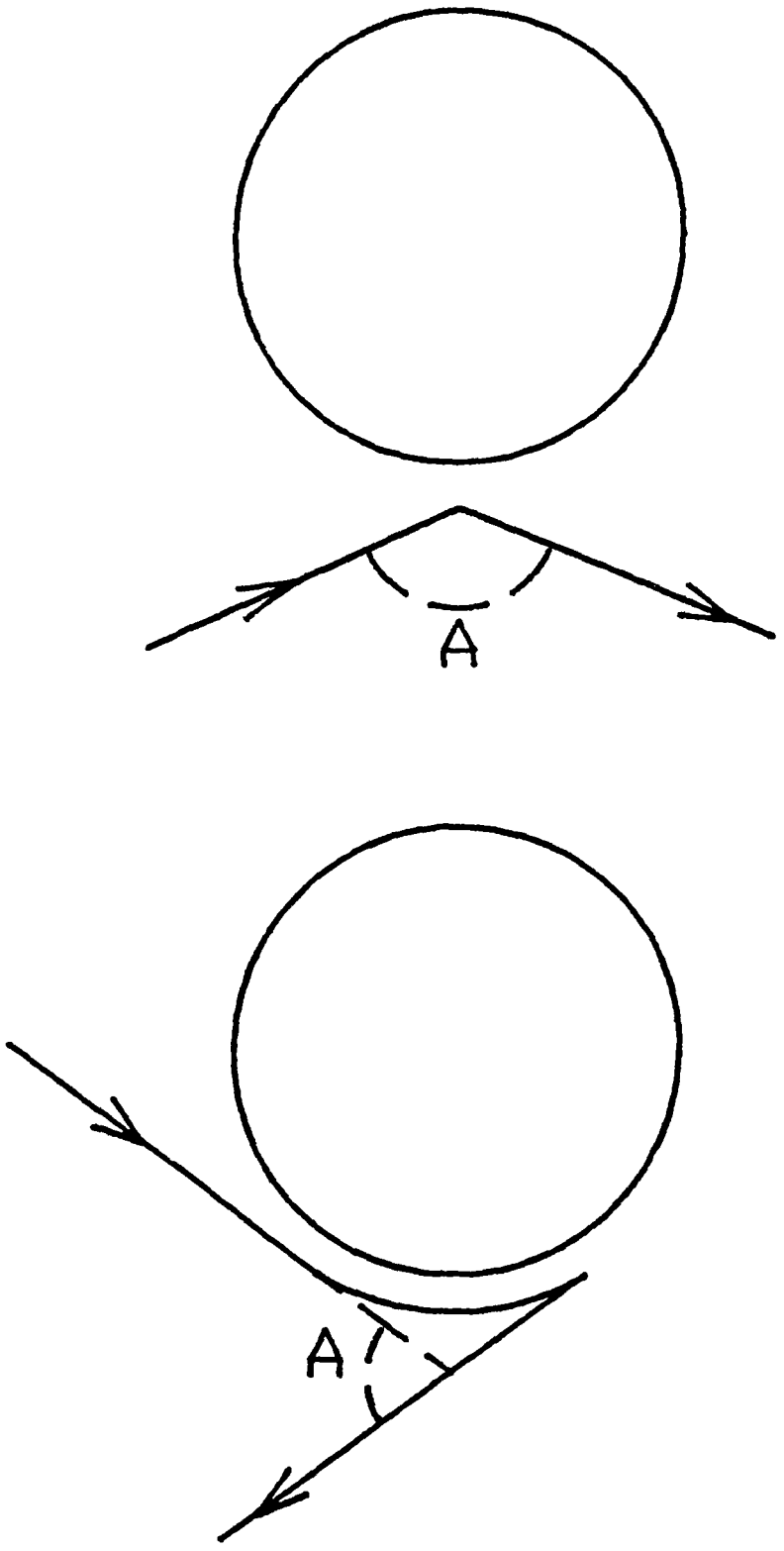


Figure 6.3 Paths which can be neglected

To adapt the method used for upper arm planning to the forearm problem, an equivalent to the 'straight branch' path had to be defined. In the two dimensional

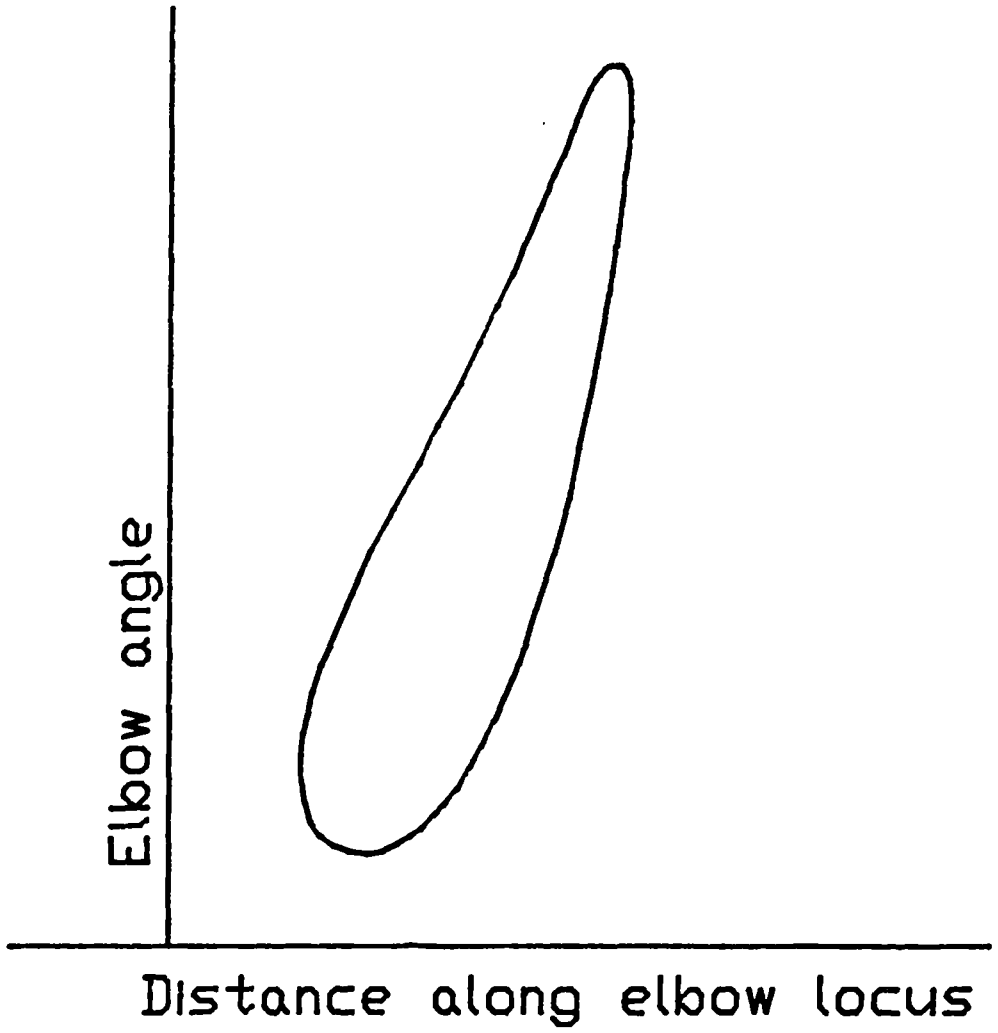


Figure 6.4 Obstacle representation in transformed reference frame for forearm path planning

transformed space of the forearm a 'straight' line was defined as:

$$e = k.d + c \tag{6.6}$$

where

e = elbow angle.

d = position on the elbow locus.

k , c = constants.

This defined a path that would be represented by a straight line in figure 6.4.

The planning strategy chosen was as follows. Test a straight forearm path between Sm and Gm for collisions and keep a note of the first collision which would occur. Propose a path which avoids the first collision, test this path and repeat until the first stage of the path is clear. Try again to reach the goal and repeat as before until Gm is reached.

6.6 Planning of the gripper and workpiece

The gripper was modelled as a series of spheres. While moving in mid-phase motion the gripper was aligned with the forearm. If the gripper is small compared with the forearm, this avoids the need to calculate the extra degrees of freedom for the gripper itself. The position, and representation of the workpiece in the gripper is known from the approach path planning.

6.7 Avoiding obstacles of the forearm, gripper and workpiece

The path planning for the forearm, the gripper and the workpiece is done together. The paths of the forearm and the spheres representing the gripper and the workpiece are calculated, and the closest point of the forearm, gripper or workpiece to the first obstacle is calculated. A path which avoids the obstacle is proposed as follows. The vector between the closest point on the robot and the centre of first obstacle is calculated. This vector is then extended so that the robot and the obstacle are a set distance apart.

CHAPTER 7

IMPLEMENTATION

7.1 Introduction

The computer programs developed to implement the path planning method described in chapters 4 to 6 were written in Pascal and run on an Intel 8086 based micro computer. Sections 7.2 to 7.5 of this chapter describe the algorithms in general terms and the important mathematical techniques behind certain procedures are described in detail in the appendices. Full listings of the programs may also be found in the appendices.

Figure 7.1 is a flowchart of the main program called Mainrpf (Appendix C). The main parts of the path planning program were :

(a) To define the problem in terms of a world model and a task description.
(b) To solve the problem which results in a robot trajectory.
(c) To output the data to the robot.

Data defining the world model is stored on disk and is read by Mainrpf when required. The world models are developed and stored on disk by a separate program called Storedta (Appendix D). This ensures that the data for a particular model only has to be entered once. Storedta is also used to edit the data so that minor changes in a world model can be made easily.

The world model data is transformed into two different spaces, one for the upper arm search and one for the forearm search. This is necessary in order to deal with the lateral property of the robots. (See 4.5 for details of the transformation).

Having retrieved the data for the world model, the program then commits itself to reading in successive task descriptions, solving the find path problems

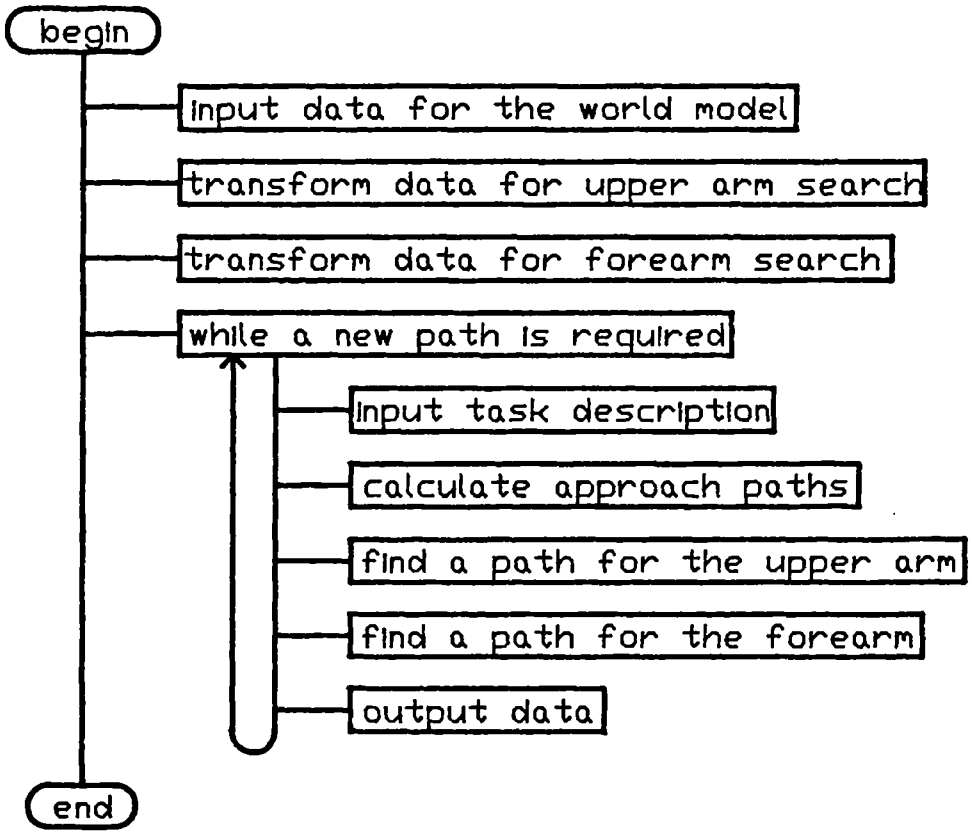


Figure 7.1 Flowchart of Mainrpf

and outputting instructions to the robot. Hence the loop on the flowchart in figure 7.1. The program is only stopped if either the world model has changed, or because it is necessary to switch the computer off.

The task description is input by the operator, this consists of the initial and final coordinates of the centre of a part. From the task description and the world model data, the program is able to define the approach paths and specify the path planning problem in terms of start and goal configurations.

The path trajectory is calculated in two parts, the upper arm path (section 7.4) and the forearm path (section 7.5). The trajectory is converted into the robot coordinates suitable for the robot control computer and is down loaded to the robot control computer (section 7.6).

7.2 Storedta

An important part of the path planning system is the program Storedta. This program enables the operator to build up a model of the robot surroundings by specifying the positions and sizes of spheres. Appendix D gives the program listing of Storedta, and figure 7.2 shows a flowchart of the program.

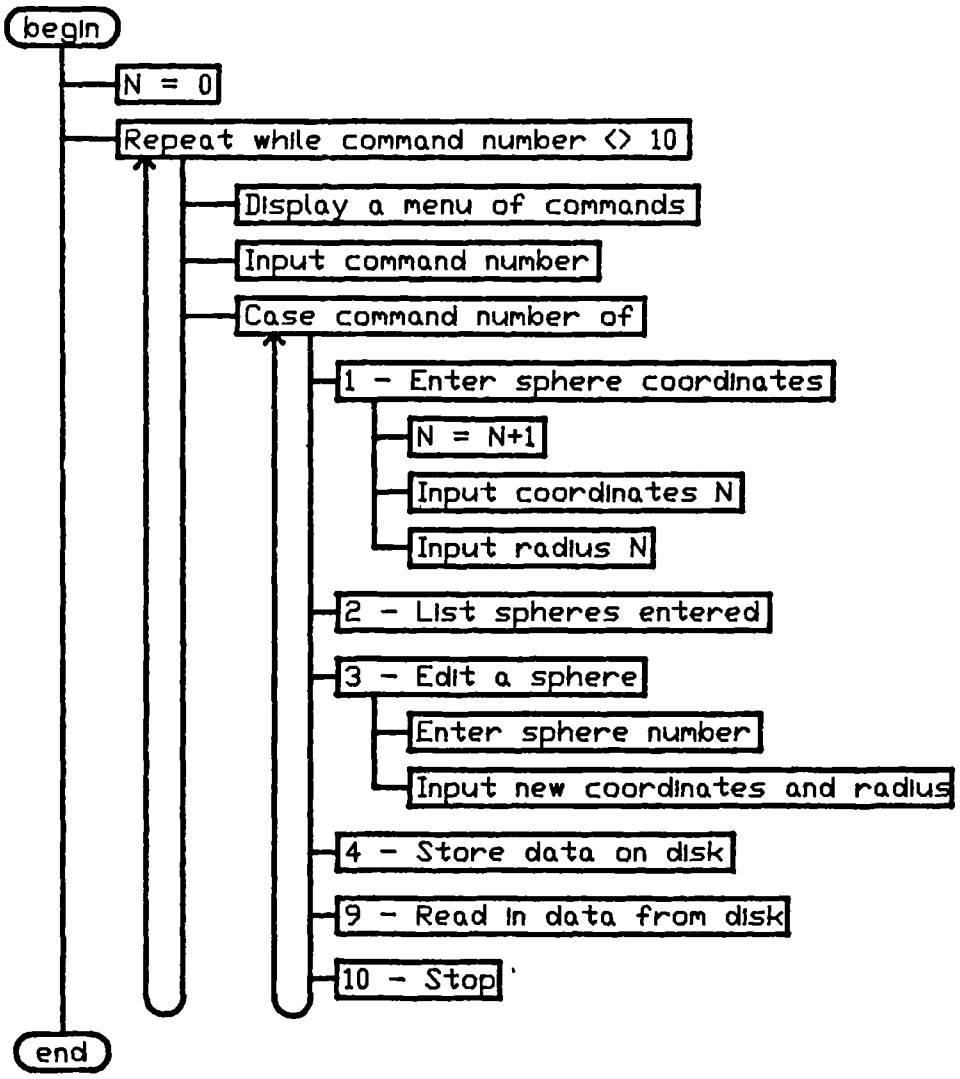


Figure 7.2 Flowchart of Storedta

Storedta is a very simple menu driven program which enables the operator to create and edit different sphere models. The commands available to the operator

are, Enter sphere coordinates, List sphere coordinates, Edit sphere coordinates, Store model on disk, Load model and Stop.

7.3 Task Description

Very simple task descriptions only are permitted by the software developed here. The program allows the operator to input the initial and final coordinate positions for the centre of a part.

The approach path to pick a part up is defined as follows.

(a) Move the robot to a position 100mm above the part.

(b) Move down slowly and in a straight line until the robot is in a position to grip the part.

(c) Grip the part.

(d) Raise the part slowly by 100mm.

The approach path to put a part down is the same as above but in reverse order.

The approach paths used are defined by only a few lines of program code. If any new approach path is required it is easy to provide this by adding to the program code. Similarly different task descriptions can be accommodated easily by small changes in the software.

7.4 Upper arm path planning

Figure 7.3 shows the data structure for the graph used for the upper arm path planning. The start node is designated as sphere 0, nodes 1 to N represent the spheres 1 to N in the world model and sphere N+1 represents the goal node.

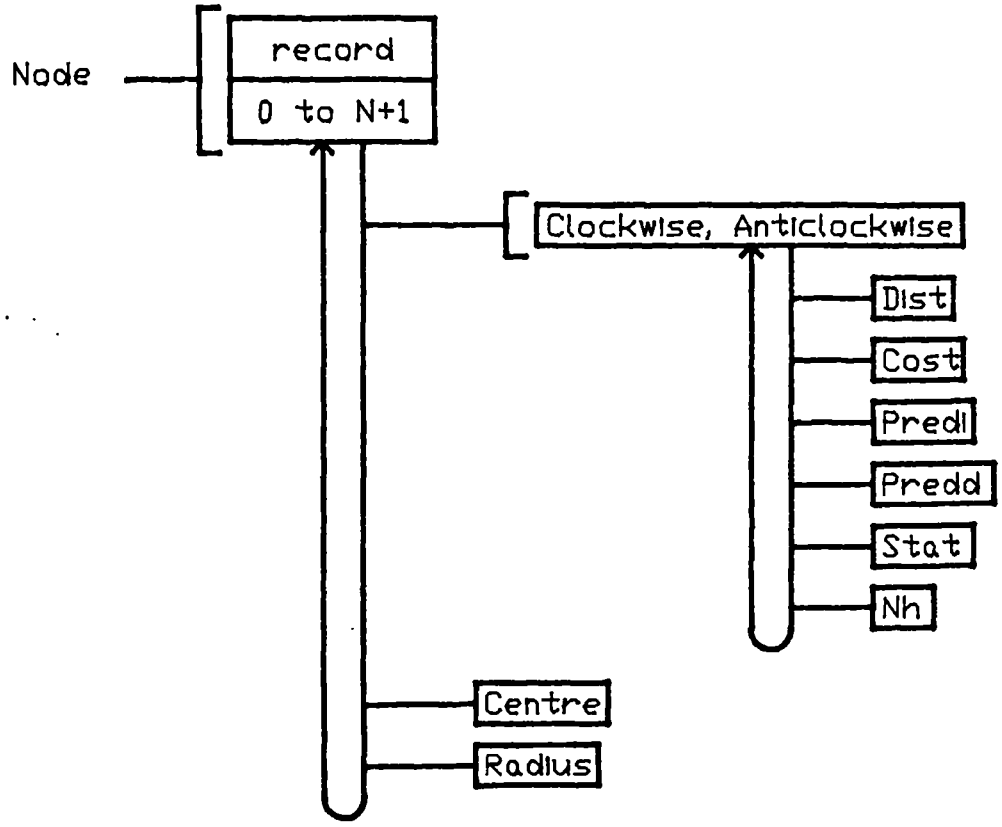


Figure 7.3 Data structure for the upper arm graph

Before the path planning is started some of the variables in the data structure are given initial values. The start node status is set to 'open', all other node statuses are set to 'closed'. The start node is initially counted as the opennode. The start node has a radius of 0 and coordinates equal to the coordinates of the end of the upper arm in the starting configuration. The path cost function is set to 0 for the start node and a very high number (9999) for all other nodes.

Each sphere is represented by two nodes, one for a clockwise path direction and one an anticlockwise path direction. Thus between any two spheres there are four possible paths. The paths between any two spheres are calculated in one pass of the appropriate routines. The reasons for this are that:

(a) The extra calculations required to find and test all the paths between any two spheres are not great in comparison to the calculations required to find and test one particular path.

(b) If a node is being expanded because of its low cost then it is likely that the next node to be expanded is its opposite node on the sphere.

Figure 7.4 shows a flowchart of Routep which is the procedure used to plan a collision free path for the upper arm.

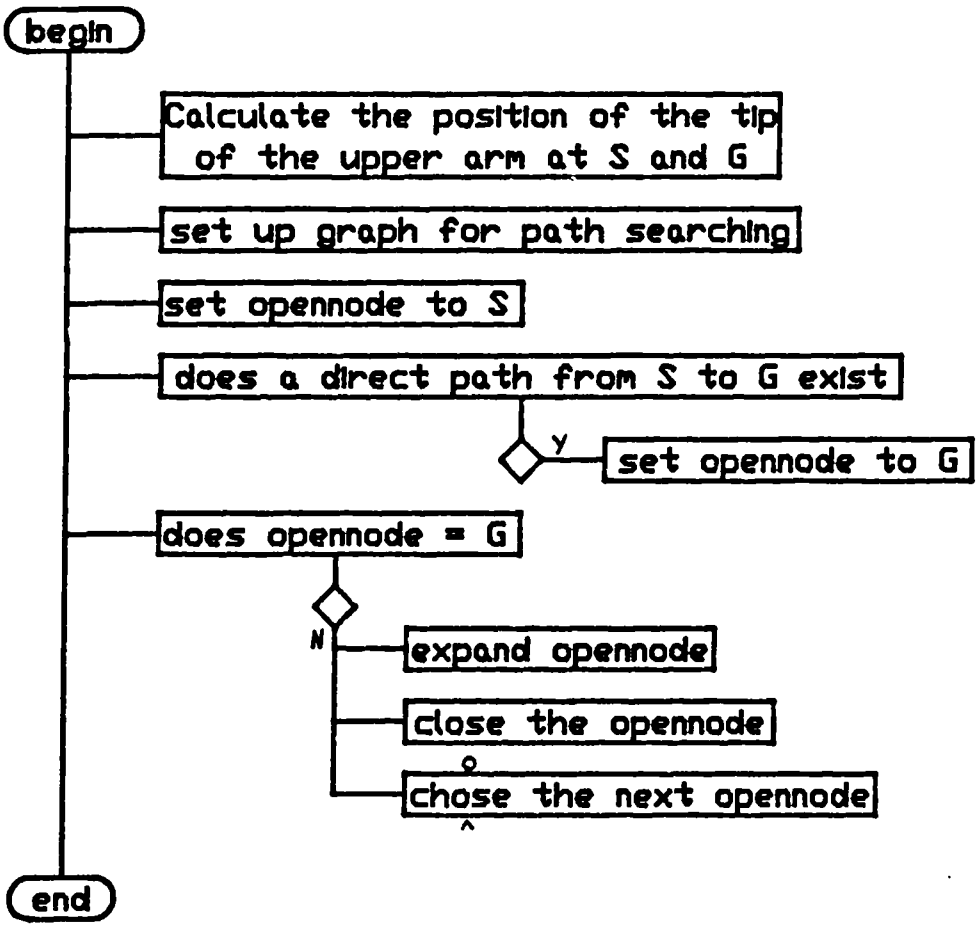


Figure 7.4 Flowchart of RouteP

After the graph has been set up for searching, the first test is to find out whether a clear path exists directly from S to G. This test is incorporated in order to save computer time for simple solutions.

The algorithm continues by expanding nodes and choosing new opennodes until either the goal point is reached, or all nodes have been expanded. The theory behind this method is described in sections 6.3 and 6.4. If all nodes are expanded and no path to the goalnode is found then it is assumed that no path is possible and the program is stopped.

Once a node has been expanded its status is set to 'closed' so that it is not chosen as an opennode again. To choose the next opennode the node which has 'open' status and has the lowest cost path from S is chosen.

Figure 7.5 shows a flowchart of expand. The first node to be expanded is the start point node. From this node paths are proposed to all other nodes. Each path is tested for collisions with obstacles. Provided the path is clear, the cost of the path to this new node is determined. If the cost of this new path to the new node is less than any previous path then the following information is stored for the new node.

(a) The new cost to the node, which is equal to the cost to the opennode plus the cost from the opennode to the new node.
(b) The successor of the new node, which is the opennode.
(c) The status of the new node is set to 'open'.

7.5 Forearm path planning

7.5.1 Data Input

From the upper arm path planning a series of configurations of the upper arm are produced. Between these configurations the upper arm moves in planes. The forearm path planning algorithm (Fapath) has to adjust the position of the forearm, by varying the elbow angle to avoid obstacles. An initial configuration S and a final configuration G are known. In between these configurations there

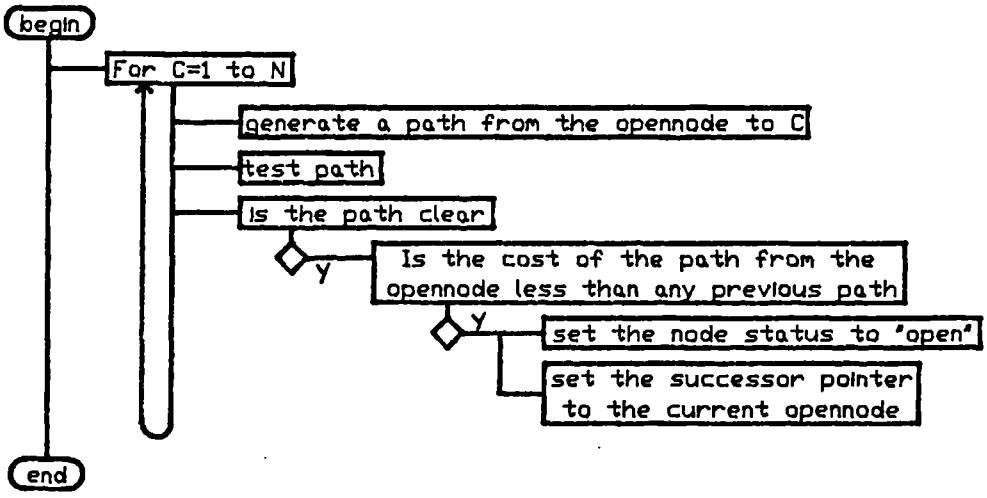


Figure 7.5 Flowchart of Expand

are certain intermediate configurations where the position of the upper arm is known but that of the forearm is undefined.

7.5.2 Data output

The data required at the end of the forearm path planning routine is a series of robot configurations. The path is then defined by the movement of the robot directly between these configurations.

7.5.3 Fapath

Figure 7.6 shows a flowchart of the forearm path planning procedure, called Fapath. As the path is calculated the successive configurations found are stored in an array called Robcoor[CountRC] where CountRC denotes the position in the list.

From the input data a series of configurations are defined for the upper arm. The planning of the forearm is carried out in stages between these configurations. For instance, if four configurations of the upper arm are defined, including the start and goal configurations, then the path planning for the forearm takes place

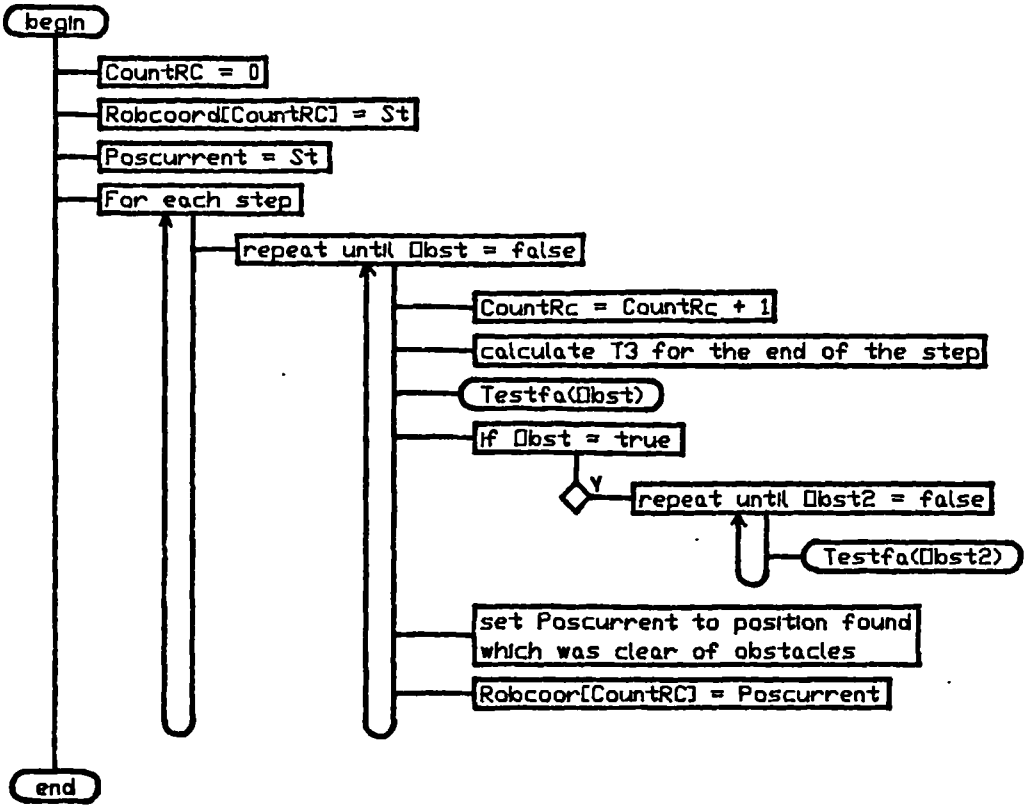


Figure 7.6 Flowchart of Fapath

in three stages, between 1 and 2, between 2 and 3, and between 3 and 4. The stage number is held by the variable Stepnum in the flowchart.

The elbow angle T3 is known at the start of the first stage. The procedure Stepgoal is used to propose a value for the elbow angle for the end of the stage. Stepgoal calculates the proportion of the path, P1, for the upper arm between Poscurrent and the Goal configuration. It calculates the total elbow angle difference between Poscurrent and the Goal configuration. The proposed elbow angle for the end of the stage is equal to the elbow angle at the start of the stage plus the proportion of the difference in elbow angle between Poscurrent and the goal configuration.

$$T3_{prop} = T3_{Poscurrent} + P1(T3_{goal} - T3_{Poscurrent}) \qquad (7.1)$$

The path between the current configuration and that at the end of the stage is tested by the procedure Testfa. If the path is blocked then an alternative configuration is proposed such that the path between Poscurrent and this position should be collision free.

If no collision is detected then T3goal becomes Poscurrent and Poscurrent is recorded on the list of configurations Robcoor. That stage of the path is then complete and so the next stage of the path is investigated.

If a collision is detected then the path from Poscurrent to the alternative configuration is tested. If the path is collision free then Poscurrent becomes the alternative position and is recorded on the list Robcoor, and the path to the end of the stage is investigated. If the path is not collision free then the next alternative position is investigated.

7.5.4. Procedure Testfa

Figure 7.7 shows a flowchart of Testfa. Firstly the flags Obst and Obst2 are set to false. Each sphere is considered in turn for possible collisions. The position and size of each sphere is compared with the range of configurations for the robot arm. If a collision is possible then the path is tested by procedure Testfa2 (figure 7.8). If a collision occurs then the flag Obst2 is set to true. If more than one collision is detected then the collision closest the starting configuration, Poscurrent, is recorded.

If the proposed path is obstructed then procedure Avoidobs is used to propose a new path.

7.5.5. Procedure Testfa2

For the range of configurations through which the robot moves the sub-range in which the robot can hit the obstacle sphere is determined. The configurations at either end of this sub-range are calculated. The closest point on the robot,

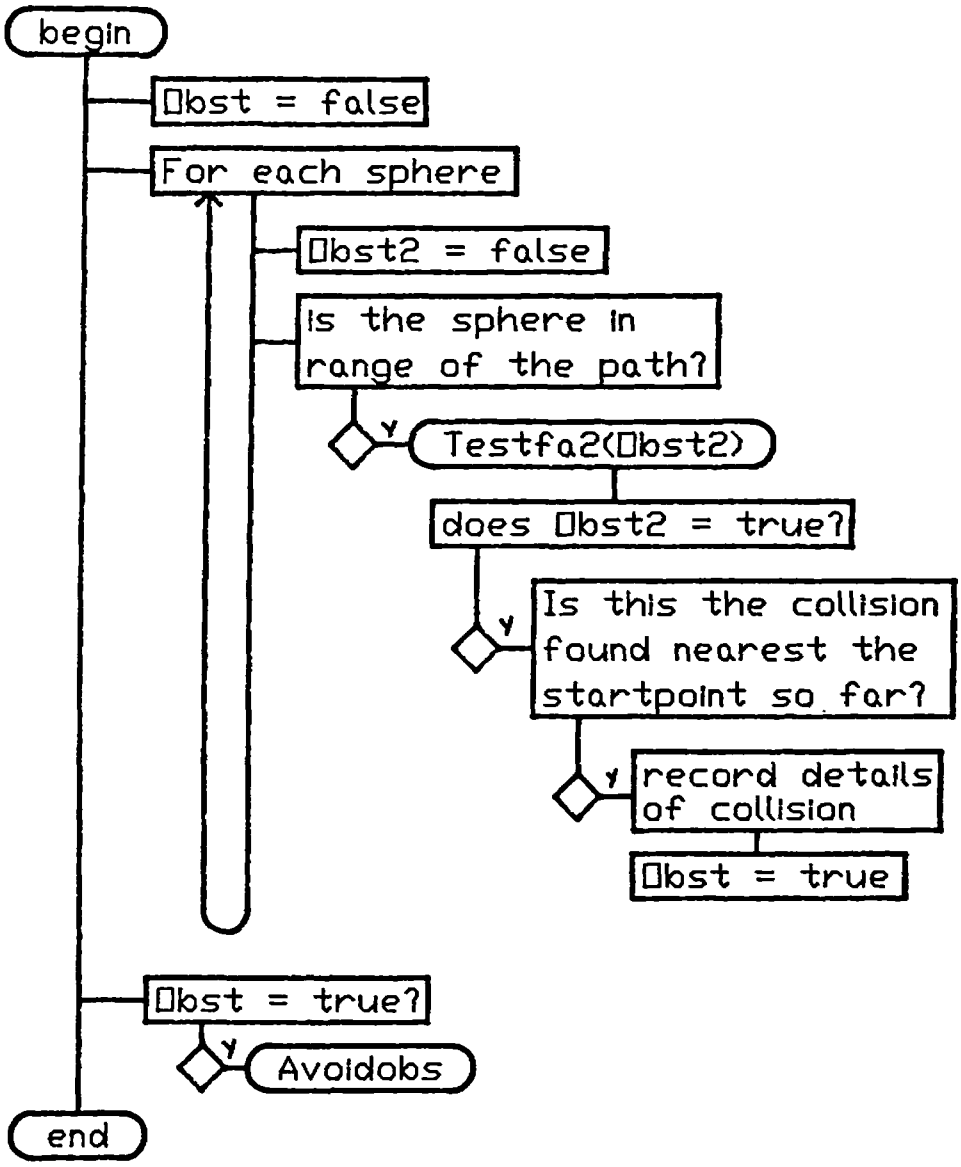


Figure 7.7 Flowchart of Testfa

to the obstacle, is found for each configuration. The closest point on the line between these points is then found.

This is the closest point on the robot's forearm to the centre of the obstacle sphere. This point is shown in figure 7.9.

This method is then repeated for parts on the robot forearm such as the gripper and the part the robot has gripped. The closest point on either the

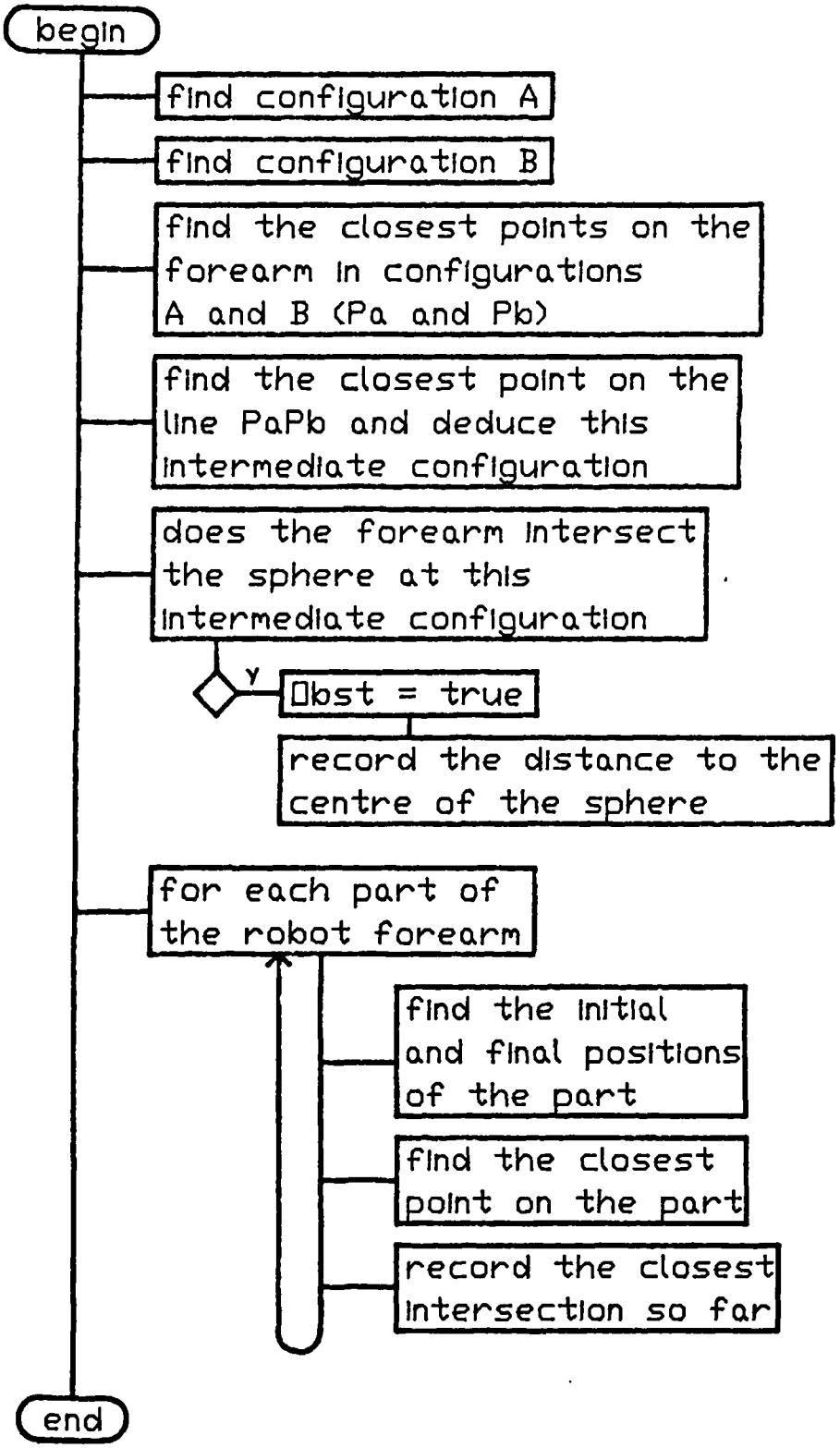


Figure 7.8 Flowchart of Testfa2

forearm or the parts of the forearm is found.

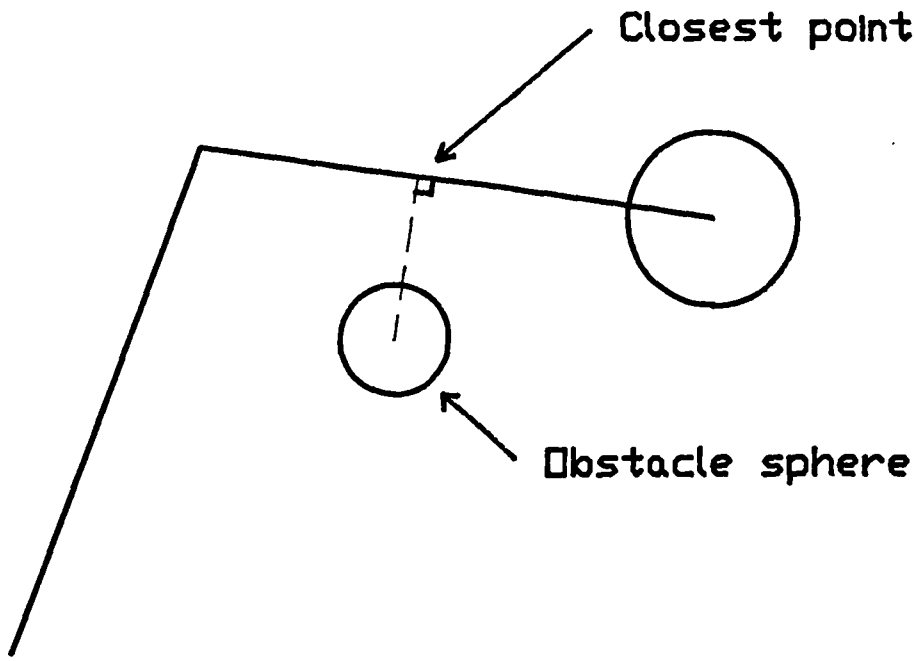


Figure 7.9 The closest point on the robot forearm

7.5.6 Procedure Avoidobs

Figure 7.10 shows a flowchart of the procedure which calculates a new path for the robot which avoids colliding with an obstacle.

The closest point on the robot arm to the obstacle is known from procedure Testfa2. The vector between the centre point of the sphere and the closest point is enlarged until the robot is a safe distance away from the obstacle.

7.6 Data transfer to the robot control computer

After the path planning program has finished, the robot trajectory data is in the form of a series of robot configurations, defined by the values of the robot joint angles. This is called the configuration in 'joint space'.

The robot control computer also operates in joint space. However the coordinate systems of the robot control computer and the path planning computer

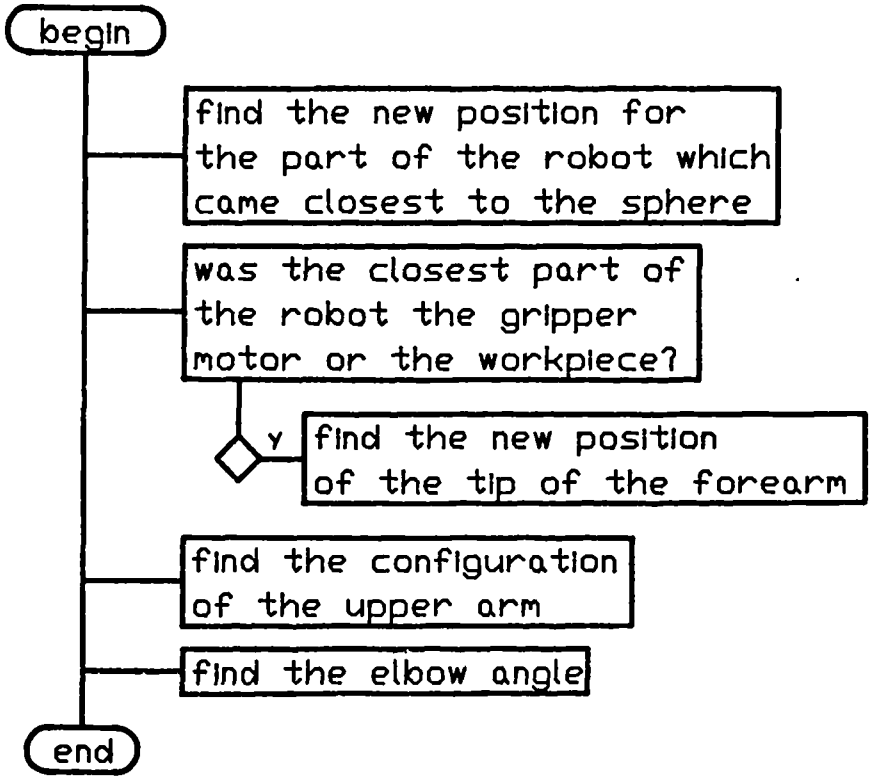


Figure 7.10 Flowchart of Avoidobs

are different. For the robot control computer the total range of each axis of the robot is divided into 1000 positions (0-999). The path planning computer simply uses the values of the joint angles in radians. Thus a coordinate transformation was carried out to determine the robot control computer coordinates. The transformation for each axis is of the form of equation 7.2.

$$Rc = C1.T + C2 \tag{7.2}$$

where

Rc = robot control computer coordinate (rounded to the nearest integer).

T = path planning computer coordinate.

C1, C2 = constants.

The constants C1 and C2 were found by measuring the joint angles at the robot control computer coordinates of 0 and 999 and substituting these values into equation 7.2.

It was found that the robot control computer does not always interpolate between coordinate positions in a predictable manner. Also the interpolation required for the upper arm is interpolation in planes and this is different from the joint angle interpolation of the robot control computer. To solve these problems many intermediate configurations are generated by the path planning computer and these are translated into RCC coordinates. This means that the robot moves only small distances between defined configurations and hence the difference between the two types of interpolation is negligible.

7.7 System performance

The phrase 'Real-time' has several interpretations, in this thesis it is used in the context of collision avoidance. Real-time collision avoidance means that the solution of the path planning problem takes no longer than the robot takes to execute the path. When this is the case the path planning computer may be provided with a sequence of problems, and it will down load the solutions to the robot computer in time to ensure that the robot is kept busy all the time.

Unfortunately the time of execution of a robot trajectory is not necessarily proportional to the time the trajectory takes to plan. The calculation time is dependent upon the following.

(a) The number of obstacles in the model.

(b) The number of obstacles in the space between the start and goal configurations.

(c) The arrangement of the obstacles.

(d) The size of the obstacles.

In certain circumstances the calculation time required to solve the path planning problem can not be predicted. However upper limits can be placed on the calculation time to ensure that the program does not go into an infinite loop, but if the upper limit of the calculation time is reached then no path is found.

In practice it was found that many environments can be used such that solutions are always found to the path planning problem and that the calculation time is within the limits of the robot execution time.

An example of an environment for which the system has performed satisfactorally is shown in figure 4.7. In this example the robot moved 5 parts at random to different places such as areas on the base and to the tops of the boxes. For a typical task in this environment, such as pick up a part and move it to a different position the robot trajectory took 30 seconds and the calculation time was 25 seconds.

The path planning method performs differently for different hardware. For testing the algorithms the robot used was particularly slow in movement as it was necessary to ensure that the robot followed the planned trajectory precisely. However the computer used was also slow in comparison with the latest micro computers.

7.8 Discussion

The calculation time for the example task of the previous section was 25 seconds, this compared with a programming time of 12 minutes for an experienced programmer to get the robot to do the same task using a teach pendant. Although the world model took some time to develop (dependent upon complexity), this only had to be entered once. So automatic programming may be used to save programming time in the future.

The task descriptions available to the operator are limited to specifying the centre coordinates of parts in their starting and goal positions. An example of a task description at a higher level might be 'carry out operation X on part C'. Here no coordinates need specifying, this is a higher level of control and this may be implemented in the future.

The approach paths available to the operator are also limited, although any particular approach path can be quickly implemented by adding to the software. Again this is a higher level of control which may be implemented in the future and is a topic for further work.

In certain environments the path planning computer always produces satisfactory paths in 'real time'. However there are certain situations where this method becomes 'trapped' and no path is found where a path exists or that paths are found after calculation times greater than robot execution times. Given the unlimited complexity of paths there will always be these problems no matter what path planning algorithms are used. However improvements were and still can be made to the basic path planning method.

One problem that occurred early on in the testing was that when objects were represented by more than one sphere the path planning algorithm tried to find paths for the forearm between spheres of the same object. This is shown in figure 7.11.

This problem was overcome by adding extra heuristics into the trajectory planning algorithm. When avoiding a sphere the algorithm checked for this situation and if it occurred the algorithm planned a path above the highest sphere.

The performance of the automatic programming system can be improved by the following.

(a) A different design of robot. This would simplify the algorithms and thus improve performance.

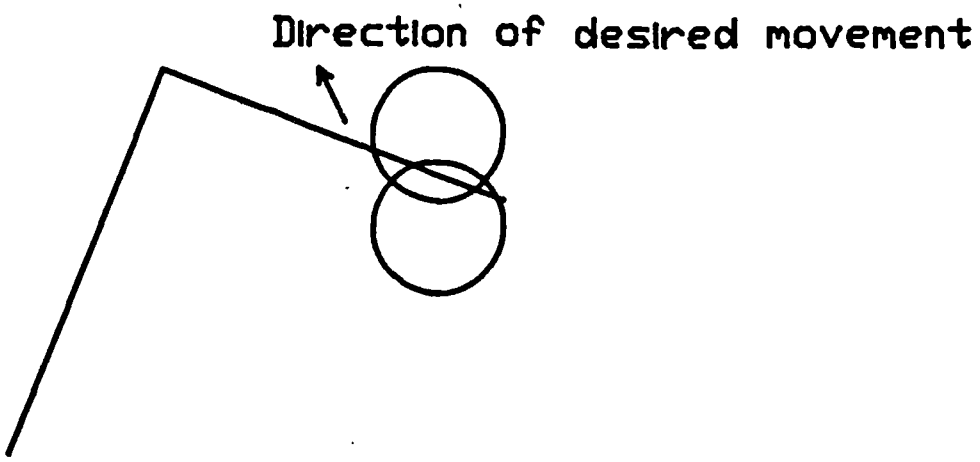


Figure 7.11 Paths between spheres

(b) A more modern computer. This would increase the speed of running of the path planning program.

(c) Improvements in the software. It may be noted that relatively simple changes in software made improvements of 10 or 20 percent in the performance of the algorithm.

The advantage of this method is that it uses simple rules to solve a problem which is difficult to analyse. The method used for the upper arm is efficient in terms of time and paths produced. The method for the forearm uses simple heuristic rules to avoid obstacles.

The disadvantages of the method are that it does not always find a path where one exists. The fact that the forearm and upper arm are planned separately means that many possible paths are not considered and hence the paths produced are not necessarily the shortest paths. Also the program is closely tied to one type of robot. Some of the program code would need changing to accommodate the kinematic chain of a different robot.

The performance of the system is encouraging in that robots can now re-calculate their trajectories with minimal delay. The performance is difficult to

quantify. Computer programs are often compared by carrying out Bench Mark tests. However there is no other automatic programming system or data available for this comparison to be made.

$$E2x = E1x - 54.sinT1$$

$$E2y = E1y + 54.cosT1$$

$$E2z = E1z \tag{8.4}$$

$$P = T1 + T2 - Pi \tag{8.5}$$

$$T1x = E2x + 376.cosT2.cosP$$

$$T1y = E2y + 376.sinT2.cosP$$

$$T1z = E2z + 376.sinP \tag{8.6}$$

$$T2x = T1x - 22.sinT1$$

$$T2y = T1y + 22.cosT1$$

$$T2z = T1z \tag{8.7}$$

More general details of robot coordinate transforms may be found in Paul (1981)

8.4.2 To transform obstacles one at a time

The most straight forward method for transforming obstacles into joint space is to check each unit of the joint space graph for intersections with each obstacle. However, this method uses large amounts of computing time. The actual amount of time is proportional to the number of obstacles and the number of units.

A faster method is to consider each obstacle at a time and test all the units which could possibly contain the transformed obstacle. The following is the basic method by which this can be achieved.

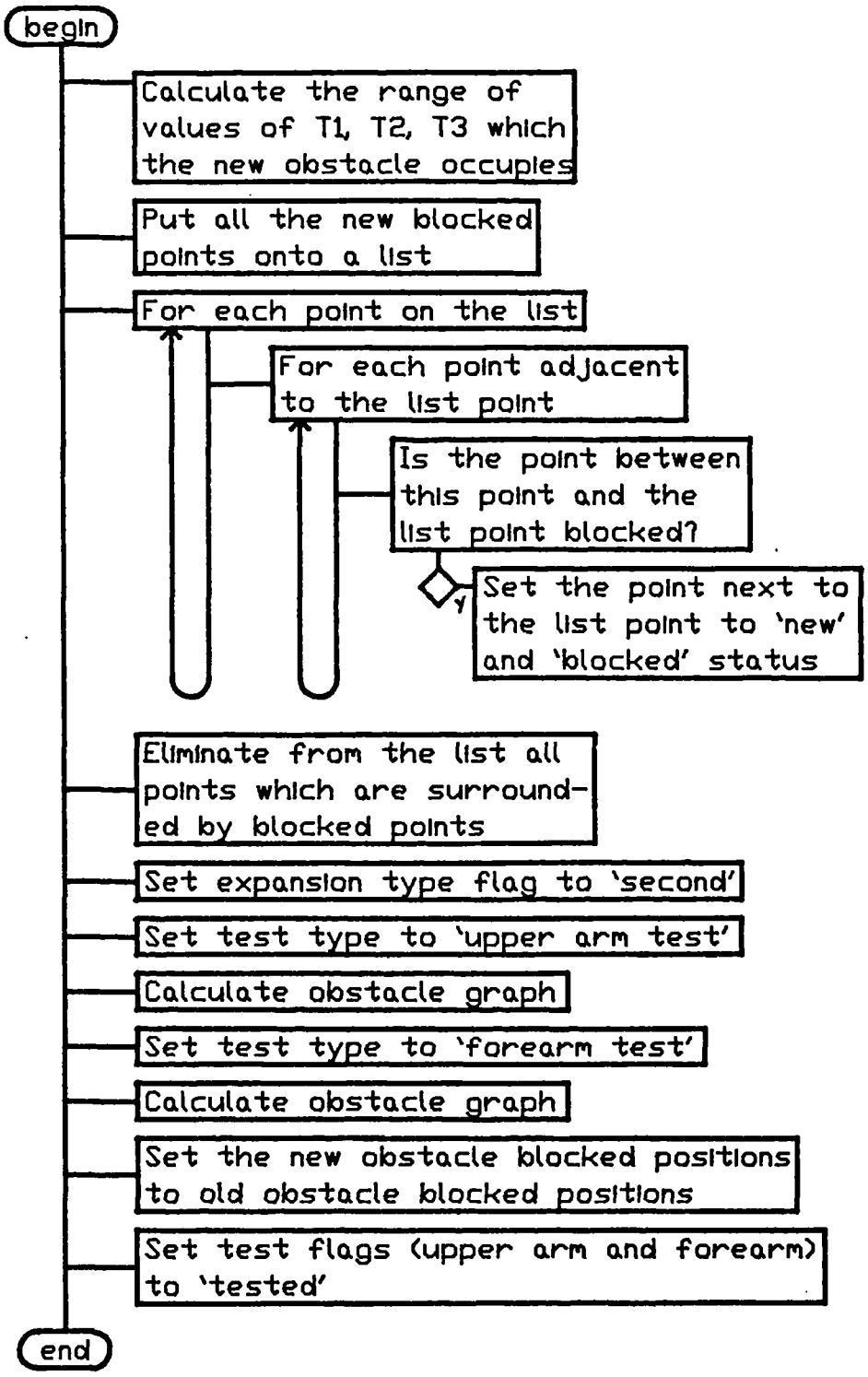


Figure 8.4 Flowchart of Fill

8.6 Results of obstacle transformation

The obstacle transformation program was tested for sizes and types of obstacles. The most important consideration for the program was that it should be

The configuration of a robot expressed as a function of time specifies the robot trajectory. A trajectory locus is the curve the robot configuration traces in joint variable space. The trajectory planning problem is to find a trajectory locus that will take the robot from the start to the goal configuration subject to any given constraints.

Trajectory calculation deals with computing a trajectory from a trajectory locus. The executive system responsible for powering the movement of the physical robot is called the robot control computer. This uses the trajectory locus to calculate and perform the robot trajectory.

8.3 Space Transformation

Obstacles are conveniently described in cartesian space, and robot trajectories are best represented in joint-variable space. The complexity of the collision detection and avoidance problem is partly due to having these two diverse representations. If obstacles and trajectories could both be represented in one space, the overhead of conversion between the two spaces would be eliminated.

The conversion of obstacles into robot joint space is not a simple problem. For example a point obstacle in cartesian space is not transformed into a point in joint space. If the point is outside the robot workspace then it is not represented at all in joint space. If it is inside the robot workspace it is transformed into one or more complex three dimensional shapes.

Representing complex shapes on computers may be done by approximating the shapes by mathematical curves, geometric shapes or units of space. The method adopted here was to represent the obstacles as regions of space consisting of small units. Each unit represented a range of configurations for the robot, in terms of a central configuration, (T1a, T2a, T3a) for example, plus a degree of

movement away from these values ($\pm dT1, \pm dT2, \pm dT3$). Thus all units together represented the whole robot workspace.

The number of units in the graph, Ng, was given by :-

$$Ng = \frac{T1u - T1l}{2.dT1} \times \frac{T2u - T2l}{2.dT2} \times \frac{T3u - T3l}{2.dT3} \tag{8.1}$$

where

T1u, T1l = the upper and lower limits of T1.

T2u, T2l = the upper and lower limits of T2.

T3u, T3l = the upper and lower limits of T3.

If at any configuration in a unit, the robot intersected an obstacle, then the unit was said to be blocked. If at all configurations in a unit the robot did not intersect an obstacle then the unit was said to be clear. Before a unit could be declared blocked or clear two things were done. Firstly the position problem was solved for the configuration at the centre of the unit. Secondly the maximum distance that the robot could move away from this position, within the limits of a unit, was calculated. A blocked unit was defined as a unit where the minimum distance from the robot in the configuration at the centre of the unit, to the nearest obstacle, was less than the maximum distance that the robot could move within the unit.

The pathfind problem was now to find a series of neighbouring units which were clear between the start configuration and the goal configurations.

8.4 Theory

8.4.1 Solution to the position problem

Figure 8.1 shows a diagram of the robot model, giving dimensions of the distances between links (105, 54, 22mm), and link lengths (385, 376mm). The

origin of cartesian coordinates was set at B1. The positions of the other 5 points were calculated as follows:

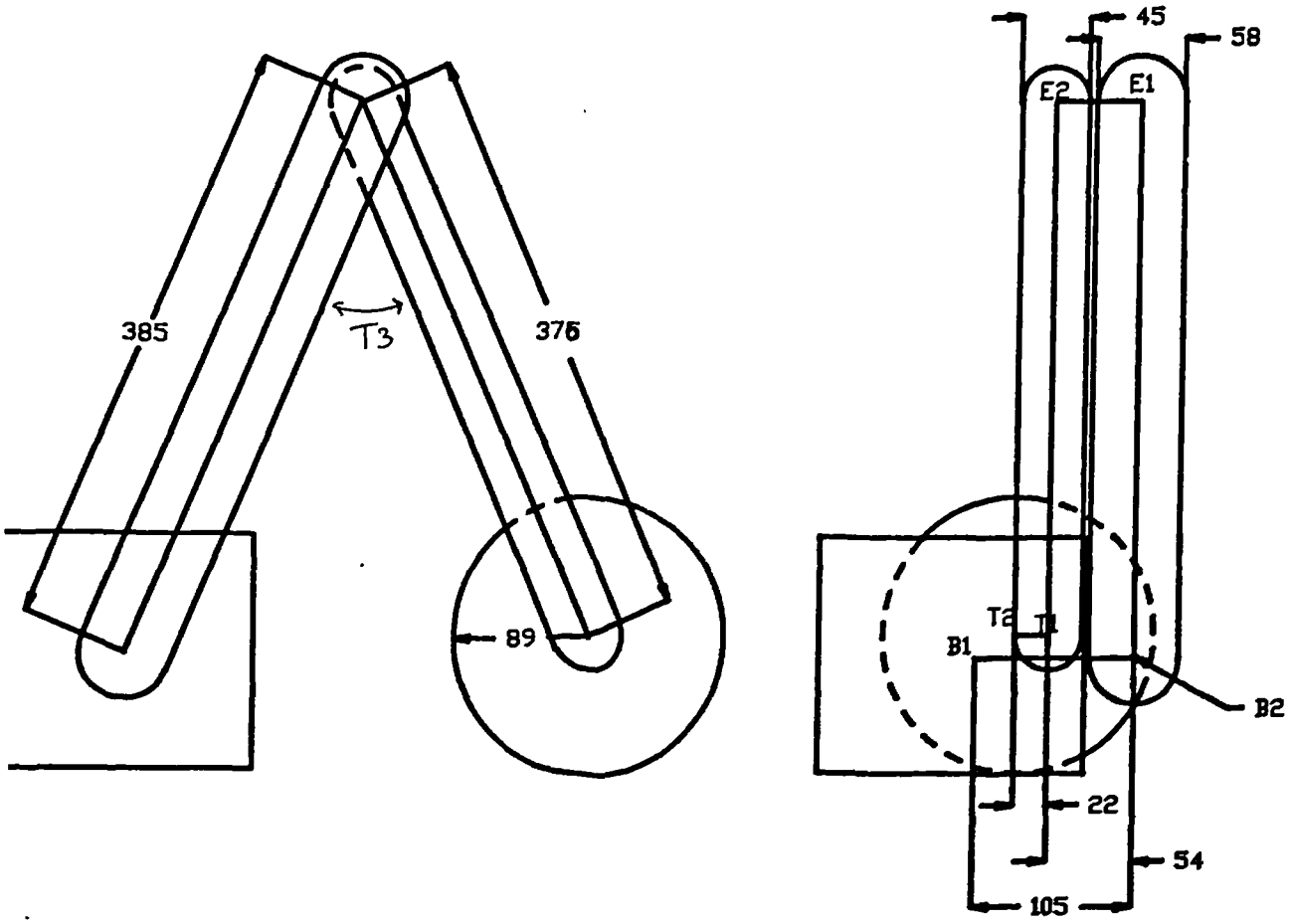


Figure 8.1 Model Robot

$$B2x = 105.sinT1$$

$$B2y = 105.cosT1$$

$$B2z = 0 \qquad (8.2)$$

$$E1x = B2x + 385.cosT1.cosT2$$

$$E1y = B2y + 385.cosT1.sinT2$$

$$E1z = 385.sinT2 \qquad (8.3)$$

$$E2x = E1x - 54.sinT1$$

$$E2y = E1y + 54.cosT1$$

$$E2z = E1z \qquad (8.4)$$

$$P = T1 + T2 - Pi \qquad (8.5)$$

$$T1x = E2x + 376.cosT2.cosP$$

$$T1y = E2y + 376.sinT2.cosP$$

$$T1z = E2z + 376.sinP \qquad (8.6)$$

$$T2x = T1x - 22.sinT1$$

$$T2y = T1y + 22.cosT1$$

$$T2z = T1z \qquad (8.7)$$

More general details of robot coordinate transforms may be found in Paul (1981)

8.4.2 To transform obstacles one at a time

The most straight forward method for transforming obstacles into joint space is to check each unit of the joint space graph for intersections with each obstacle. However, this method uses large amounts of computing time. The actual amount of time is proportional to the number of obstacles and the number of units.

A faster method is to consider each obstacle at a time and test all the units which could possibly contain the transformed obstacle. The following is the basic method by which this can be achieved.

Find a unit on the joint space graph where the robot intersects the obstacle. Then test all the neighbouring units to see if they are blocked as well. With each blocked unit found, test all its neighbours.

Providing that the transformation produces a single obstacle shape, the method will always calculate the complete transformation.

8.5 Program description

The basic method described above was developed further in order to further reduce computation time. However, the principles have remained the same.

The program described in this section was relatively short (700 lines of pascal). The important procedures were called Espace, ObstGraphCalc, Fill, Test position, Expand position, Put on list and Pull off list. The flowcharts are shown in figures 8.2 to 8.9 respectively. The procedure descriptions are given below.

8.5.1 Procedure 'Espace'

The obstacle data was entered by a separate program and stored on disc. Thus the first task was to read the obstacle data.

Firstly the data structure was initialised. The limits of the graph corresponded to the angular limits on the robot's joints. Limits were also set for the robot's workspace so that obstacles outside this workspace could be ignored.

As the graph used a limited number of positions at which intersection checks were carried out, only a limited number of trigonometric values needed to be used. To save computer time all the trigonometric values which were needed were calculated at the start, so that time was not wasted in repetitively evaluating trigonometric functions.

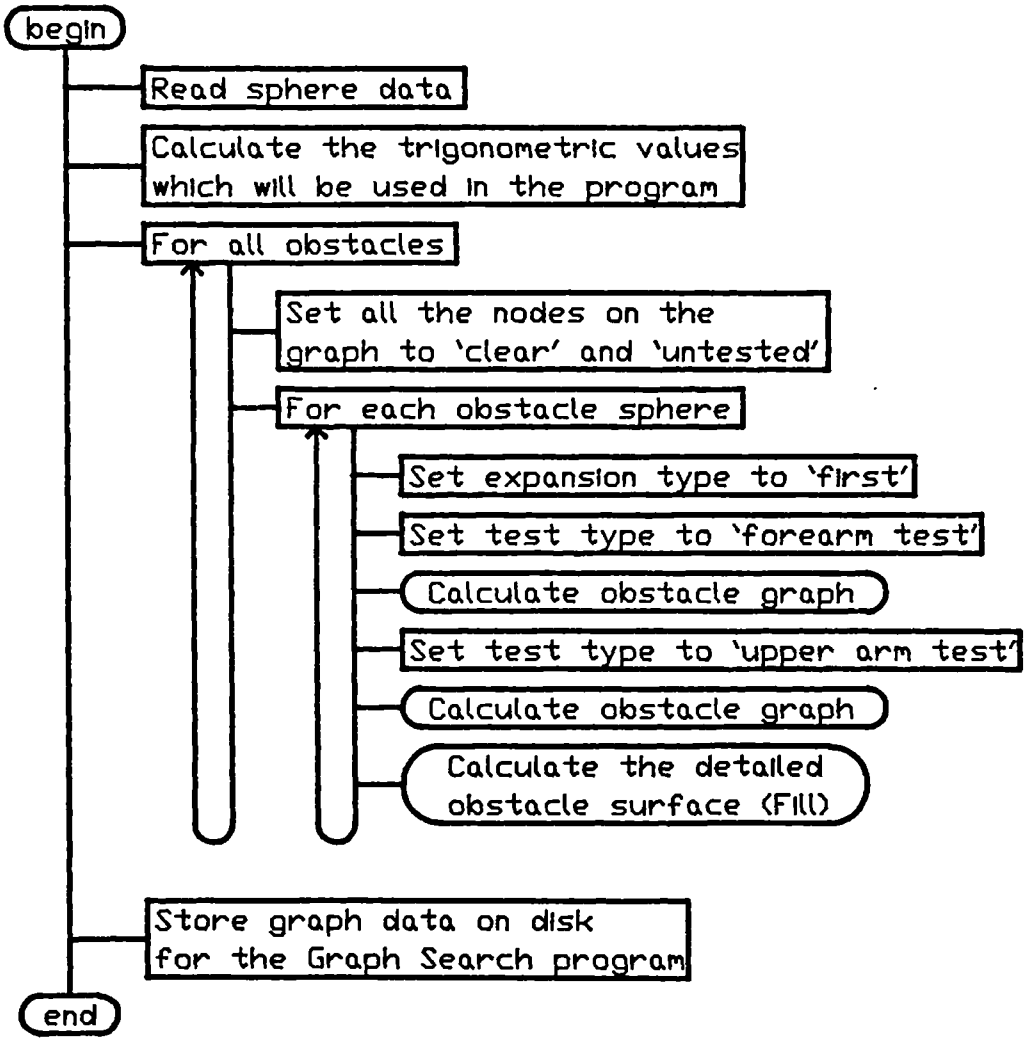


Figure 8.2 Flowchart of Espace

Before the obstacles were calculated all the units in the graph were set to 'clear' status. Four other flags were used with each unit, these were, 'new obstacle', 'forearm tested', 'upper arm tested', and 'on list'. Each unit was stored as one byte of computer memory, and the flags used one bit each of the byte.

Initially, each obstacle was transformed into a sparse graph. This task was split into two, calculating the upper arm graph, and calculating the forearm graph. A sparse graph, of a transformed obstacle, was one in which only one in eight units of the graph were tested. Clear units in the centre of a sparse graph

were set to blocked and then the outside blocked units were expanded so that the surface of the transformed obstacle was defined.

8.5.2 Obstacle graph calculation (Procedure : ObstGraphCalc)

A configuration was calculated at which the part of the arm under consideration was closest to the obstacle centre. If the forearm was being considered, then the configuration where the gripper was at the centre of the sphere was calculated. For the upper arm, the configuration was calculated for which the centre line of the upper arm pointed directly at the sphere centre. If the obstacle was within the workspace of the link being tested, then the configuration found above was then the first unit of the transformed obstacle.

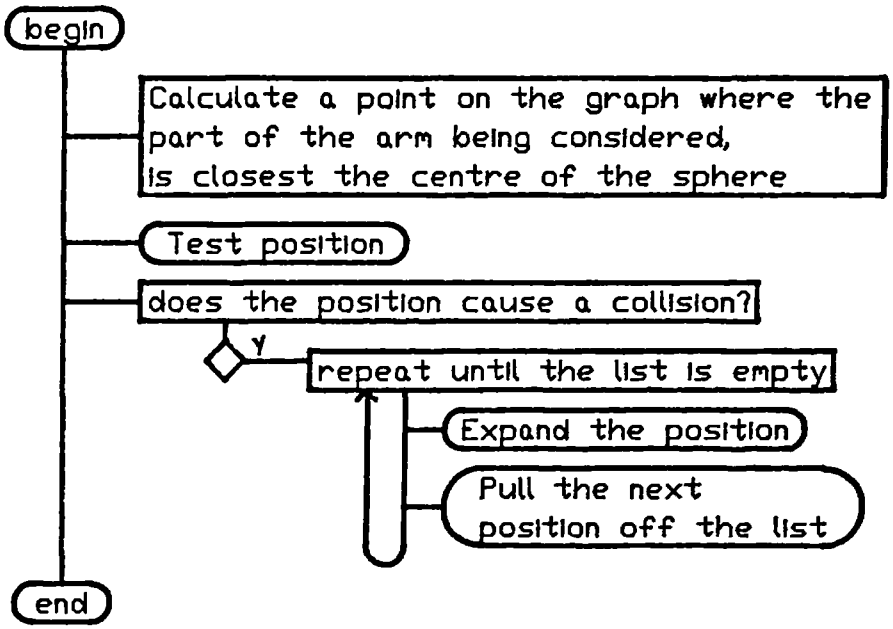


Figure 8.3 Flowchart of ObstGraphCalc

The first configuration was tested, and if it was blocked then its neighbouring units were also tested. If they were blocked then their neighbours were checked, and so on until the whole obstacle transformation was found.

8.5.3 Procedure 'Fill'

The first obstacle graph was a sparse graph. The units which were tested were spaced two units apart. For example if unit (4,4,4) was blocked then the units (6,4,4), (4,6,4), (4,4,6) were tested. Thus on average one eighth of the obstacle transformation was calculated.

All units were set to blocked, which had any two opposite neighbouring units which were also blocked. Any units which were on the edge of the now solid obstacle were recorded on a list. Then all the neighbours of the units on the list were tested, and the process repeated until the edges were completely defined.

8.5.4 Procedure 'Test position'

'Test position' solved the position problem and calculated the minimum distance between the obstacle and the robot arm, provided that it had not done the calculation before. If the test was carried out and the position was blocked then this position was put onto the list.

8.5.5 Procedure 'Expand position'

'Expand position' extended an obstacle's transformation by testing neighbouring units to those which were blocked. If a sparse graph was being generated then 'next but one' units were tested. For the detailed graph the 'next door' units were tested.

8.5.6 Procedures 'Put on list' and 'Pull off list'

Nodes which were found to be blocked were stored on a list of units which were to be expanded later. When a unit had been expanded it was pulled off the list. When all the units on the list had been exhausted the obstacle transformation was complete.

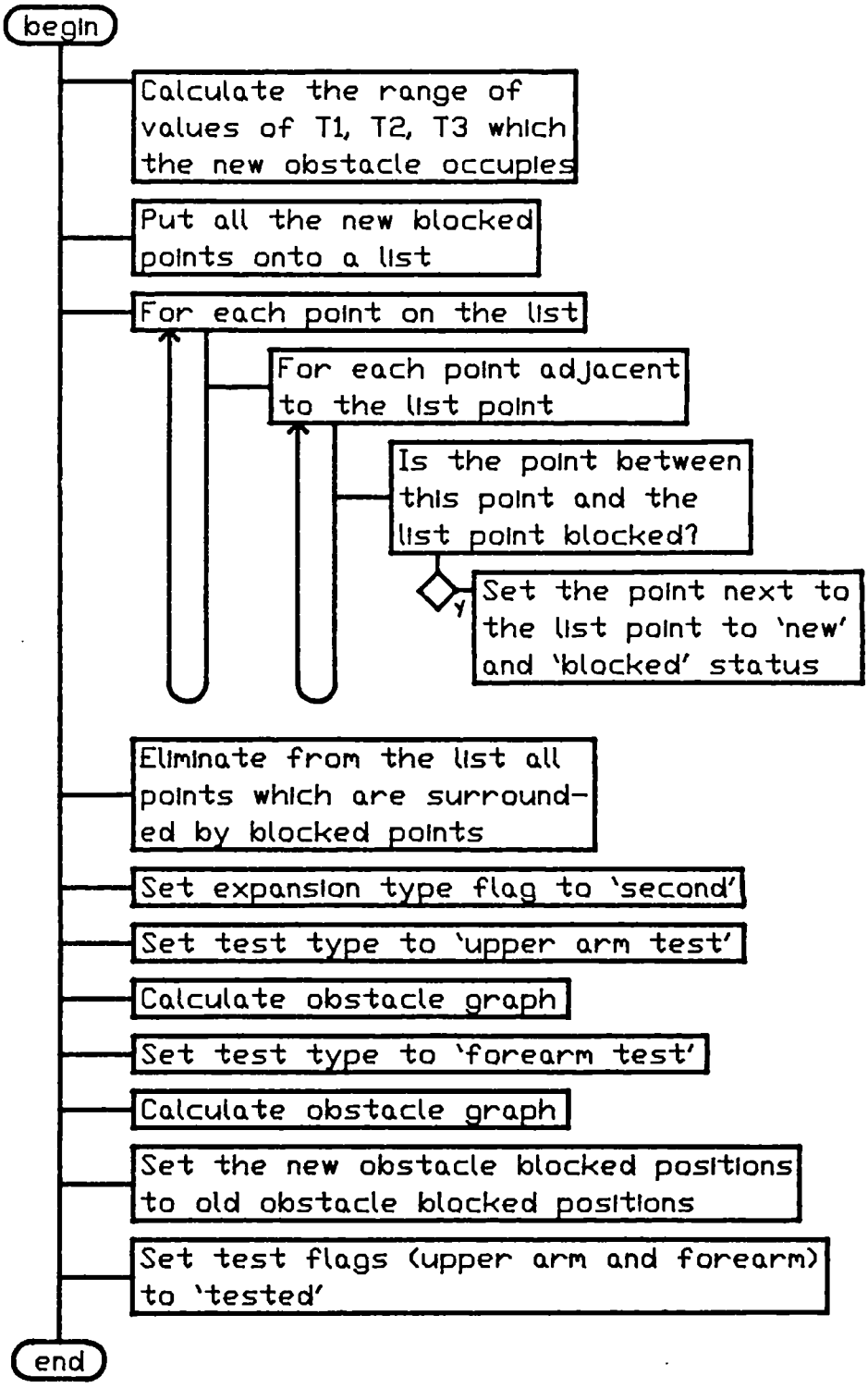


Figure 8.4 Flowchart of Fill

8.6 Results of obstacle transformation

The obstacle transformation program was tested for sizes and types of obstacles. The most important consideration for the program was that it should be

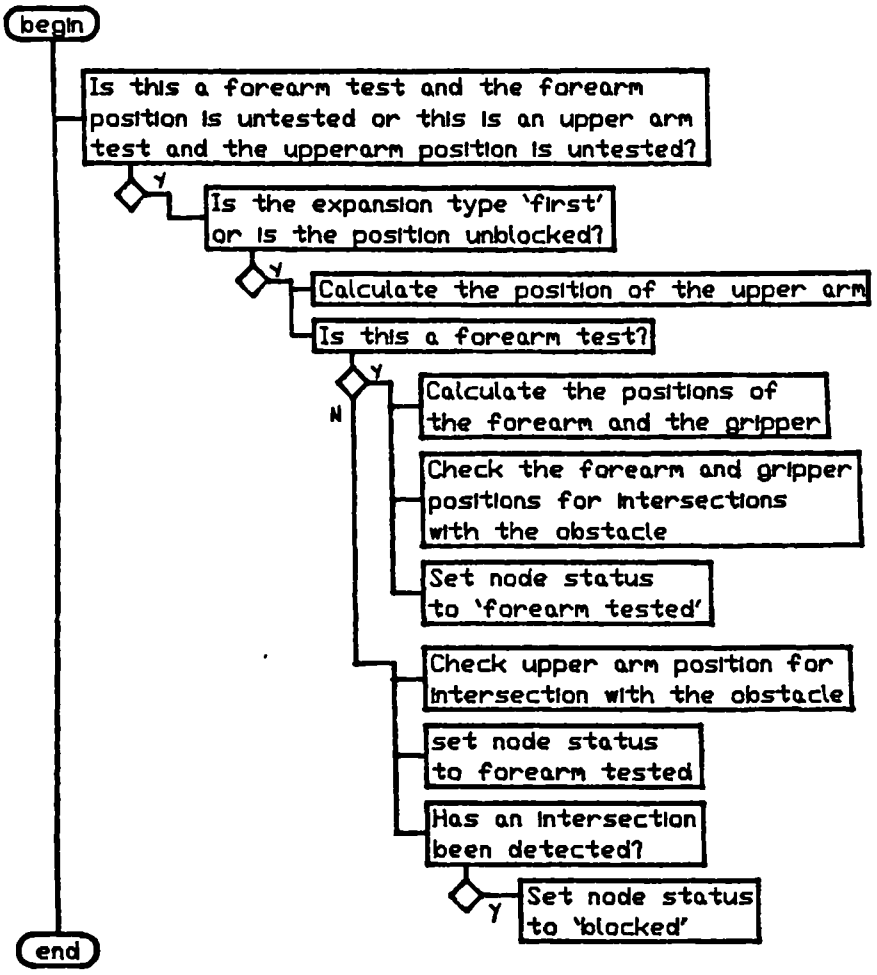


Figure 8.5 Flowchart of Test position

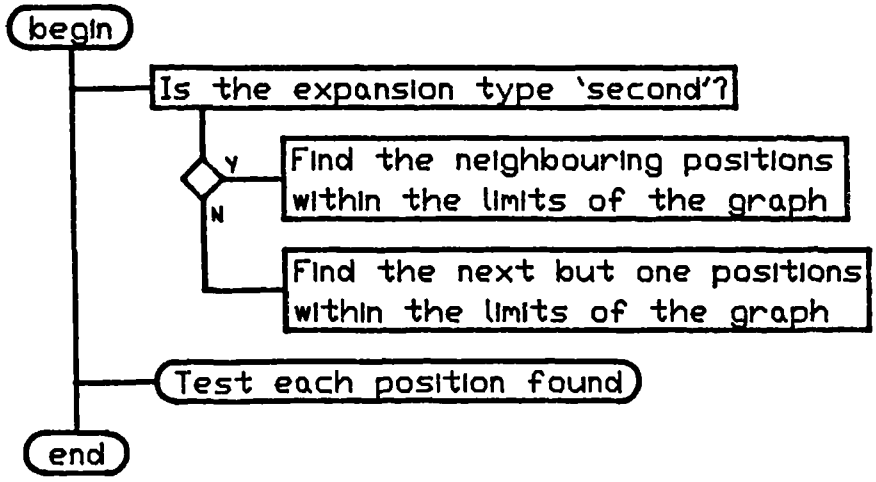


Figure 8.6 Flowchart of Expand position

fast, and so the times for calculating obstacles were recorded.

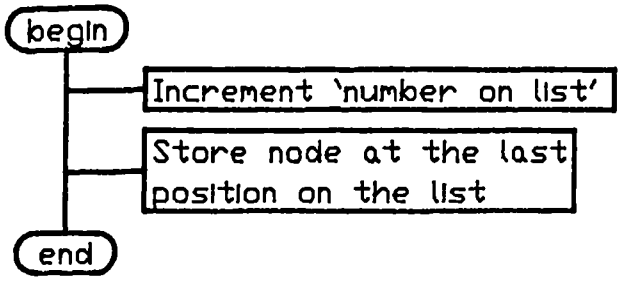


Figure 8.7 Flowchart of Put on list

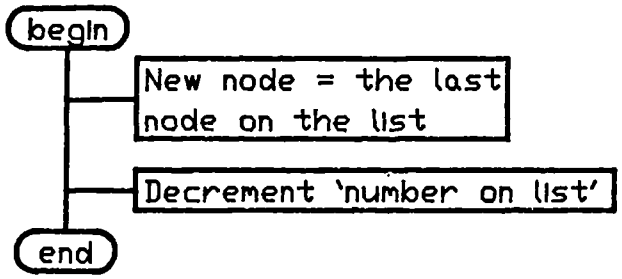


Figure 8.8 Flowchart of Pull off list

Figure 8.9 shows a graph of varying sphere radius against time of computation. The first point on the graph represents the time of calculation for a sphere with a radius of 1mm, the transformed obstacle for this example is shown in figure 8.10. However, even though this was almost a point object the transformation still took 26 seconds. This was because there was a wide range of configurations where the robot intersected the sphere. The sphere's centre was at coordinates (50,650,50) which is a position outside the joint space of the upper arm but inside the joint space of the forearm. However once the obstacle radius increased above 200 mm part of the sphere intersected the upper arm joint space. Thus the workspace occupied by the sphere suddenly increased and so the calculation time increased also.

Figure 8.11 shows a graph of calculation time vs. workspace volume. This is approximately linear, which means that the calculation time for one sphere is

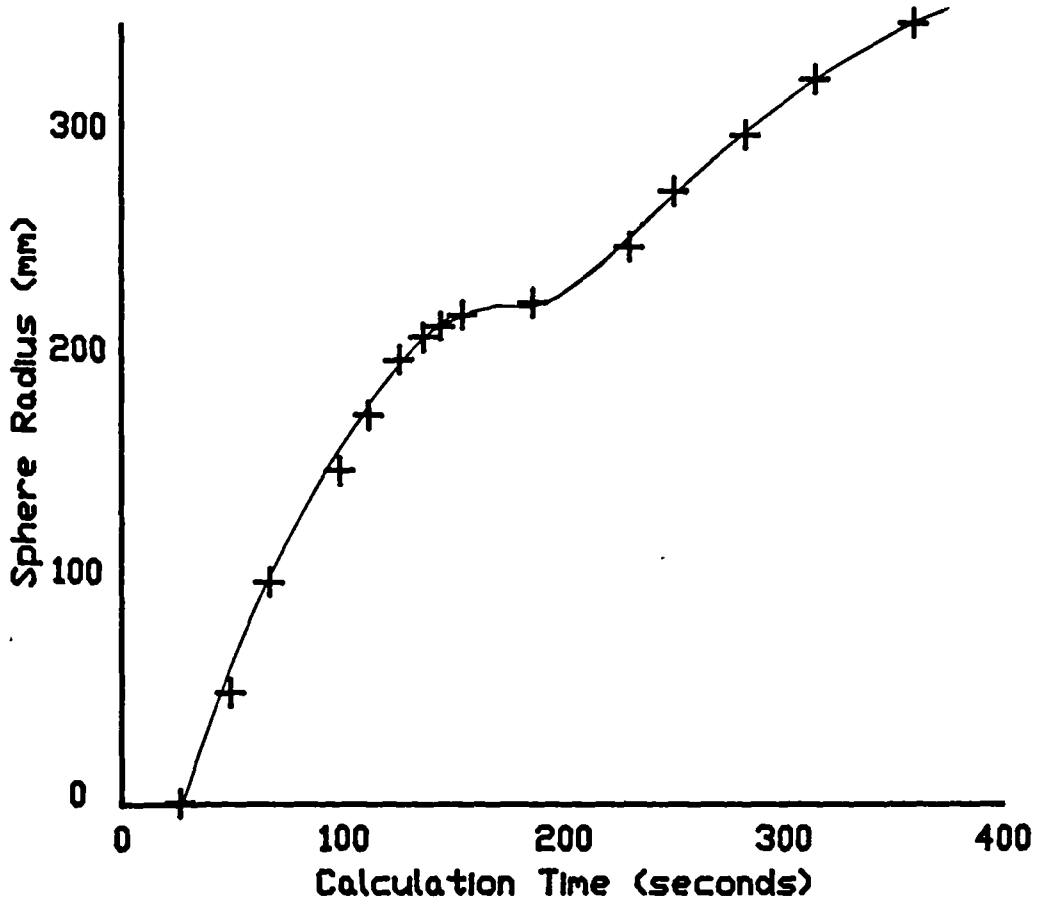


Figure 8.9 Sphere radius vs calculation time

approximately proportional to the number of units tested, the total number of units being the workspace volume.

Figure 8.12 shows a graph of the calculation time for modelling a cube using different complexities of model. A cube was modelled first as a single sphere of the smallest radius which would enclose the cube (point a). It was then modelled by two smaller spheres (point b), four (point c) and then eight spheres (point d). It may be seen that the calculation time increased linearly with the number of spheres. The reason for this was probably that the overhead of calculation for each sphere was much greater than the saving in time which was achieved as the spheres became smaller.

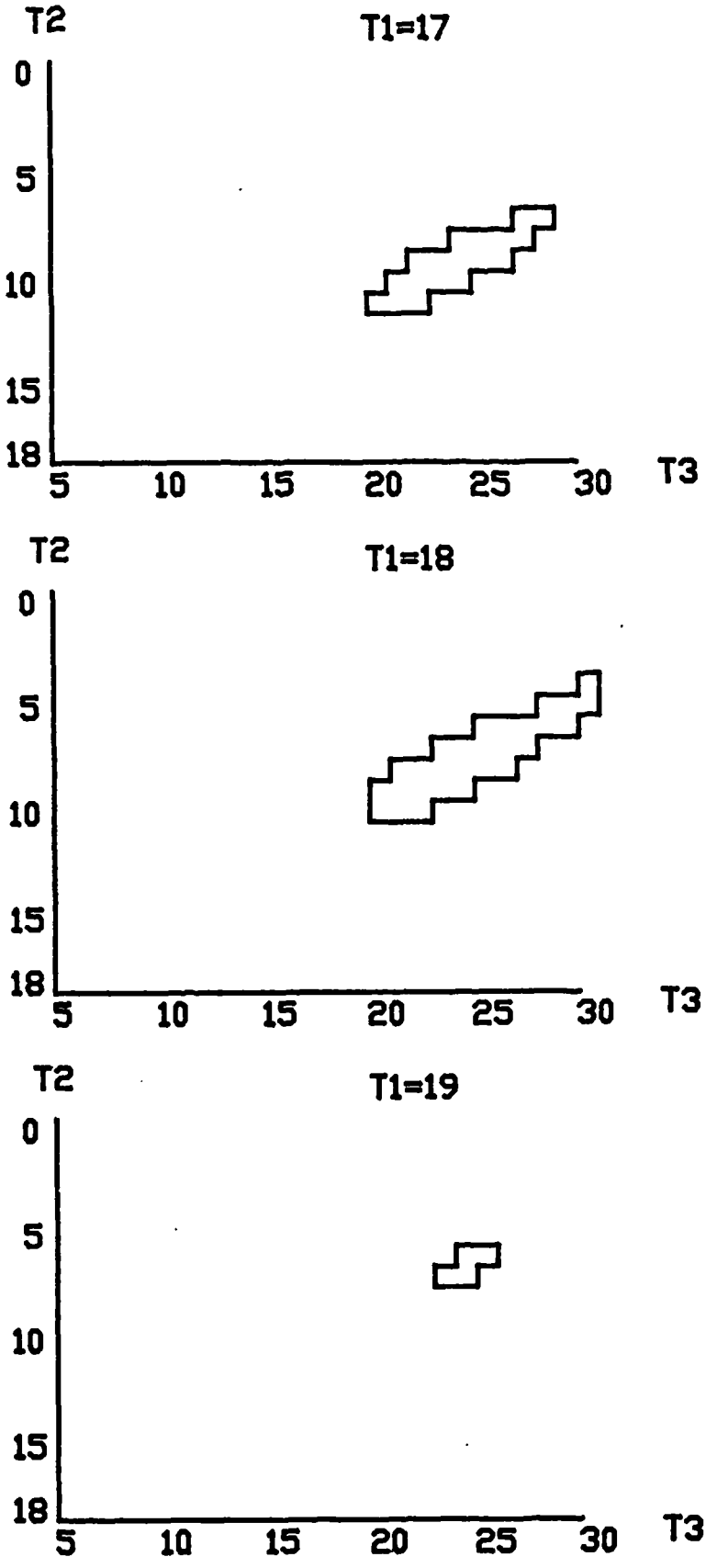


Figure 8.10 Transformed obstacle point

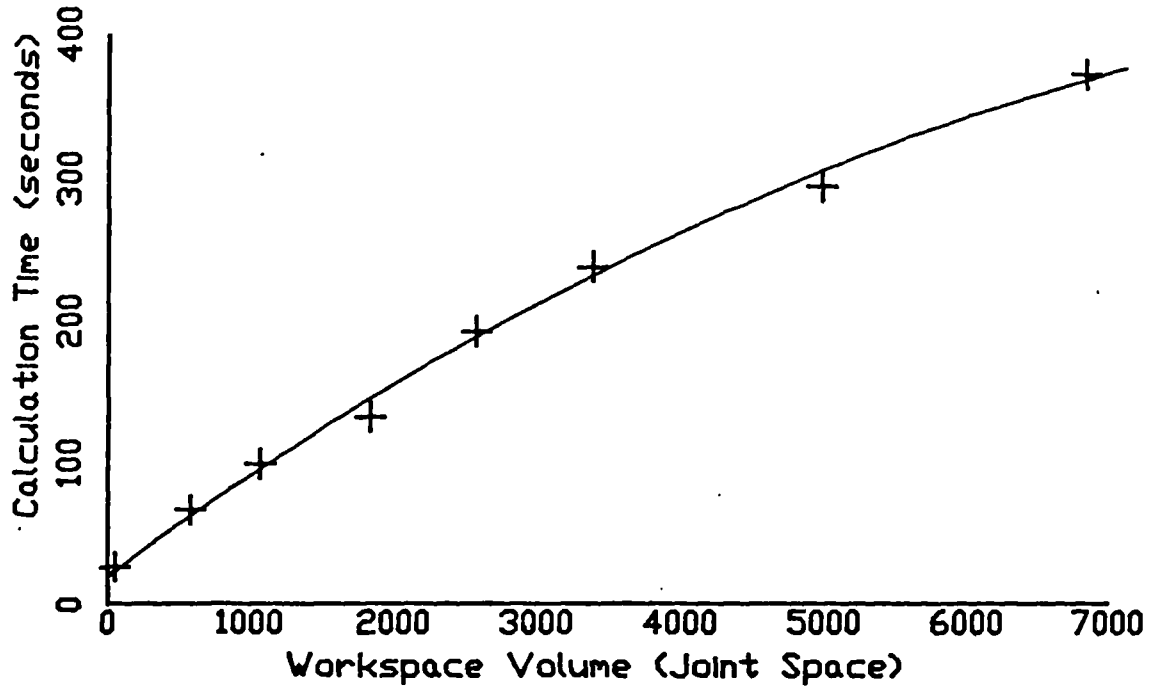


Figure 8.11 Calculation time vs workspace volume

8.7 Discussion and conclusions

It may be noted from the results that the computer time required for obstacle transformations was very high. To have computer times of some minutes for single obstacles seems impractical at first especially when computers solve many problems in milliseconds. However if the workspace has to be transformed only once, for many path planning operations then the overhead of conversion is acceptable.

It has been seen that the calculation time is approximately proportional to the workspace of an obstacle and the complexity of the obstacle representation (number of spheres). These results are only relevant to the particular program used. Judging from past experience the shape of graphs will change significantly as different techniques are used to improve the program.

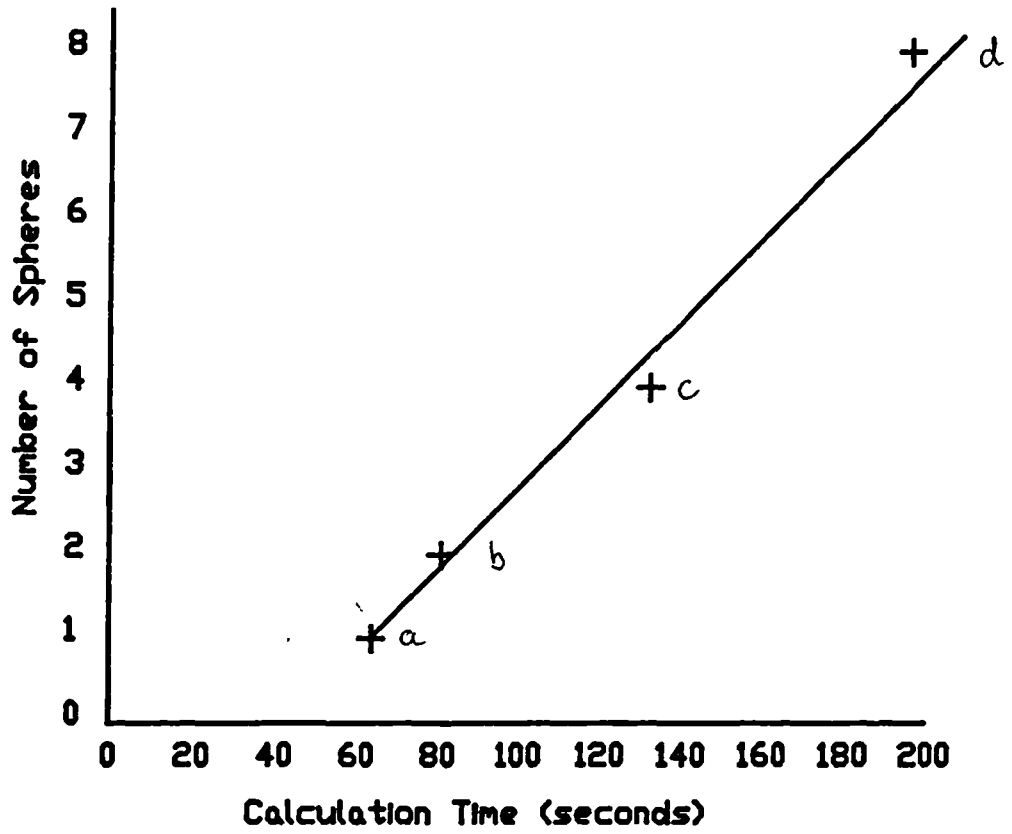


Figure 8.12 Calculation time vs number of spheres

The program used was relatively short (700 lines of pascal), and much improvement may still be made. Having discovered the shapes of transformed obstacles from this technique it may be possible to define a set of formulae which give the shapes of transformed obstacles directly.

Currently the transformation algorithm can only transform spherical obstacles. Spherical obstacles were chosen because they require much less computational effort for intersection calculations than other types of obstacle. However most real objects would be more accurately represented by polyhedral obstacles. Therefore there must be a point at which increasing the number of spheres, in order to increase the accuracy of the model becomes impractical. In this solution it would increase the calculation time beyond the time required to do the

transformation for a polyhedral obstacle. This trade off point still has to be investigated.

For many circumstances the obstacle transformation may only be performed once. For example in a situation where a robot services several machines it may be required to service the machines in an unpredictable order. This problem requires automatic path planning but the workspace remains constant. Another example is where a robot is required to pick components in random order from a conveyor. The path may need to change substantially, but the basic obstacles such as the conveyor, machines and fixtures do not vary.

Another factor which affects the transformation time and model accuracy is the range of robot configurations in each unit of the graph. The range of values for each degree of freedom for the robot in this example was five degrees. The total workspace contained 21,312 units. If the range of values were extended to ten degrees for each degree of freedom then this would reduce the number of units by a factor of eight and the calculation time would be reduced by a similar amount.

The range of values for each degree of freedom was set to the same value for simplicity. However certain degrees of freedom may be more important than others. If this is the case then smaller ranges of values should be used for the more important robot axes.

For manoeuvring the workpiece close to obstacles, such as putting a part into a vice, it is possible to take into account the degrees of freedom of the gripper. This would create a graph of more than three dimensions, which would have to be searched. The drawback is the size of graph, and so it could only be used for small areas of the total workspace.

In order to save memory a dynamic size of graph could be used. In large areas of clear or blocked space the unit ranges would be large, but in the areas

around the surfaces of obstacles the graph would be more refined, using smaller units. An example of such a graph is shown in figure 8.6. The larger grid was defined first, then units which contained both space and defined obstacles were refined into smaller units.

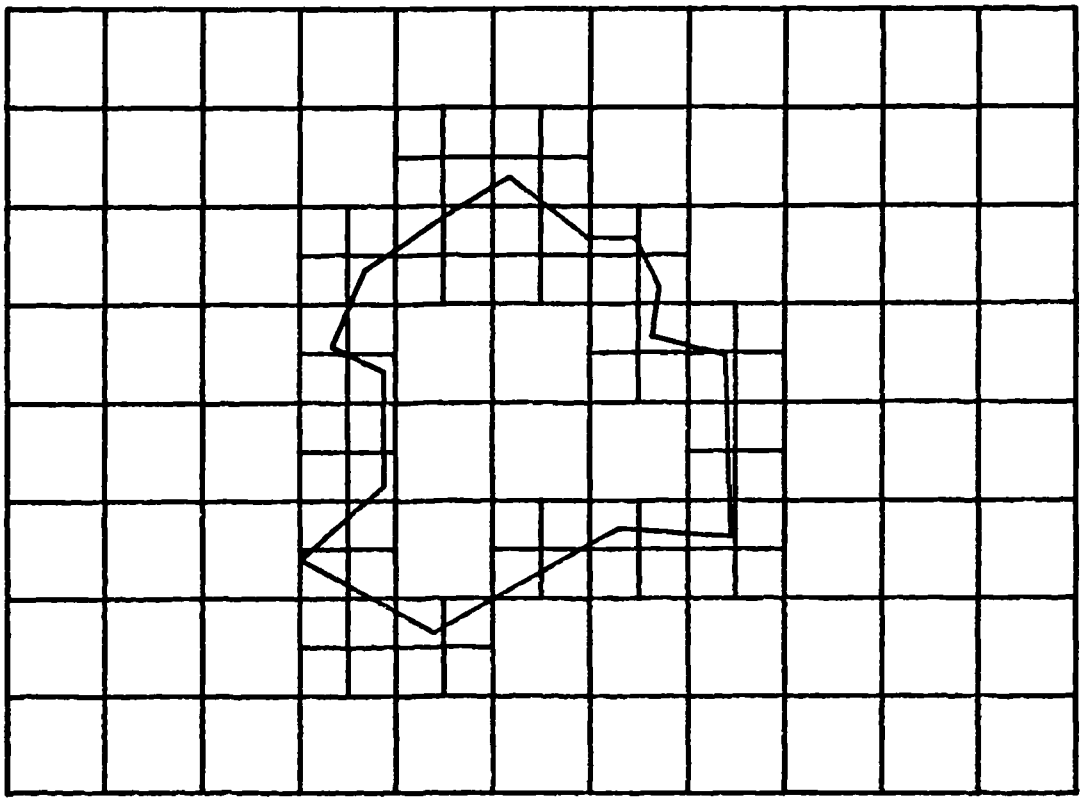


Figure 8.13 Example of variable unit size

CHAPTER 9

GRAPH SEARCHING

9.1 Introduction

Having transformed the findpath problem from real space into the joint space of the robot, the findpath problem is reduced to that of finding a path for a point, from position S to position G between various obstructed volumes of space. The transformed space may be regarded as a graph. The centre of each unit in joint space becomes a node on the graph and the branches between nodes form a grid structure, so that there is a branch between each node and its closest neighbours.

The graph searching problem is a familiar one in many research fields. An example of this is the travelling salesman problem which is to find the shortest route by car from town A to town B, or to find the shortest route which takes the salesman between a number of destinations.

For the robot findpath problem the nodes form a grid of points in joint space. The coordinates of any particular node can be represented by coordinates (i1, i2, i3, ... in), where i1 to in are integers. Branches exist between all adjacent 'clear' nodes. Adjacent nodes are defined as nodes which had a vector between them with an absolute value of 1. Thus the path between S and G is a connected set of nodes on the transformed graph containing nodes S and G. The cost of the path is defined as the total number of units that the path passes through.

As the directions that the path could take are restricted, the path found can not be regarded as an optimum path. However, an optimum path is often impossible to find as many factors have to be taken into consideration including execution time, energy used, wear on the robot, safety from hitting obstacles, etc.

However the path found can be regarded as a good estimate. The method also ensures that paths are always found if they exist.

It was found that the basic method of path finding could be changed, or 'tuned', to produce different characteristics of calculation time, path length, etc., depending on the obstacles present. For simple environments, a strategy which aimed more directly at the goal produced a solution most quickly, where as a more conservative strategy was required for complex environments.

9.2 Theory

In chapter 8 it was shown that as the detail of the transformed obstacles was increased so the calculation time increased. In order to find a satisfactory solution to the findpath problem the transformed obstacles had to be as detailed as possible. This ensured that a path was found if one existed and that the most efficient solution possible was produced.

The required detail of obstacles meant that the number of branches between nodes had to be restricted. The graphs developed previously have considered all possible branches. In this case such a method was impractical because the graph contained thousands of nodes and millions of possible branches.

The method used instead was to consider only the adjacent nodes of the node being expanded. Thus for a two dimensional graph, only 4 nodes were considered. Of these, one of the nodes was the predecessor of the current node, which left three possible new directions for the path. In general for a n dimensional graph there are 2n-1 possible paths from each node except for S and G where there are 2n possible paths.

The cost of a path, Cg, was defined as the length of the path in joint space. This was equivalent to the number of branches in the path. This cost function

meant that many different paths had the same cost value. For example figure 9.1 shows three paths which are quite different but have the same value of Cg.

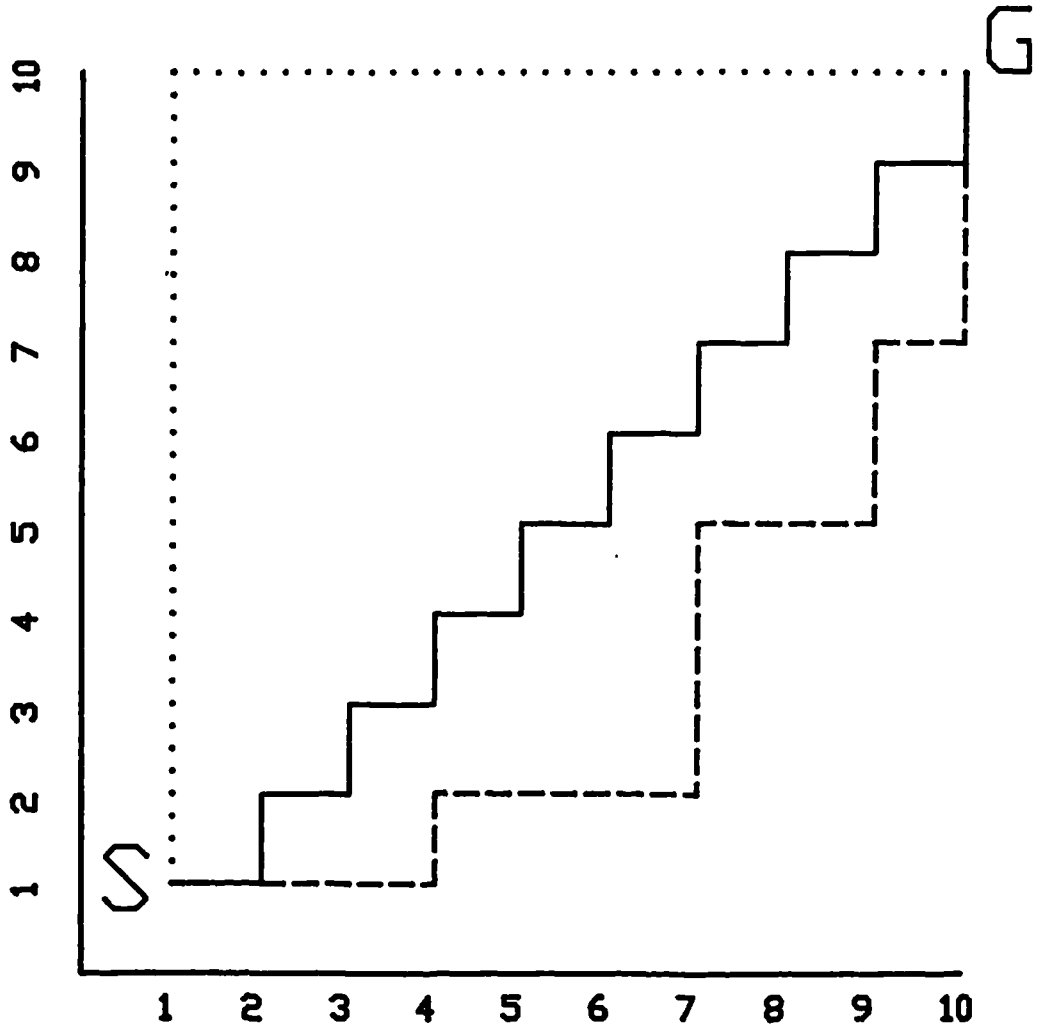


Figure 9.1 A graph of alternative paths with the same cost

When a path was being planned, two cost functions were used; Cg was the cost of a path from S to a particular node n and Ch was the estimated cost to n from G. It was found that by changing Ch different paths were found for the same problem. However the optimum path length was only found when Ch was equal to the minimum possible path length from the opennode to G.

9.3 Program Description

9.3.1 Main program (figure 9.2)

Firstly, the data required for the algorithm was read from the disk. This included S, G and the obstacle in joint space. A check was made to ensure that the S and G were not obstructed before the rest of the program was executed.

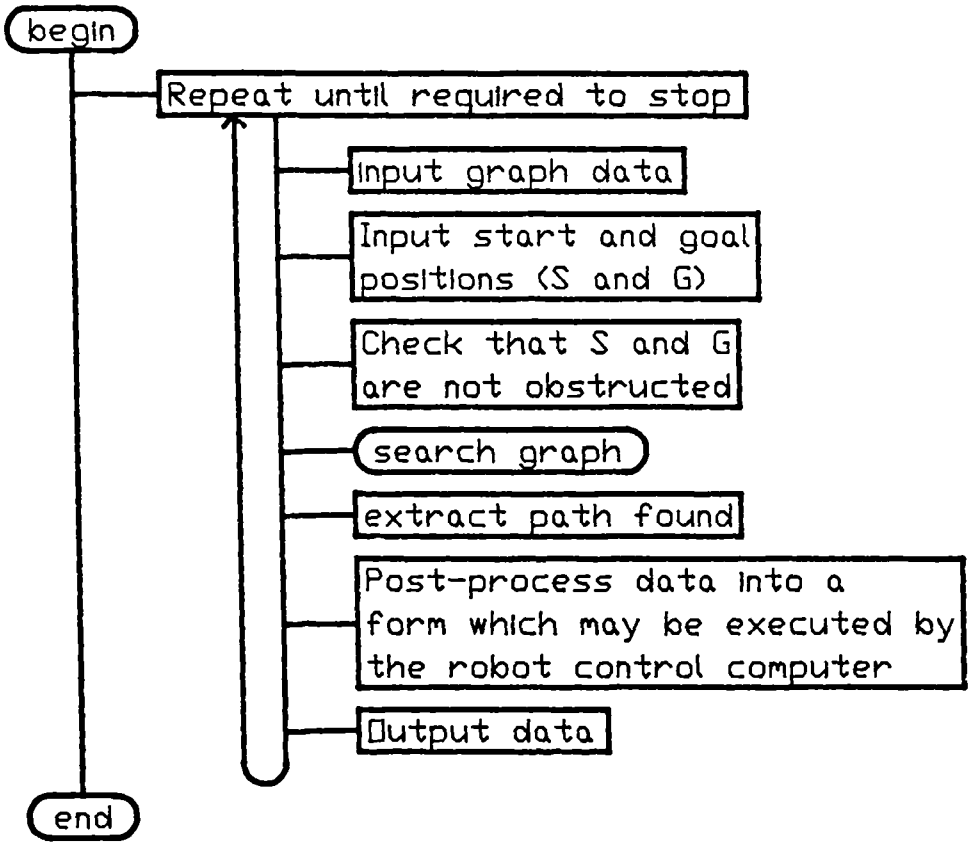


Figure 9.2 Flowchart of the graph search program

The graph was searched by the procedure 'Search graph', which was an algorithm which searched through the graph starting at S and continuing until G was reached. Then the path was traced back from G to S and this information was post-processed into a form which the robot control computer could accept. The trajectory locus was then down loaded to the robot control computer via a serial link.

9.3.2 Procedure 'Search graph' (figure 9.3)

This procedure was similar to that described in chapter 6. The opennode was set to S initially, then it was 'expanded' and the next opennode chosen. This process was repeated until G was reached.

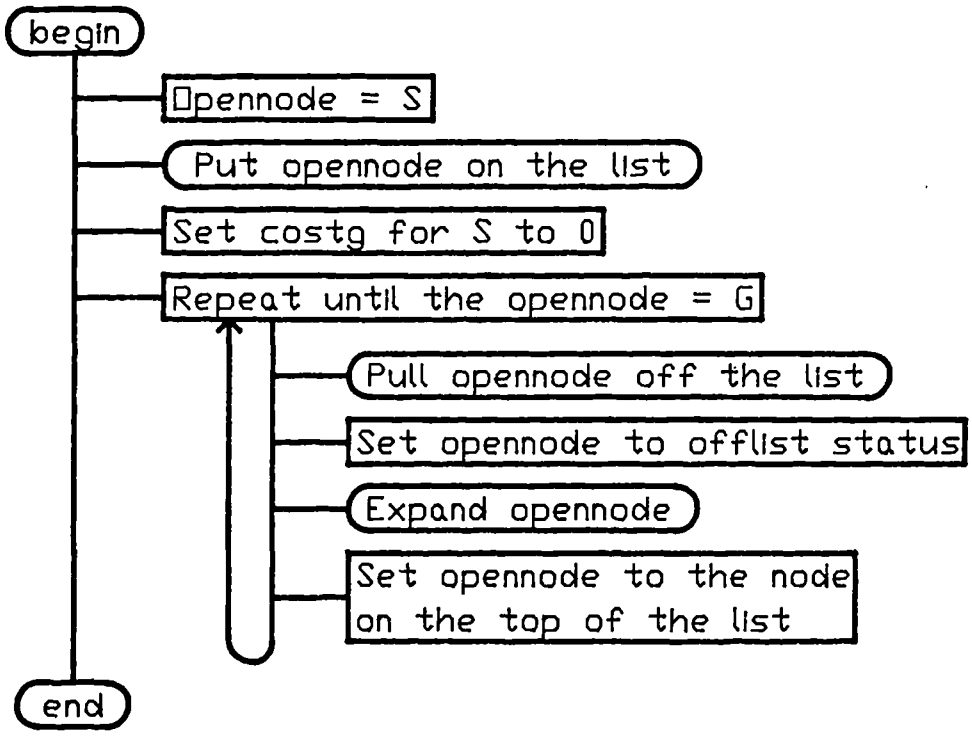


Figure 9.3 Flowchart of Search Graph

9.3.3 Procedure 'Expand' (figure 9.4)

Each adjacent node of the opennode was checked to see if it was a candidate for the next position on the path. The node was not a candidate if either the position was blocked or if it was the predecessor of the opennode. If the node passed these tests then Ch was calculated and it was put onto a list of nodes to be expanded in the future.

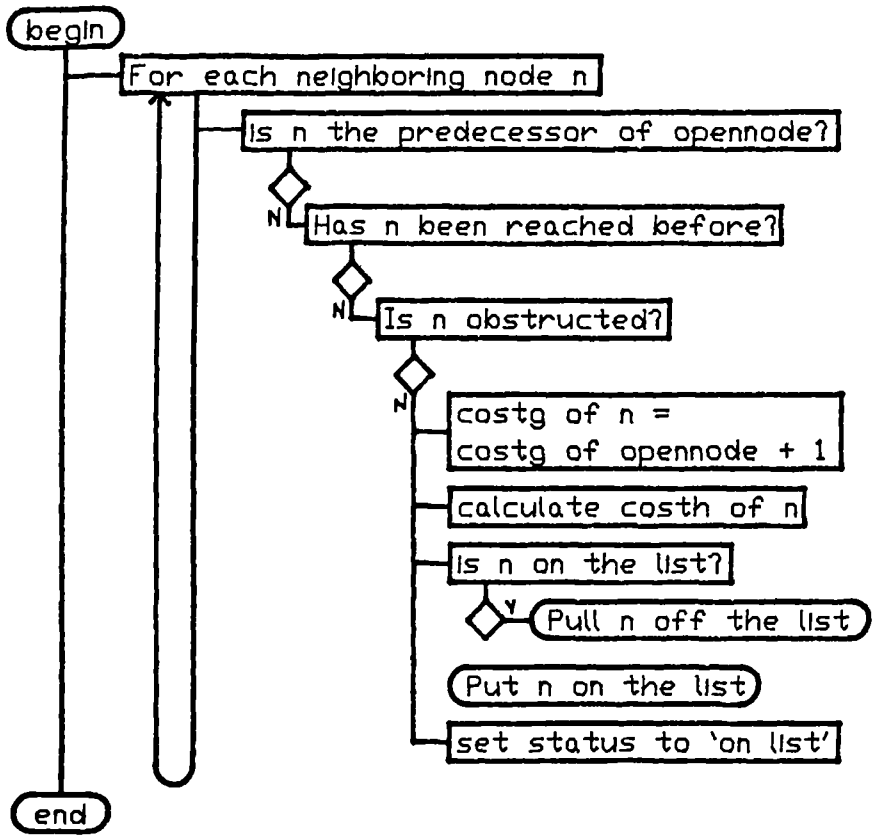


Figure 9.4 Flowchart of Expand

9.3.4 Procedures 'Putonlist' and 'Pullofflist' (figure 9.5-6)

A list of the nodes which had yet to be expanded was kept. The nodes were kept in order of cost (Cg + Ch) so that the next node to be expanded was always at the top of the list.

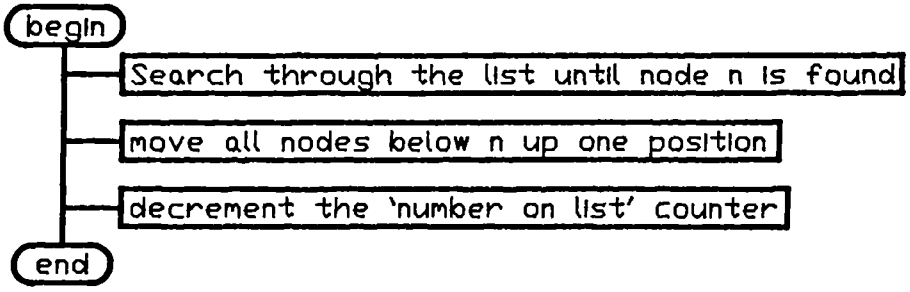


Figure 9.6 Flowchart of Pull off List

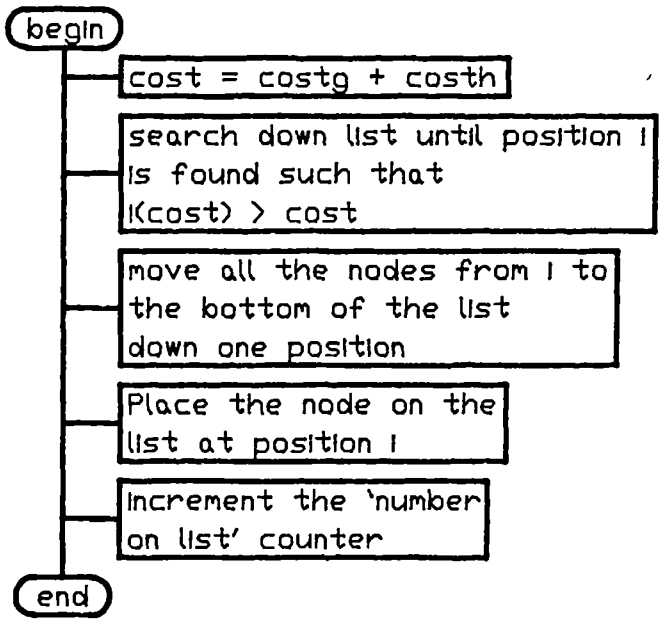


Figure 9.5 Flowchart of Put on List

9.4 Results for two dimensional graphs

It is simplest to look at the shapes of paths which were found by searching two dimensional graphs, because of the ease of representation. Figures 9.7(a) to (c) each show a very small size of graph (100 nodes) with increasing complexities of paths.

The cost function Ch in these examples was defined as the minimum distance of the path from the node to G. It may be seen that by increasing the size of obstacles, the calculation time actually decreased from 2.5 seconds to 1.5 seconds. Increasing the complexity of the graph had no effect on the calculation time.

The graphs given in figures 9.8(a) to (c) represent larger graphs (2500 nodes). Two different cost functions were investigated, Ch1 and Ch2, the paths that these produced are shown on the figures. Ch1 was the same as Ch defined above and Ch2 was defined as the square of the value of Ch1. It may be seen that Ch1 took longer for simple environments such as 9.8(a) compared with the more complex

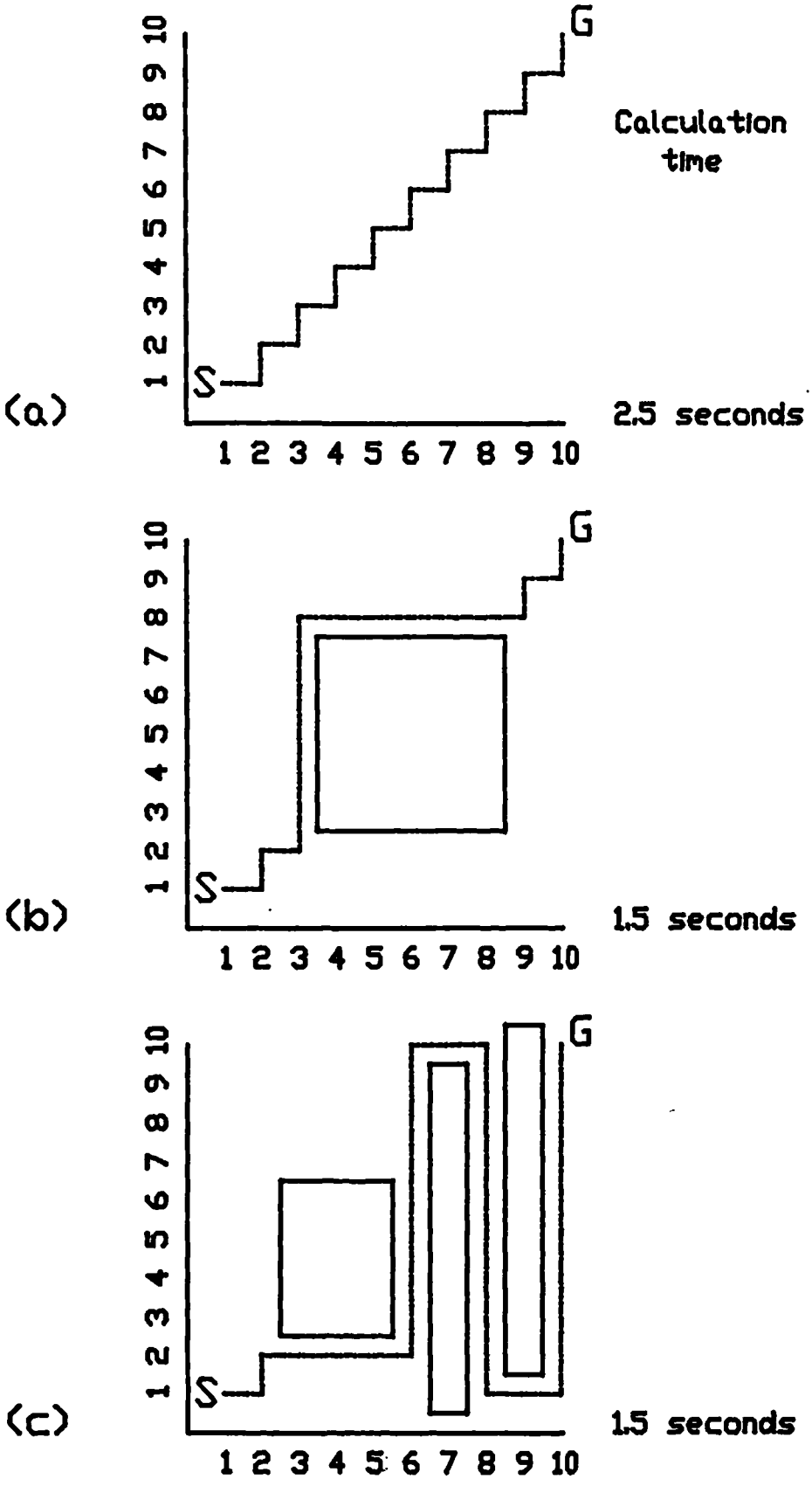


Figure 9.7 Examples of paths through two dimensional space

environments of 9.8(b) and (c). However, it had the advantage that it always produced the minimum cost path.

note :- The terms minimum cost path and optimum path, used above and for the rest of the chapter refer to paths which have the shortest lengths possible, in joint space, and which are also sets of branches of the graph.

Figure 9.8(b) shows how Ch2 tended to pull the path towards G. This caused the path to be bent away from the optimum path in the direction of G. Because of this property Ch2 produced small calculation times in most environments. However figure 9.8(c) shows an example where this property was a disadvantage, and consequently Ch2 produced a large calculation time.

The advantages and disadvantages may be summarised as follows.

Ch1 advantages

(a) It always found the optimum solution.
(b) The solution time was relatively predictable.
(c) The solution time reduced as the obstacle volumes increased.

Disadvantages

(a) The solution time was relatively long.

Ch2 advantages

(a) The solution time was very short for most environments.

Disadvantages

(a) In certain circumstances an optimum solution was not achieved.
(b) In certain extreme circumstances the calculation time was very long.

Both Ch1 and Ch2 provided solutions if they existed.

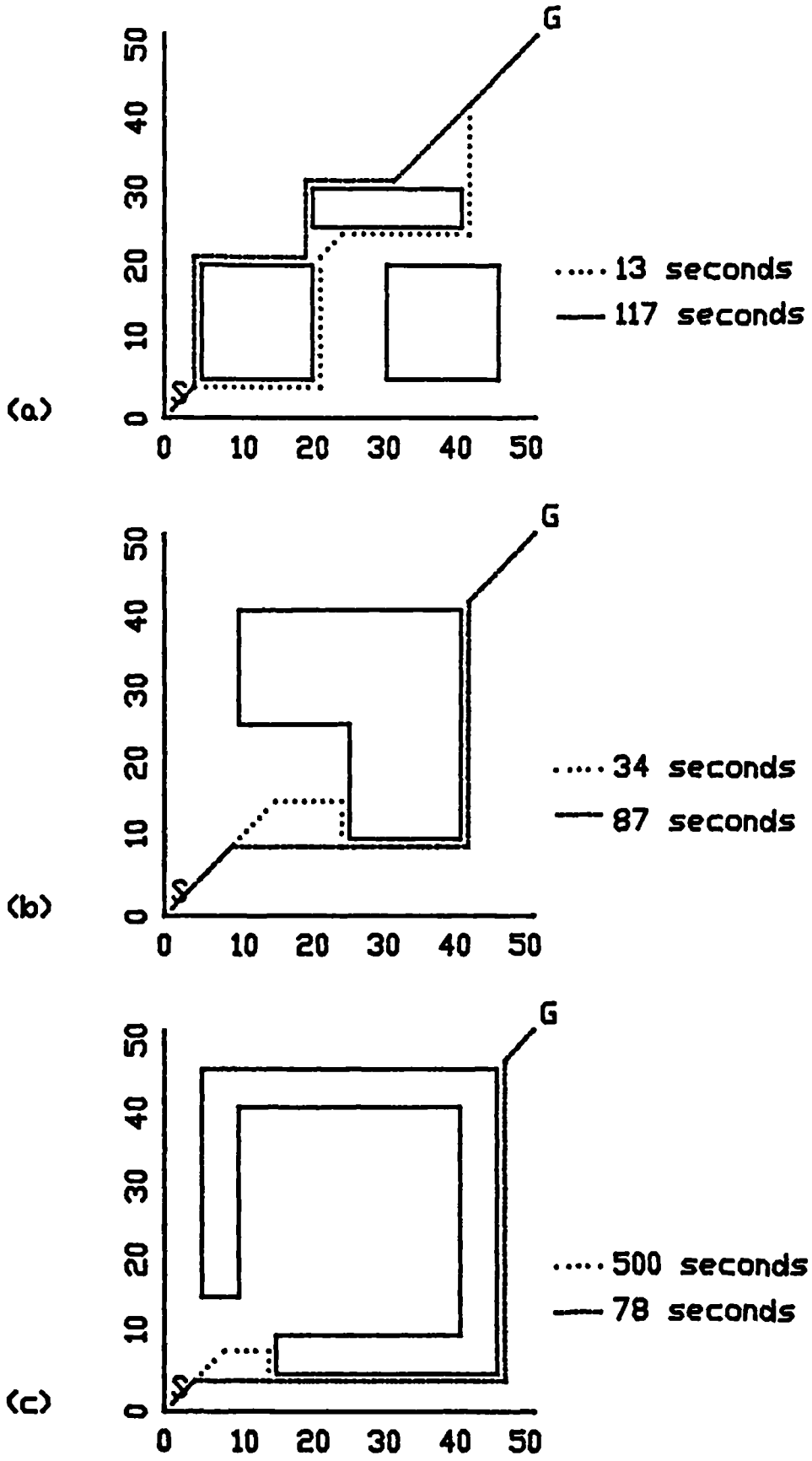


Figure 9.8 A comparison of different cost functions

9.5 Results for three dimensional graphs

With an extra degree of freedom it was found that it was more difficult to propose obstacles which obstructed the path. Tests on real problems, although these were not exhaustive, showed that traps such as that in figure 9.3(c) were very unlikely to occur.

The calculation time for Ch1 was increased by increasing the distance between S and G and also it was increased greatly by changing from two dimensions to three dimensions. The calculation time for Ch2 was not affected by the number of dimensions of the graph but it did increase with increasing the distance between S and G.

An example of a path is given in this section which illustrates the difficulty of representing paths and shapes in three dimensions. Figure 9.9 shows different layers through an obstacle which corresponded, in real space, to a sphere with radius of 300mm, positioned at coordinates (50,650,50) relative to the base of the robot.

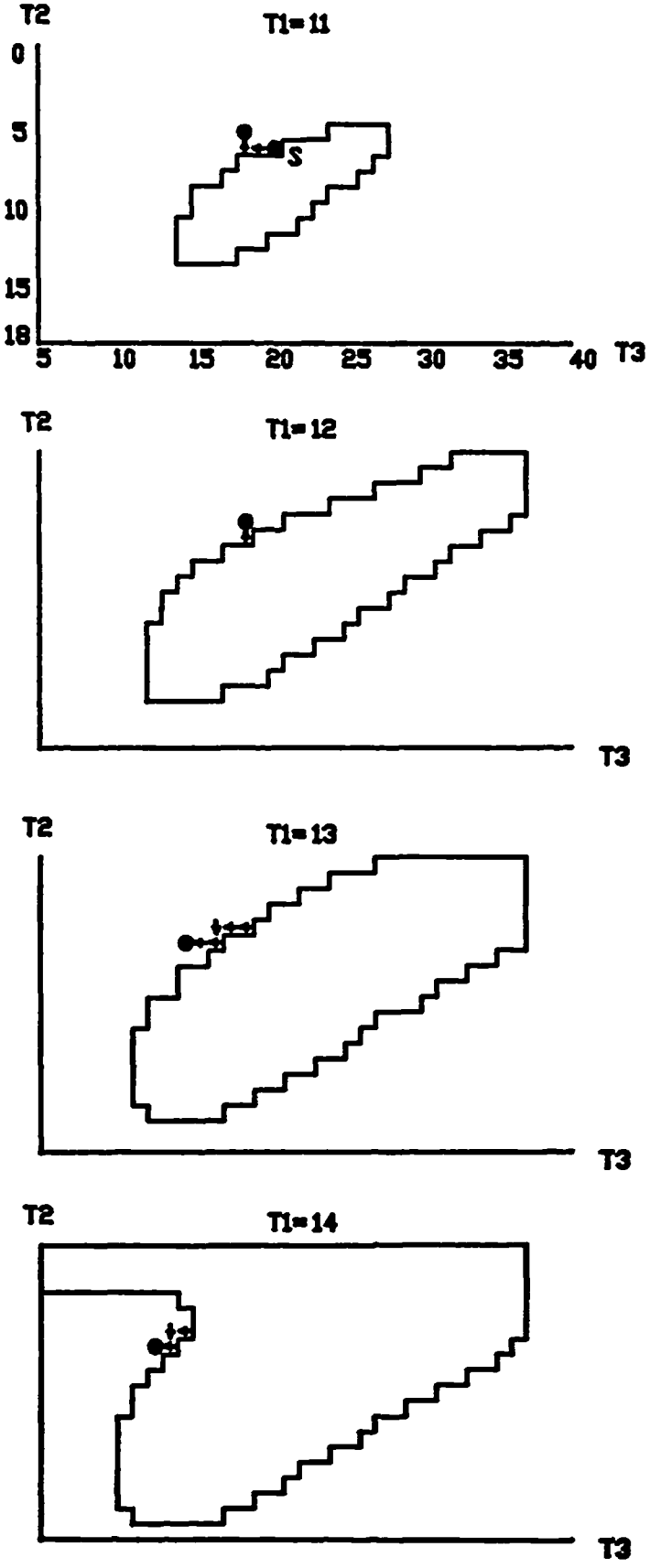


Figure 9.9(a) Path around an obstacle

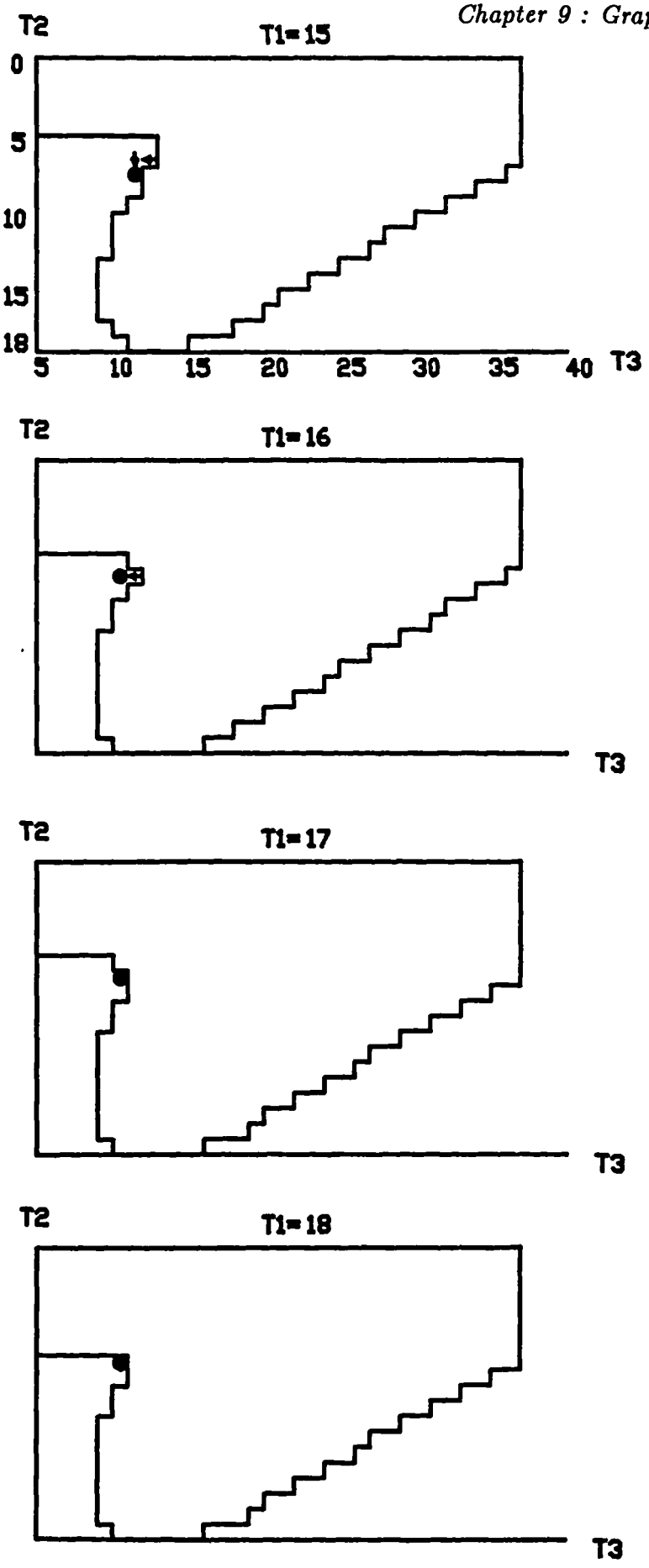


Figure 9.9(b) Path around an obstacle

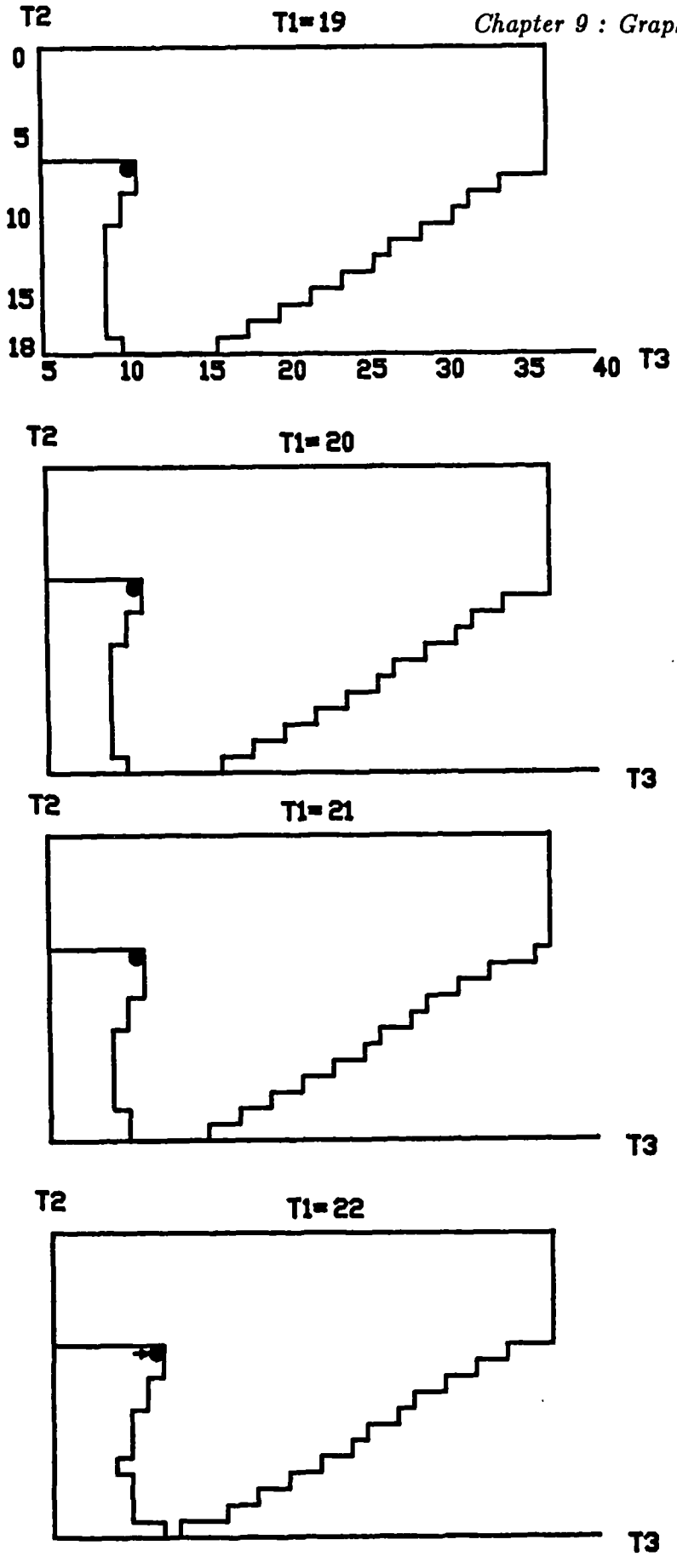


Figure 9.9(c) Path around an obstacle

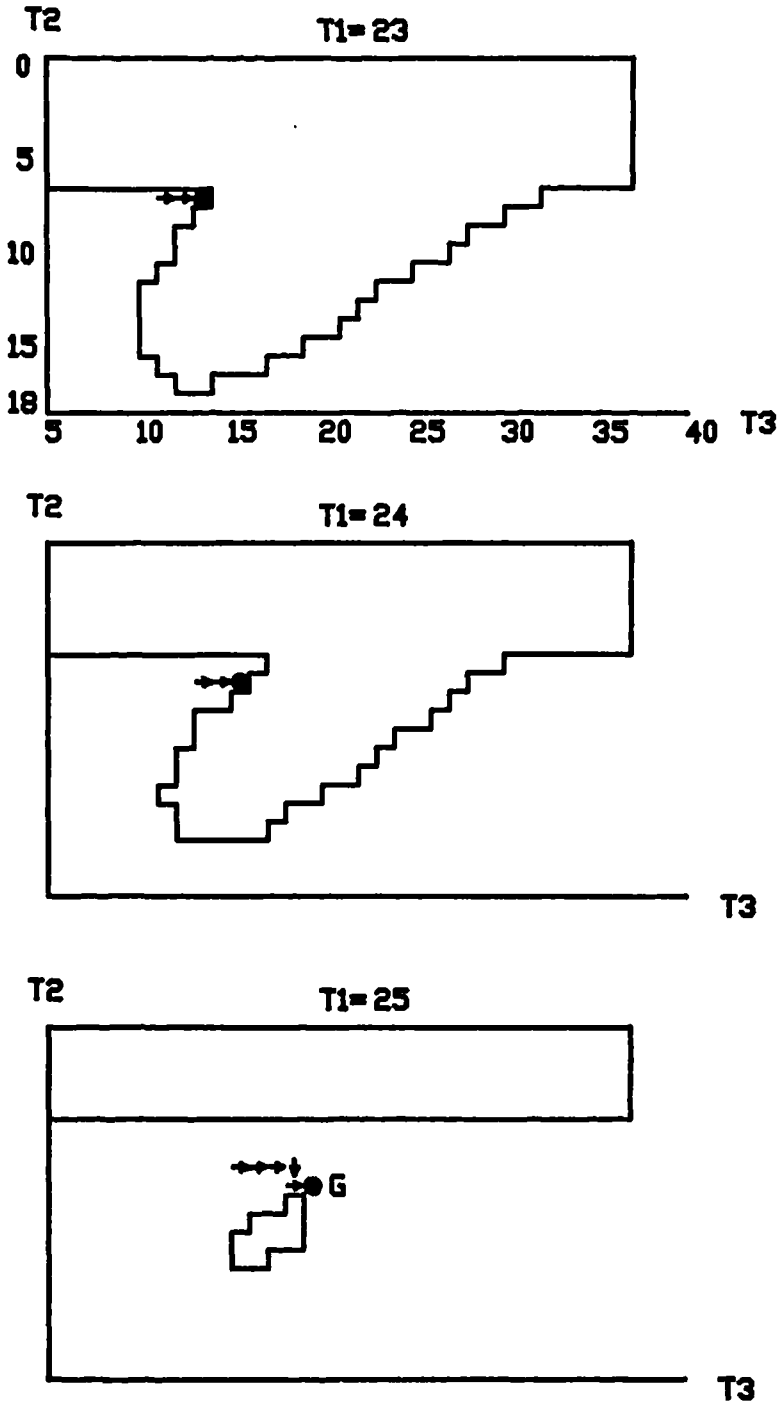


Figure 9.9(d) Path around an obstacle

A problem was proposed which was to move the robot from one side of the obstacle to the other. The path is denoted by arrows on the various layers of figure 9.9. It took approximately 20 seconds to calculate. The path moved close to the obstacle surface because by doing so the shortest path length between S and G was obtained.

Two separate obstacles are shown on the final layer (T1=25). The obstacle at the top was an obstacle due to the upper arm of the robot. The position of the upper arm depended only on T1 and T2, hence the obstacle existed for all T3 values and was rectangular in shape. The smaller obstacle was due to the gripper of the robot. The reason that there was no obstacle for the forearm was that the joints of the robot were overslung (see figure 8.1) and hence it was out of range.

The limiting factor on the resolution of the graph was the memory required to store the nodes. The maximum size of graph used was 19x35x19 = 12,635 nodes. Each node on the graph required two bytes of memory storage space.

9.6 Discussion and Conclusions

The graph searching method defined in this chapter provides good solutions to the difficult findpath problem. Moreover, the short computer calculation times required will enable real-time operation of the algorithm for industrial purposes. The important factors which provide this solution are :

(a) A graph containing large numbers of nodes, representing the collision free positions of the robot, each node havivg a few branches to its adjacent nodes.
(b) Cost functions for calculating the suitability of paths and for guiding the path searching algorithms.

9.6.1 The graph

The large numbers of nodes which can be used means that the accuracy of world model is high. This enables the solution to be 'good' in terms of short path lengths and finding solutions if they exist. However the number of nodes which can be used is restricted by

(a) The time available to calculate the nodes.
(b) The computer memory available to store the data, even though each node requires only two bytes of memory.

By restricting the number of branches between nodes the problem became solvable. However, the limiting of branches limited the directions in which the path could move. On the micro scale of figures 9.7(a) to (c) it may be seen that the path zigzaged in the correct direction but it was constrained by the directions of the branches from moving in the correct direction.

If the zigzags are small the macro movement of the robot will not be effected by the micro zigzag movements, in fact this is the way robots are programmed to move anyway. However if the zigzags are larger, then either a smoothing algorithm could be applied to the final path or the number of branches in the graph could be increased to offer more directions for the solution path.

9.6.2 The cost functions

Two different cost functions were used in the planning algorithm. The cost of a path at a particular node was defined as the cost of the path already found to that node, Cg, plus a cost, Ch, which was the hypothesised cost of the path from the node to the Goal.

Two functions of Ch were considered. For Ch1 the value of Ch was defined as the minimum possibly path length from the node to the Goal. For Ch=Ch1 the shortest path possible was always produced. However, in order to calculate

the shortest distance path every possible path which could provide the shortest path had to be considered. Consequently the algorithm took a long time for large graphs.

Ch2 was defined as the square of the minimum distance to the goal. This function ensured that nodes which were further from the goal had disproportionately large proposed costs and hence the algorithm favoured nodes which were closest to the Goal. The result was that the path finding effort was pulled to the goal at the expense of alternative paths further away.

From the results of calculation times for the different functions of Ch it was concluded that different environments and different requirements for path efficiency would require different functions of Ch. Where the path cost is of prime importance Ch1 should be used, otherwise Ch2 will be better as the calculation time is generally much shorter.

9.6.3 Application to the real problem

The algorithm has been successfully used to control a Smart Arms robot at the University of Durham to plan the mid phase motion of the robot. The robot was modelled with a three dimensional graph, the gripper and part being accounted for as spheres which enclosed all possible positions of the gripper and part.

In order to carry out approach path planning automatically it is possible to use a graph with the same number of dimensions as there are degrees of freedom for the robot. However, these graphs should be restricted to the small areas of workspace where they are needed, as they require large numbers of nodes.

9.7 Further Work

The amount of memory required by the graph may be reduced by using a

variable size of unit in the space transformation algorithm of chapter 8. This graph would also speed up the pathfinding algorithm as the same length path would generally require fewer nodes.

The effect of different functions of Cg and Ch on the path produced could be investigated further. Particularly, if for the sake of safety the path should not move to close to obstacles unless necessary. A cost function could be used which produces high costs for paths which go too close to obstacles.

CHAPTER 10

CONCLUSIONS

10.1 The world model

The use of spheres to model real obstacles makes it possible to calculate collision free paths in real-time using a small micro-computer. All the other world models considered require greater computational effort to solve the pathfind problem for a robot. Hence solutions to practical problems in real-time would be more difficult or impossible to achieve without using the sphere model.

The use of this modelling technique implied a compromise in the accuracy of representation of the work space model. Except in very tightly constrained operating environments this compromise was shown to be acceptable, indeed the sub-optimal paths calculated tended to move further away from obstacles, and to increase safety margins.

The robot arm was modelled as two connected cylinders with hemi-spherical ends. The advantages of this representation were that the cylinders modelled the robot arm efficiently, and the intersection calculations between the robot arm and the obstacle spheres were very fast. The workspace which was 'lost' by this representation was around the centres of the faces of the robot links. The corners of the links were at the limits of the cylinder model.

Figure 10.1 shows how the inaccuracies of the world model can affect shortest distance paths. (a) and (b) are the start and goal positions of a cross section of the robot arm, (c) is an obstacle. The rectangular shapes represent the real world sizes of the obstacle and the robot arm. These shapes are enclosed by circles which represent the surfaces of the arm cylinder of the sphere. It may be

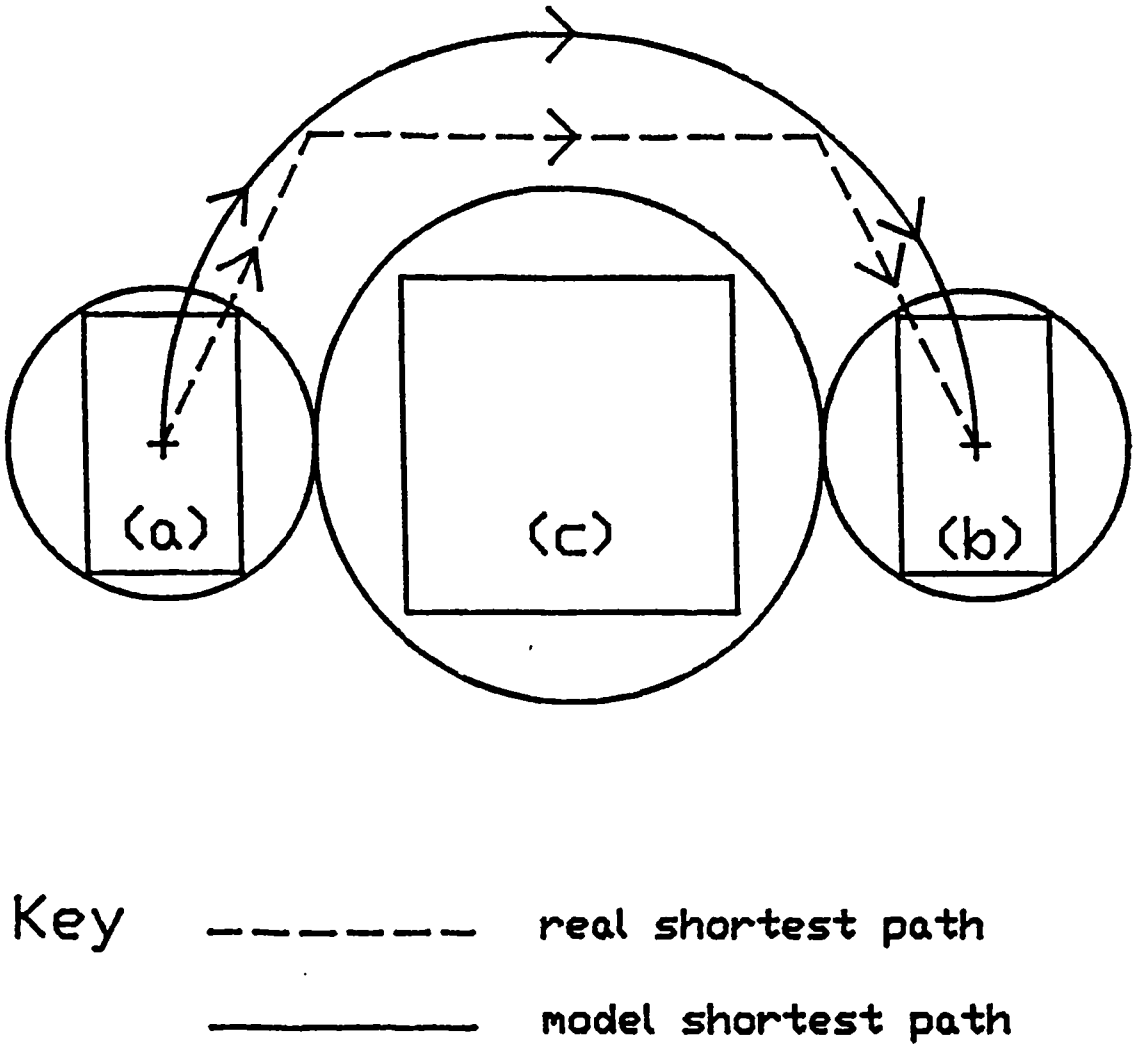


Figure 10.1 A comparison of the model shortest path with the real shortest path

seen that the minimum distance path for the model is approximately 7 percent greater than that for the real world. However the path is much smoother.

It could be argued that the minimum distance path produced using the model is better than that for the real objects when considering factors such as energy used, time of execution and robot wear, because the accelerations required to produce the path are lower as there are no 'corners' in the path.

As the sphere model is such a simple model it would be possible to convert data from other models such as solid models. This would provide automatic planning facilities for systems such as GRASP which rely upon the programming skills of the GRASP operator.

Automatic methods for world model measurement such as vision or ultrasonic techniques could be used to provide data for sphere model generation. In some cases sphere models would provide a more realistic interpretation of vision or ultrasonic data, as the general size, the distance and the direction of obstacles are easily determined from vision or ultrasonic sensors where as the edges, faces and corners required for polyhedral models are more difficult to determine.

One disadvantage of the sphere model was that most of the objects which were modelled were not made up of spherical shapes. Thus when diagrams of the spherical model approximations were produced it was found that the diagrams were difficult to interpret. For instance when the spherical model in figure 4.7 was drawn it was thought helpful to superimpose the real obstacles to help visualisation.

10.2 Local pathfinding methods

The local pathfinding method used to calculate trajectories for the robot forearm in section 6.5 is based on a few simple rules. This method works adequately with a limited set of path finding problems. Solutions are achieved quickly and reliably for many robot environments. More complex problems require more heuristic methods to solve situations which can not be solved by the current algorithms.

An extension of this method could be based on a different programming language such as LISP. This language is designed for artificial intelligence problems and can deal with finding solutions given a set of data and a series of rules. This

the last recorded holder, Joseph Naylor. It remained in Basire's possession until his death in 1676, when it passed to his successor in the Seventh Stall, John Morton. He died in 1722, and at his request, the volume was given to the Dean and Chapter Library where it now remains. While he had possession of the book, Basire studied it carefully, and certain marginal notes, as well as some of the memoranda, are in his handwriting. Hutchinson, in his History of Durham, mentions an attempt made by Basire in 1665 to procure an 'exemplification' of the statutes for the Dean and Chapter. There had been some dispute between them and the Bishop about his rights of visitation in the Cathedral, and Basire had been commissioned to try and clarify the position:

> 1665. Sept. 12. At a meeting between Bishop Cosin and the Dean and Chapter, it was agreed amongst other things - 'That an exemplification of the Statutes of the Church should be procured from the Rolls on [sic] the Tower, or any of the King's Courts, within a twelve month after it hath pleased God to cease the prevalent pestilence.'

The following is Dr Basire's answer to the Chapter, and literally transcribed from the original:

> I took the pains to cause a search to be made in the rolls, but found nothing. The like I did with Mr Dugdale, when he was searching the records of the dioceses, and the records of St. Paul's Church, and to encourage him, gave him a gratuity from the Dean and Chapter, but sped [sic] no better. What may be found in the Tower I know not, having had neither the use nor opportunity to search there; Mr William Prynn (no great friend to cathedrals) being keeper of these records. (1)

The 1554 statutes set out in detail how the Cathedral was to be organised, what endowments were to be provided for its upkeep and the maintenance of the Dean and Chapter, and what were to be the responsibilities of them and their

---

(1) W.Hutchinson, The History of Durham, vol.ii, p.181. cf. Cosin's Correspondence, ii, p.139. After the Restoration, Charles II rewarded Prynne's staunch royalism, with the appointment as Keeper of the Records of the Tower. (W.M. Lamont, Marginal Prynne (London, 1963), p.206.)

In certain circumstances the calculation time required to solve the path planning problem could not be predicted. However upper limits could be placed on the calculation time to ensure that the program did not go into an infinite loop, but if the upper limit of the calculation time was reached then no path was found.

In practice it was found that many environments could be used such that solutions were always found to the path planning problem and that the calculation time was within the limits of the robot execution time.

An example of an environment for which the system performed satisfactorally is shown in figure 4.7. In this example the robot moved 5 parts at random to different places such as areas on the base and to the tops of the boxes.

For a typical task in the above environment, such as to pick up a part and move it to a different position the robot trajectory took 30 seconds and the calculation time was 25 seconds.

In certain environments the path planning computer always produced satisfactory paths in 'real time'. However there were certain situations where this method became 'trapped' and no path was found where a path existed or that paths were found after calculation times greater than robot execution times. Given the unlimited complexity of paths there will always be these problems no matter what path planning algorithms are used.

The advantage of this method is that the method uses simple rules to solve a problem which is difficult to analyse. The method for planning the upper arm path is efficient in terms of time and paths produced. The method for planning the forearm path uses simple heuristic rules to avoid obstacles.

The disadvantages of the method are that it does not always find a path where one exists. The fact that the forearm and upper arm are planned separately means that many possible paths are not considered and hence the paths produced

are not necessarily the shortest paths. Also the program is closely tied to one type of robot. Some of the program code would need changing to accommodate the kinematic chain of a different robot.

10.5 The second method

10.5.1 Obstacle transformations

A high cost in computer time was incurred to transform the real obstacles into joint space. For the current application this is acceptable if the environment is static for several pathfinding operations.

The factors which were found to affect the computer time for the obstacle transformations were :-

(a) The total workspace of the obstacles.
(b) The number of obstacle spheres.
(c) The size of the robot workspace.
(d) The resolution of the graph for searching.

There are two ways of transforming the workspace: transform the obstacles or transform the empty space. The first method was adopted as it was felt that the volume of obstacles would generally be much less than the volume of empty space. As a result of this approach the total workspace of the obstacles and the number of obstacle spheres present both affected the transformation time.

The calculation time was also affected by the size of the units which made up the transformed graph. The smaller the unit size the greater the resolution of the graph. For the results of chapter 8 a graph of 21,312 units was used. The size of this graph was limited by the computer memory available and the calculation time which was practical.

10.5.2 Graph searching

The graph searching method used in method 2 provides good solutions to the difficult pathfind problem. The important factors which provided this solution were :

(a) A graph containing a large number of nodes.
(b) The adaption of cost functions in the graph searching algorithm.

The large numbers of nodes which can be used means that the accuracy of world models is high. This provides a 'good' solution in terms of short path length and finding solutions if they exist.

The nodes form a grid of points in joint space so that the coordinates of any particular node can be represented by coordinates (i1, i2, i3, ... in), where i1 to in are integers. Branches exist between all adjacent 'clear' nodes. Thus the path between S and G is a connected set of nodes on the transformed graph beginning with S and ending with G. The cost of the path is defined as the total number of branches in the path.

The restriction of the number of branches in the graph reduces the graph complexity and enables large numbers of nodes to be used. However this limits the directions in which the path can move. On the micro scale it was seen that the path zigzagged in the correct direction. For the test robot it was found that the zigzags were small and the macro movement of the robot was not affected by the micro zigzag movements, in fact this was how the robot's movements were programmed by its own controller. However if the branch length of the graph is too long for the smooth movement of robots in the future the path may be straightened by post processing.

Two different cost functions were investigated for the planning algorithm. The cost of a path at a particular node was defined as the cost of the path

already found to that node, Cg, plus a cost, Ch, which was the hypothesised cost of the path from the node to the Goal. By changing Ch the properties of the algorithm changed. This is very useful as it means that Ch can be set to provide either fast solutions or optimal solutions.

In order to carry out approach path planning automatically it is possible to use a graph with the same number of dimensions as there are degrees of freedom for the robot. However, these graphs should be restricted to the small areas of workspace where they are required as they require large numbers of nodes.

10.6 Comparison of methods

It can be said that the performances of the methods were affected by different factors. Providing no 'traps' were encountered then the speed of the first method was affected by the number of potential collisions avoided. The second method's speed was dependent upon the time taken to calculate the graph for searching (dependent upon number and size of obstacles) and the time taken to search the graph depended on how far apart the start and goal configurations were and how complex the solutions found were.

For certain problems the first method became irrevocably stuck and the second method took extra time to find the solution. Although this is an unsatisfactory conclusion, it is impossible to develop a pathfinding method which will solve every problem in a limited time. However in favour of the second method was the fact that if a solution existed it found it eventually.

The quality of solution produced depended firstly on the quality of the representation of the real problem in the problem solving domain. This is true of all path finding methods.

The set of solutions of method 1 was limited because planning for the forearm and upper arm was carried out separately. Having fixed the trajectory of the upper arm the obstacles could only be avoided by altering the forearm trajectory. This meant that sub-optimal solutions were produced.

The quality of solution for the second method was affected by the size of the units in the graph of possible configurations. Each node in the graph represented a set of configurations for which the robot was free of collisions. For the tests carried out, one node on the graph represented one configuration +/- 2.5 degrees of movement in each of the three degrees of freedom of the robot.

Several different cost functions were investigated for the second method. The cost function chosen had a great effect on the performance of the graph searching algorithm. One cost function investigated produced minimum cost paths. Another cost function calculated its paths more quickly (in general) but the solutions were not necessarily the minimum cost solutions.

The second method can be adapted to a wide range of problems. The graph searching method may be applied to systems with any number of degrees of freedom. The cost functions may be changed to suit the need either for a fast or an optimum solution. The cost function can also be changed in order to optimise different variables such as distance travelled for the payload or the safety of the path.

The graph calculation method (Espace) may also be adapted for different types of robot or even for different types of model, ie to a polyhedral model. Although other types of world model require greater computational effort, for off-line programming this would not be a problem.

10.7 Trajectories to suit calculations

In section 5.4 it was found that the 'natural' trajectory for the robot between two configurations did not produce the shortest distance path for a point on the upper arm. It is the author's opinion that the interpolation of joint angles is the method of movement used for most revolute robots, and that savings in the time and energy required for robot movements could be made if the interpolation method were changed.

10.8 Design of robot

There were certain cables on the test robot which were not included in the robot model. This was justified by the following :-

(a) Many robots do not have this protruding cabling.

(b) The test robot could have been redesigned such that the cables were inside the robot links.

(c) The time required to model the cables would have made the completion of this research very difficult.

Another difficulty was created by the particular design of the test robot. This was that the robot's links were overslung. This problem was overcome (section 4.4). However the final solution was more complex and slower because of this difficulty.

The idea of an 'ideal robot design' was discussed in section 4.8. In practice a compromise will have to be made between other robot design requirements and those of automatic programming systems.

10.9 Further work

A real-time path planning system may only be used when information on the required task is available in real-time. This information is most readily acquired from sensory devices such as vision systems. A vision system can recognise a part and hence infer the appropriate operation required and it can determine the location and orientation of the part. By adding a vision system to the automatic planning system at the University of Durham, some real applications for this research may be investigated.

While this work has been carried out technology has become more and more sophisticated. Understandably after three years the equipment is now out of date. A new implementation of any of the techniques described in this thesis on more modern technology, an IBM PC for example, would provide benefits in performance.

The very exciting developments in multi-processor computers could be applied to path planning. Particularly for the transformations of obstacles, where the task of splitting the work load into bits for different processors would not be difficult. This would provide great reductions in calculation time.

This section would not be complete without stressing that software development is never finished. There are always things which can be done to improve the programs. The software development work has concentrated mainly on getting the software to operate and a proportionately small amount of time has been devoted to optimising the code.

10.9.1 The first method

The first method would be improved by providing extra heuristic rules which would unstick the algorithm in certain situations. In particular if the forearm

becomes stuck a facility for replanning the upper arm trajectory would be very useful.

10.9.2 The second method

The amount of memory required by the graph may be reduced by using a variable size of unit in the space transformation algorithm of chapter 8. This graph would also speed up the pathfinding algorithm as the same length path would generally require fewer nodes.

The effect of using different functions for Cg and Ch on the path produced could be investigated further. These cost functions may be adapted to suit many different path requirements. For example it may be that paths which move close to obstacles are undesirable from a safety point of view. If this is the case then a cost function may be developed which produces high costs for paths which pass close to obstacles. This would cause the path planning algorithm to favour safer paths.

It may be possible to speed up the graph searching algorithm by planning in a hierarchy of levels.

The approach paths required for paths used in this research were planned by the human operator. This was justified by saying that the geometry of part and fixture ensured that generally one approach path would be obvious to the operator. An example given was that of putting a part into the chuck of a lathe. In this case the approach path is clearly to orient the part along the lathe axis and then move along that axis until the part is within the lathe jaws. However there is a need for the automatic planning of approach paths. A flexible assembly system for instance might be supplied with the details of parts and details of the final assembly. In this situation a method of planning the approach path would be required.

As approach path planning requires a detailed knowledge of the parts geometry a solid modelling system would be an important requirement. In this case the second path planning method could be adapted to deal with the fine motions of approach path planning.

This research has relied upon the use of a robot to verify the computer programs. There is however a simpler and more versatile method of doing this and that is to use a graphics simulation of the world model and the robot's movement.

The addition of such a facility is seen as vital to any future development of this work. However the use of real robots in a real situations will still be the final test of the success of a particular program.

As the second method is independent of the world model used it may be a useful exercise to try using a polyhedral model for similar problems as those described here. This would provide a useful comparison between sphere and polyhedral models. It might also pave the way towards integrating a path planning system with an automatic programming system such as GRASP.

Some recent work of Kant (1986) has investigated the possibility of planning paths in a time varying environment. Kant has decomposed this problem into 2 sub-problems, the path planning problem and the velocity planning problem. Although this might be the simplest solution it is not necessarily the best and the paths could turn out to be very inefficient. An alternative solution following on from this work might be to consider time as an extra dimension in the graph searching method and solve the problem in the existing way.

10.10 A look into the future

At present, ninety percent of robot usage in industry is on what one might

call 'open loop'. This means that robot manipulator arms go about their pre-programmed routines with virtually no feedback from the environment in which they operate. Even those robots which do benefit from feedback have a very limited range of responses to their changing environment. This way of thinking has had an immense affect on the potentials and limitations of what can be achieved in industries through the use of industrial robots.

In humans the most important sensory feedback which we possess is vision. For robots too this is the most important form of feedback, although touch is also vital for many applications. Already vision has been used to correct the programmed positions of robots in real-time applications (El Zorkany 1984). Future generations of robots will make more and more use of vision data.

The primary objective of robot vision is to allow the robot program to branch depending on what is seen by the robot eye. For example consider a situation where a vision system surveys objects arriving at a pick up point. The vision system would decode the shape of each object as it arrived and feed information to the robot computer about the part's type, its orientation and its position. The robot computer would then generate a path which would pick up the part and carry out the appropriate operations. In this example it may be noted that for each new part a different path may be required.

A system such as this allows much more flexibility in the production environment. Parts do not have to be accurately positioned in the robot's workspace. Parts may even be picked out of bins.

A whole new range of tasks can be achieved if both vision and automatic path planning capabilities are incorporated into a new range of robots. Two areas where these techniques can be useful are, the agricultural industry and automatically guided vehicle (AGV) technology.

In the agricultural industry robots could be used for applications such as fruit and vegetable harvesting. Here the vision system would provide a world model such as the positions of apples and branches in an apple tree. The robot computer would then move the robot between the branches to pick the apples. Avoiding collisions with the branches is clearly important to avoid damaging the robot and the tree.

AGVs are beginning to be used in conjunction with robots. The robot sits on top of the AGV and manipulates parts to be transported. This may be a difficult operation as accurately positioning the AGVs is a difficult task and warehouse environments are notoriously cluttered with obstacles. Here vision systems and automatic path planning will provide great benefits in increasing the flexibility of AGV/robot devices.

REFERENCES

AHUJA, N., CHIEN, R.T., YEN, R., BRIDWELL, N. Interference detection and collision avoidance amoung three dimensional objects. 1st Annual National Conference on Artificial Intelligence, Stanford University, Stanford, C.A., August 1980. pages 44-48.

ALBUS J.S., BARBERA A.J., FITZGERALD M.L. Programming a hierarchical robot control system. Proceedings of the 12th International Symposium on Industrial Robots, Paris, France. 9-11th June 1982.

ALBUS J.S., Mc LEAN C.R., BARBERA A.J., FITZGERALD M.L. Hierarchical control for robots in an automated factory. 13th International Symposium on industrial robots and Robots 7. April 17-21 1983. Chicago Illinois. Pages 13.29-43.

ARAI Y., HATA S., IMAKUBO T., KIKUCHI K. Production control system of microcomputers hierarchical structure for FMS. 1st International conference on Flexible Manufacturing Systems, Brighton, 20-22 October 1982. IFS Publications Ltd. ISBN 0-903608-30-8, pages 3259-68.

BALDING N.W., PREECE C. Real-time collision-free path calculation for robots. Proceedings of the Institution of Mechanical Engineers. Conference on UK research in advanced manufacture. London, 10-11th December 1986, pages 61-67.

BALL A. Designers guide to shapes. Cadcam International. March 1983, pages 32-34 and October 1983, pages 42-44.

BONNEY M.C. Off-line programming and robot safety. Presented to the 4th SERC Vacation School in robotic technology. 9-14th December 1984. Loughborough University.

BONNEY M.C., MOSER J., YONG Y.F. Evaluation and use of a graphical robot simulator - a case study from ITT/AMTC using GRASP. International conference on simulation in manufacturing, Stratford Upon Avon, United Kingdom, 1985.

BOURNE, D.A., FUSSELL, P.S. Designing programming languages for manufacturing cells. Proceedings of Electro 82 – 14, IEEE June 1982, pages 23/3 1-9.

BOYSE J.W. Interference detection amoung solids and surfaces. Communications of the ACM, January 1979, Volume 22, Number 1, pages 3-9.

BRAID I.C. Designing with volumes. PhD thesis, Cambridge University, England, 1973.

BRAID I.C. The synthesis of solids bounded by many faces. Communications of the ACM, volume 18, page 209-221, (1975).

BROOKS R.A. Planning collision free motions for pick-and-place operations. The International Journal of Robotics Research, volume 2, number 4, Winter 1983(a), pages 19-44.

BROOKS R.A. Solving the find-path problem by good representation of free space. IEEE Transactions on Systems, Man, and Cybernetics, Volume SMC-13, number 3, March 1983(b), pages 190-197.

CAMERON S. The clash detection problem. Department of Artificial Intelligence. University of Edinburgh. DAI working paper number 126. September 1982.

CASSINIS R. Hierarchical control of integrated manufacturing systems. 13th International symposium on industrial robots and Robots 7. April 17-21, 1983. Chicago, Illinois. Pages 12-9 to 12-20.

CHIEN R.T., ZHANG L., ZHANG B. Planning collision-free paths for robotic arm among obstacles. IEEE Transactions on Pattern Analysis and Machine Intelligence, volume 6, number 1, January 1984, pages 91-6.

CWIAKALA M., LEE T.W. Generation and evaluation of a maniputator workspace based on optimum path search. Transactions of the ASME, Journal of Mechanisms, Transmissions, and Automation in Design. June 1985, volume 107, pages 245-255.

DE PENNINGTON A., BLOOR S., BALILA M. Geometric modelling : a contribution towards intelligent robots. Conference proceedings of the 13th international symposium on industrial robots and robots 7. April 17-21st 1983. Chicago. North Holland Publishing Company, pages 7.35-54.

DILLMANN R. A graphical emulation system for robot design and program testing. Conference proceedings of the 13th International Symposium on Industrial Robots and Robots 7. April 17-21st 1983, Chicago. North Holland Publishing Company, pages 7.1 to 7.15.

EL-ZORKANY H.I. Automatic location correction in off-line programming of industrial robots. Proceedings of the 14th International Symposium on Industrial Robots. Gothenburg, Sweden (IFS) 1984, pages 335-346.

FIKES E., HART P.E., NILSSON N.J. Learning and executing generalised robot plans. Artificial Intelligence 3 (1972) pages 251- 288.

FUSSELL P., WRIGHT P.K., BOURNE D. A design of a controller as a component of a robotic manufacturing system. 13th International symposium on industrial robots and robots 7, April 17th to 21st 1983, Chicago, Illinois, pages 16.48-62.

GASPART P., BAUDOT W., Flexible and decentralized control of a machining shop. 1st International conference on flexible manufacturing systems, Brighton, 20-22nd October 1982, IFS (Publications) Ltd. ISBN 0-903608-30-8, pages 379-388.

GILBERT E.G., JOHNSON D.W. Distance functions and their application to robot path planning in the presence of obstacles. IEEE Journal of Robotics and Automation, volume RA-1, number 1, March 1985, pages 21-30.

GOUZENES L. Strategies for solving collision-free trajectories problems for mobile manipulator robots. International Journal of Robotics Research, volume 3, number 4, pages 51-65.

HANSEN J.A., GUPTA K.C., KAZEROUNIAN S.M. Generation and evaluation of the workspace of a manipulator. International Journal of Robotics Research. Volume 2, number 3, Fall 1983, pages 22-31.

HART P.E., NILSSON N.J., RAPHAEL B. A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions of Systems Science and Cybernetics, Volume SSC-4, number 2, July 1968, pages 100-107.

HART P.E., NILSSON N.J., FIKES R.E. Learning and executing generalised robot plans. Artificial Intelligence 3, 1972, pages 251-288.

HOPCROFT J.E., SCHWARTZ J.T., SHARIR M. Efficient detection of intersections amoung spheres. International Journal of Robotics Research. Volume 2, number 4, winter 1983, pages 77-80.

IRVINE J. CAD puts robots in reach. CADCAM International. December 1986, pages 26-29. EMAP Buisness and Computer Publications Ltd. London.

KANT K., ZUCKER S.W. Toward efficient trajectory planning : the path-velocity decomposition. The International Journal of Robotics Research, volume 5, number 3, Fall 1986, pages 72-89.

KHATIB O. Real-time obstacle avoidance for manipulators and mobile robots. The International Journal of Robotics Research. Volume 5, number 1, spring 1986, pages 90-98.

KIRSCHBROWN R.H., DORF R.C. 'KARMA' - A Knowledge-based robot manipulation system. The International Journal of Robotics Research. Volume 1, March 1985, pages 3-12.

*

LEE T.W., YANG D.C.H. On the evaluation of a maniputator workspace. Transactions of the ASME, Journal of Mechanisms, Transmissions, and Automation in Design. March 1983, volume 105, pages 70-76.

LOZANO-PEREZ T. Automatic planning of manipulator transfer movements. IEEE Transactions on Systems, Man and Cybernetics, Volume SMC-11, number 10, October 1981, pages 681-698.

LOZANO-PEREZ T. Spatial planning : a configuration space approach. IEEE Transactions on Computers. Volume 32, number 2, Feburary 1983, pages 108-130.

LOZANO-PEREZ T., WESLEY M.A. An algorithm for planning collision-free paths amoung polyhedral obstacles. Communications of the ACM. Volume 22, October 1979, number 10, pages 560-570.

LOZANO-PEREZ T., MASON M.T., TAYLOR R.H. Automatic synthesis of fine-motion strategies for robots. International Journal of Robotics Research. Volume 3, number 1, pages 3-24. Spring 1984.

* KREYSZIG E. Advanced Engineering Mathematics. Fourth Edition John Wiley and Sons, New York.

LUH J.Y.S., CAMPBELL C.E. Minimum distance collision-free path planning for industrial robots with a prismatic joint. IEEE Transactions on Automatic Control, volume AC-29, Number 8, August 1984, pages 675-680.

PAUL R.P. Robot manipulators : mathematics, programming and control. Cambridge, MA : Mass. Inst. Tech., 1981.

PIEPER D.L. The kinematics of manipulators under computer control. PhD Thesis, Stanford University, October 1968.

RED W.E., HUNG-VIET TRUONG-CAO. Configuration maps for robot path planning in two dimensions. Transactions of the ASME, volume 107, December 1985, pages 292-298.

REQUICHA A.A.G. Representations for Rigid Solids : Theory, Methods and Systems. Computing Surveys. Volume 12, number 4, December 1980, pages 437-463.

REQUICHA A.A.G. Constructive solid geometry. Rochester University, N.Y. Production Automation Project, November 1977. NSF/RA-770476.

SACERDOTI E.D. Planning in a hierarchy of abstraction spaces. Artificial Intelligence 5 (1974), pages 115-135.

SAHAR G., HOLLERBACH J.M. Planning of minimum time trajectories for robot arms. The International Journal of Robotics Research, volume 5, number 3, Fall 1986, pages 90-100.

SCHWARTZ J.T., SHARIR M. On the 'piano movers' problem II. General Techniques for computing topological properties of real algebraic manifolds. Advances in Applied Mathematics. Volume 4, number 3, September 1983, pages 298-351.

SMITH R.C., NITZAN D. A modular programmable assembly station. Conference proceedings of the 13th International Symposium on Industrial Robots and Robots 7. April 17-21 1983, Chicago. North Holland Publishing Company pages 5.53-75.

UDUPA S. Collision detection and avoidance in computer controlled manipulators. PhD Thesis, California Institute of Technology, 1977(a).

UDUPA S. Collision detection and avoidance in computer controlled manipulators. Proceedings of the 5th International Joint Conference on Artificial Intelligence 5. 1977(b), pages 737-748.

VUKOBRATOVIC M., KIRKCANSKI M. A method for optimal synthesis of manipulation robot trajectories. Transactions of the ASME, Journal of Dynamic Systems, Measurement, and Control. Volume 104, June 1982, pages 188-193.

YANG D.C.H., LEE T.W. On the workspace of mechanical manipulators. Transactions of the ASME, Journal of Mechanisms, Transmissions and Automation in Design. Volume 105, March 1983, pages 62-69.

APPENDIX A

CALCULATING THE VOLUMES OF SPHERE MODELS WHICH APPROXIMATE A UNIT CUBE

In order to quantify the difference between the model of spheres and the real objects the volumes of the spheres and real objects may be compared.

It should be noted that the volume of the real objects must be completely contained by the spheres for safety reasons. Therefore the model will always have a larger volume than the real objects.

To make the comparison a unit cube was modelled by increasing numbers of similar spheres. This appendix shows how the volumes of these models of spheres were calculated.

(i) The volume of one sphere surrounding a unit cube.

The radius of the sphere = distance from the cube centre to one corner of the cube.

$$R_1 = \sqrt{\frac{1}{2}^2 + \frac{1}{2}^2 + \frac{1}{2}^2}$$

Therefore $R_1 = \frac{\sqrt{3}}{2}$

The volume of a sphere is given by $V = \frac{4}{3}\Pi r^3$

Therefore $V_1 = \frac{4}{3}\Pi\left(\frac{\sqrt{3}}{2}\right)^3$

This gives $V_1 = 2.72070$

(ii) The volume of 2 spheres modelling a unit cube.

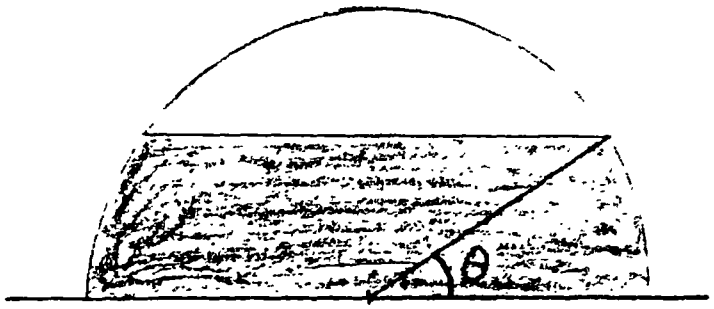
The cube may be defined such that one corner of the cube is at the origin and the opposite corner is at (1,1,1). The edges of the cube are parallel to the x,

y, or z axes.

The centres of spheres are at coordinates $(\frac{1}{2}, \frac{1}{2}, \frac{1}{4})$ and $(\frac{1}{2}, \frac{1}{2}, \frac{3}{4})$.

$$R_2 = \sqrt{\frac{1}{2}^2 + \frac{1}{2}^2 + \frac{1}{4}^2} \quad = \quad \frac{9}{16} \quad = \quad \frac{3}{4}$$

However the spheres intersect. But the volume of part of a sphere may be found as follows :-



The volume of the shaded region of this hemisphere is given by :

$$V_h = \Pi r^3 \int_0^{\Theta} cos^3\Theta d\Theta$$

but $cos^2\Theta = 1 - sin^2\Theta$

$$V_h = \Pi r^3 \left[\int_0^{\Theta} cos\Theta d\Theta - \int_0^{\Theta} cos\Theta sin^2\Theta d\Theta \right]$$

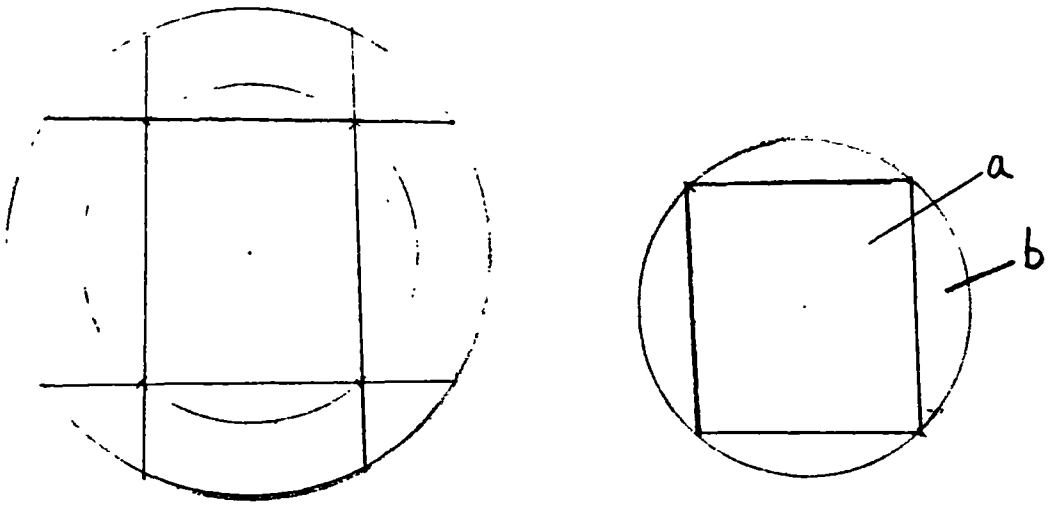
$$V_h = \Pi r^3 \quad \left[sin\Theta - \frac{sin^3\Theta}{3} \right]_0^{\Theta}$$

In this example the volume of model V_2 is the sum of one sphere plus two times the above integration, where

$$sin\Theta = \frac{\frac{1}{4}}{\frac{3}{4}} = \frac{1}{3}$$

(iii) The volume of 8 spheres modelling a unit cube

If a sphere is cut into segments such that each cut passes along the face of a cube which is just contained by the sphere then two types of segments are formed, a and b.



The sphere is made up of a cube (c) at the centre + 6a + 12b.

The volume V_t of a hemisphere - (a+4b) for a sphere containing the unit cube may be found as follows:

$$V_t = \Pi r^3 \left[sin\Theta - \frac{sin^3\Theta}{3} \right]_0^{\Theta}$$

Where $r = \frac{\sqrt{3}}{2}$ and

$$sin\Theta = \frac{\frac{1}{2}}{\sqrt{\frac{1}{2}^2 + \frac{1}{2}^2 + \frac{1}{2}^2}} = \frac{1}{\sqrt{3}}$$

Therefore

$$V_t = \Pi \left(\frac{\sqrt{3}}{2}\right)^3 \left[\frac{1}{\sqrt{3}} - \frac{1}{3}\left(\frac{1}{\sqrt{3}}\right)^3 \right]$$

This simplifies to

$$V_t = \frac{\Pi}{3}$$

Volume of hemisphere $= \frac{2}{3}\Pi r^3$

substituting in the value of r gives :

$$V_h = \Pi\frac{\sqrt{3}}{4}$$

The difference between these two volumes is the volume of a + 4b

$$= 0.31315$$

Total sphere is made up of c + 6a + 12b.

$$= \frac{\sqrt{3}}{2}\Pi = 2.72070$$

$$but \qquad c = 1$$

$$therefore \qquad 6a + 12b = 1.72070$$

$$simplifieing \qquad a + 2b = 0.28678$$

$$but \qquad a + 4b = 0.31315$$

$$therefore \qquad 2b = 0.026370$$

so a = 0.26041 and b = 0.013185

For 8 spheres modelling the unit cube the radius of each is :

$$R_8 = \sqrt{\frac{1^2}{4} + \frac{1^2}{4} + \frac{1^2}{4}} = \frac{\sqrt{3}}{4}$$

Model contains $8c_8 + 24a_8 + 24b_8$

$$c_8 = \frac{1}{8}$$

$$a_8 = \frac{1}{8} \times 0.26041 = 0.032551$$

$$b_8 = \frac{1}{8} \times 0.013185 = 0.0016481$$

Volume of model = 1 + 0.78122 + 0.039555

$$V_8 = 1.82078$$

(iv) The volume of 27 spheres modelling a unit cube.

Total volume is made up of 27c + 54a + 36b

Radius of each sphere is

$$a_{27} = \frac{1}{27} \times 0.26041 = 0.00964487$$

$$b_{27} = \frac{1}{27} \times 0.013185 = 0.000488318$$

$$V_{27} = 1.5384$$

(v) The volume of 64 spheres modelling a unit cube

Total volume is made up of 64c + 96a + 48b

Using the previous method.

$$V_{64} = 1.4005$$

APPENDIX B

TO FIND ROBOT PATHS BETWEEN SPHERES

B.1 Mathematics for procedure FindP3

This procedure calculates the two possible points P3a and P3b which are in a plane containing the line OP2 and on the surface of a sphere of centre C1 and radius R, such that the plane is tangent to the surface of the sphere (see figure B.1).

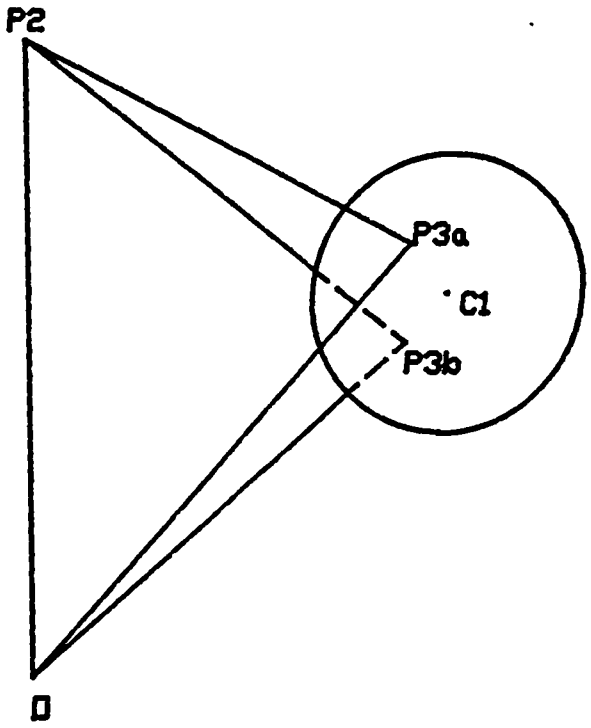


Figure B.1

Firstly a point I on the line OP2 is calculated such that $\vec{OI}$ is perpenpicular to $I\vec{C}1$ (see figure B.2).

By using dot products we get -

$$\vec{P2}.\vec{C1} = |P2||C1|cosa$$

$$but \qquad |I| = |C1|cosa$$

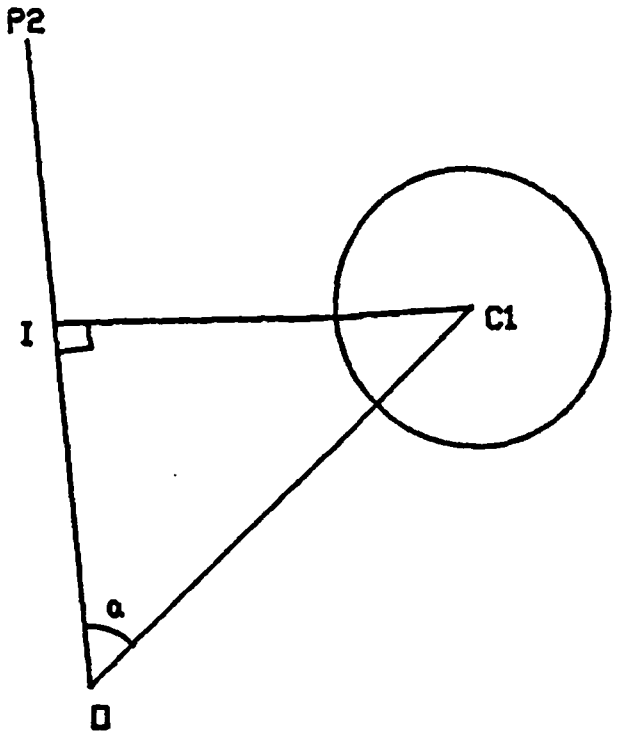


Figure B.2

$$\vec{I} = \vec{P2}\frac{|I|}{|P2|}$$

$$\textit{Therefore} \qquad \vec{I} = \vec{P2}\frac{\vec{P2}.\vec{C1}}{|P2|^2}$$

Figure B.3 shows a Plane perpendicular to OP2

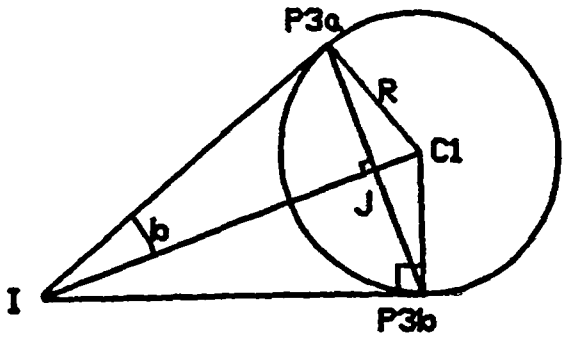


Figure B.3

$$|IP3a| = \sqrt{|IC1|^2 + R^2}$$

$$\vec{IJ} = \vec{IP3}acosb$$

$$cosb = \frac{|IP3a|}{|IC1|}$$

$$\vec{IJ} = \frac{|IP3a|^2}{|IC1|^2}.\vec{IC1}$$

The vector $\vec{JP3}a$ is perpendicular to $\vec{P2}$ and $\vec{IC1}$. Thus it may be found using vector cross products.

$$\frac{\vec{P2} \times \vec{IC2}}{|P2|.|IC1|} = \widehat{JP3a}$$

$$|JP3a| = Rcosb = R\frac{|IP3|}{|IC1|}$$

$$\vec{JP3}a = \frac{R|IP3|}{|P2|.|IC1|^2}\vec{P2} \times \vec{IC1}$$

$$\vec{JP3}b = -\vec{JP3}a$$

B.2 Mathematics to define planes between spheres

This method finds the four possible planes which pass through the origin and are tangential to the surface of two spheres. The two spheres have centres C1 and C2 and radii R1 and R2.

a) planes which pass between the spheres.

Providing that a) the two spheres do not intersect, b) both have radii greater than zero and c) no line from the origin passes through both spheres then there exists two planes which pass between the spheres and are tangential to both their surfaces.

Figure B.4 shows how these planes touch the sphere surfaces at points P2a, P2b, P3a and P3b.

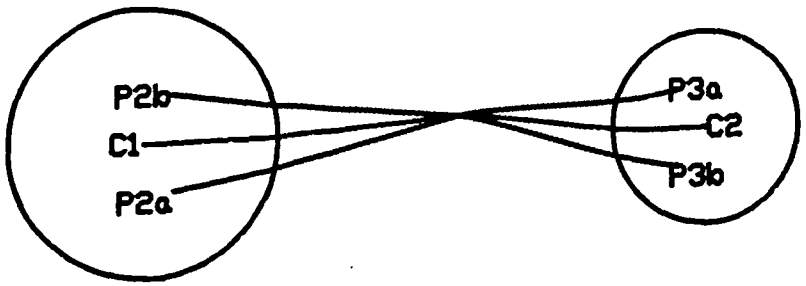


Figure B.4

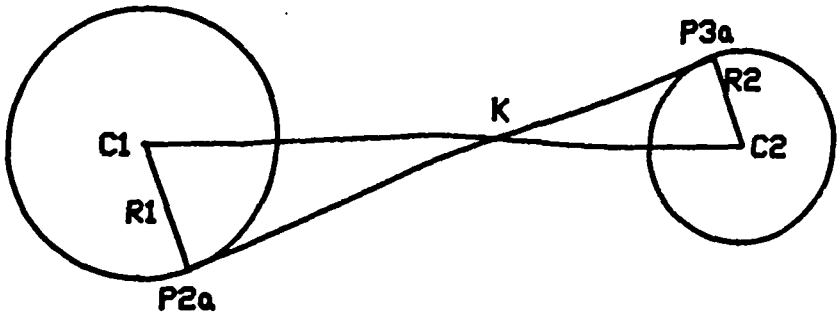


Figure B.5

If we consider the plane containing points C1, P2a and P3a we get figure B.5. As the vectors $C1\vec{P}2a$ and $C2\vec{P}3a$ are both normal to the plane of figure B4 they are parallel vectors and hence C2 lies in the plane of figure B.5 as well.

The point K lies on the line C1C2 and on the line P2aP3a.

$$\frac{R1+R2}{|C1C2|} = \frac{R1}{|C1K|}$$

$$C\vec{1}K = \frac{|C1K|}{|C1C2|} C\vec{1}C2$$

$$\vec{K} = \vec{C1} + C\vec{1}K$$

$$\vec{K} = \vec{C1} + \frac{R1}{R1+R2} C\vec{1}C2$$

It may be seen that the same point K is on the plane through points O, P2b and P3b. Hence the two planes may be found using the method of procedure findP3.

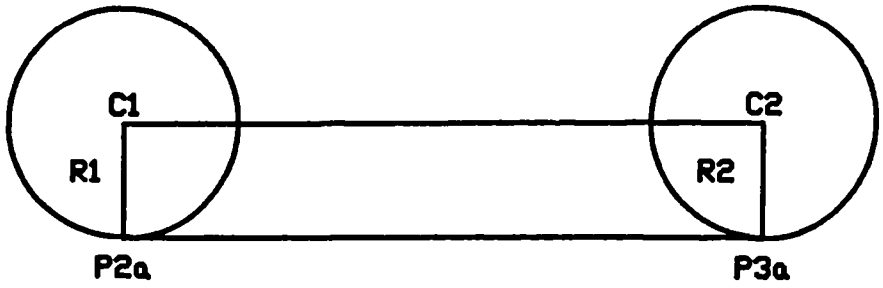


Figure B.6

b) Planes which do not pass between the spheres

(i) spheres of the same radius

For the special case of spheres of the same radius the vector C1C2 is a vector parallel to the planes tangential to the sphere's surfaces and contains the origin. This is shown in figure B.6.

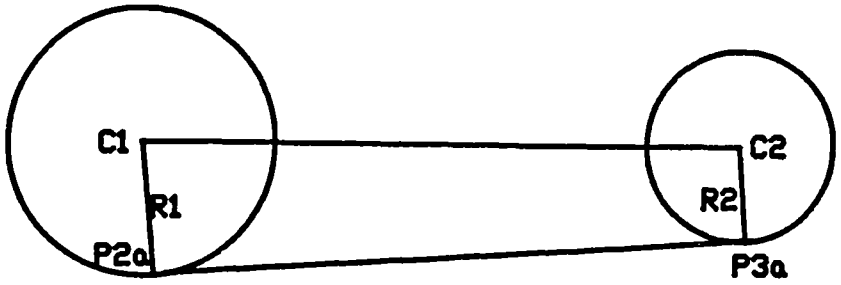


Figure B.7

Hence the point $O + C1\vec{C}2$ is in the plane and FindP3 may be used to define the plane.

(ii) Spheres with unequal radius

Figure B.7 shows the plane containing C1, C2, P2a and P3a. If the lines P2aP3a and C1C2 are extended then they cross at a point

$$\vec{L} = \vec{C1} + \frac{R2 - R1}{R1} C1\vec{C}2$$

Thus with two points the two planes may be found using FindP3.

APPENDIX C

Mainrpf program listing

```
Program Mainrpf;

const L1 : real = 386;
      L2 : real = 376;
      SqL2 = 141376.0;
      Pix3o2 : real = 4.71238898;
      Pio2 : real = 1.570796326;
      Pix2 : real = 6.283185307;
      Safety : integer = 5;
      Zeroi : integer = 0;

type Range = 0..10;
     Dir = (Cl,An);
     Coords = (x,y,z);
     Coord = array[Coords] of real;
     Rangesph = array[Range] of Coord;
     Rangerad = array[Range] of real;
     RangeSp = array[1..8] of Coord;
     RangeNh = array[1..4] of Coord;
     Degree = array[1..3] of real;
     Elb = array[1..2] of real;
     Rangerob = array[Range] of Degree;
     Rangeelb = array[Range] of Elb;
     RangeSL = array[Range,1..2] of real;
     RangeR = array[1..5] of real;
     RangeT = array[1..8] of Degree;
     RangeWkP = array[1..5] of Coord;
     Nodes = record
               Dist, Cost : real;
               Predi : integer;
               Predd : dir;
               Stat : (Open,Closed);
               Nh, P1, P2 : Coord;
             end;
     Rangenod = array[Range,Dir] of Nodes;
     SMd = record
               Grip : boolean;
               R : RangeR;
             end;
     RangeApp = array[1..10] of SMd;
     RangeInt = array[1..50] of SMd;

 var Sph1, Sph2, Sphere, Goal : Rangesph;
     Elbdata : Rangeelb;
     Rad1, Rad2, Radius : Rangerad;
     tempC, TipS, TipG, Startpoint, Goalpoint : Coord;
     Area, Time, loop1, N, G, N1, N2, E, Ans, Rev,
     WkpieceN, Countmvp, Countinit, Countfin, Countint : integer;
     Sr, Gr, Offset : real;
     SLvalT1 : RangeSL;
```

```
    Initialp, Finalp : RangeApp;
    Mvp, Interp : RangeInt;
    WkpConv, WorkPiece : RangeWkP;
{-------------------------------------------------------------------------}
{ The order of procedures is as follows :-
      Invsin, Invcos, Invtan, Keypressed, Modulus, Convert1, Convert2, Convert3,
Convert4, Convtobs, Xproduct, Length_adjust.
      FindP3, Findplan, FindA, Forbidden, Testp, Listnode, Calcdist, Costp,
Pathfind, Error, OutPdata, Routep2, Routep1.
      FindT2, Find_Point, Outputd, SMdata, Avoidobs, Rangefind,
Findpos, Testfa2, Testfa, Stepgoal, Fapath2, Fapath.
      Readdata, Expobs, Rangecalc, Main program }
{-------------------------------------------------------------------------}
function Invsin(Sina : real) : real;
begin
  if Sina < 1e-4 then
    Invsin := Sina
  else
    Invsin := arctan(sqrt(1/(1/sqr(Sina)-1)));
end;
{-------------------------------------------------------------------------}
function Invcos(Cosa : real) : real;
begin
  if Cosa < 1e-4 then
    Invcos := Pi/2-Cosa
  else
    Invcos := arctan(sqrt(1/sqr(Cosa)-1));
end;
{-------------------------------------------------------------------------}
procedure Invtan(var x, y, ans : real);
begin
  if abs(x)<2e-6 then                { This calculates the angle of rotation }
    begin                            { of a line from 0,0 to x,y. If x<2e-6  }
    if abs(y)<2e-6 then              { and y<2e-6 then angle = 0.            }
      ans := 0                       { X is assumed = 0 if x<2e-6.           }
    else if y>0 then
      ans := Pi/2
    else ans := 1.5*pi;
    end
  else if x>0 then
    begin
    if y>=0 then
      ans := arctan(y/x)
    else ans := 2*Pi+arctan(y/x)
    end
  else ans := Pi+arctan(y/x);
end;
{-------------------------------------------------------------------------}
procedure Keypressed;
var B:Byte;
begin
  B := Port[$0DA];
  B:=B and 02;
  if B=2 then read
end;
{-------------------------------------------------------------------------}
procedure Modulus(var A,B,AB : Coord; var Sqmod, Modul : real);

  begin
    AB[X] := B[X]-A[X];
```

```
    AB[Y] := B[Y]-A[Y];
    AB[Z] := B[Z]-A[Z];
    Sqmod := sqr(AB[X])+sqr(AB[Y])+sqr(AB[Z]);
    Modul := sqrt(Sqmod);
  end;
{-------------------------------------------------------------------------}
procedure Convert(var P : Coord; var Ext : integer; var T : Degree);
var L3, ModXY, SqmodXY, D, SqL3, CosT3, SinB, B : real;
begin                                  { This procedure calculates the robot   }
  Invtan(P[x],P[y],T[1]);              { coordinates for when the tip of the   }
  SqmodXY := sqr(P[x])+sqr(P[y]);      { robot arm is at P                     }
  ModXY := sqrt(SqmodXY);              { T3 is always less than Pi.            }
  Invtan(ModXY,P[z],D);
  SqL3 := SqmodXY+sqr(P[z]);
  L3 := sqrt(SqL3);
  if L3 > L1+L2+Ext then begin
    writeln('ERROR Goalpoint out of range');
    end;
  CosT3 := (sqr(L1)+sqr(L2+Ext)-SqL3)/(2*L1*(L2+Ext));
  T[3] := Invcos(CosT3);
  SinB := (L2+Ext)*sin(T[3])/sqrt(SqL3);
  B := Invsin(SinB);
  T[2] := B+D;
  if T[1] > Pi then begin
    if (T[1] > Pix2-0.01) and (T[1] < Pix2+0.01) then
      T[1] := 0
    else if T[1] < Pi+0.01 then
      T[1] := Pi
    else writeln('ERROR');
    end;
  if T[2] > Pix3o2 then T[2] := T[2]-Pix2;
  if T[2] > Pio2 then begin
    if T[2] < Pio2+0.01 then
      T[2] := Pio2
    else writeln('ERROR');
    end;
  writeln('Converted','T[1]=',T[1],' T[2]=',T[2],' T[3]=',T[3]);
  keypressed;
end;
{-------------------------------------------------------------------------}
procedure Convert2(var Th : Degree; var E : Coord);
var temp : real;
begin
  temp := L1*cos(Th[2]);
  E[X] := temp*cos(Th[1]);    { Calculate the coordinates of the robot }
  E[Y] := temp*sin(Th[1]);    { elbow from the robot coordinates       }
  E[Z] := L1*sin(Th[2]);
end;
{-------------------------------------------------------------------------}
procedure Convert3(var Th : Degree; Var E, T : Coord);
var temp1, temp2 : real;
begin
  write('CONVERT3');
  temp1 := Th[2]+Th[3]-Pi;             { Convert Angular coordinates of   }
  temp2 := L2*cos(temp1);              { the robot and the cartesian      }
  T[X] := temp2*cos(Th[1])+E[X];       { coordinates of the elbow, to     }
  T[Y] := temp2*sin(Th[1])+E[Y];       { cartesian coordinates of the tip }
  T[Z] := L2*sin(temp1)+E[Z];          { of the forearm.                  }
  {writeln(T[X],T[Y],T[Z]);}
end;
```

```
{-------------------------------------------------------------------------}
procedure Convert4(var Ec : Coord; var Ea : Elb);
var Xyplane : real;
begin
  Invtan(Ec[X],Ec[Y],Ea[1]);              { Convert the cartesian coordinates}
  Xyplane := sqrt(sqr(Ec[X])+sqr(Ec[Y])); { of the elbow to angular          }
  Invtan(Xyplane,Ec[Z],Ea[2]);            { coordinates.                     }
end;
{-------------------------------------------------------------------------}
procedure Convtobs(var C : Coord; var Offset : real; var Rev : integer);
{ this converts the real coordinates of obstacles to the coordinates as }
{ the robot sees them }
var SqmodOE, ModOE, temp, Beta, A, L1xy : real;
begin
  SqmodOE := sqr(C[X])+sqr(C[Y]);
  ModOE := sqrt(SqModOE);
  L1xy := sqrt(SqmodOE-sqr(Offset));
  Invtan(C[X],C[Y],Beta);
  if Rev = -1 then
    A := Pi/2+Beta
  else begin
    Invtan(Offset,L1xy,temp);
    A := Beta-temp; end;
  C[X] := C[X]-Offset*cos(A);
  C[Y] := C[Y]-Offset*sin(A);
end;
{-------------------------------------------------------------------------}
procedure Xproduct(var A,B,C : Coord);
begin
  C[X] := A[Y]*B[Z]-A[Z]*B[Y];
  C[Y] := A[Z]*B[X]-A[X]*B[Z];
  C[Z] := A[X]*B[Y]-A[Y]*B[X];
end;
{-------------------------------------------------------------------------}
procedure Length_adjust(var P : Coord; var ModP : real);
var temp : real;
begin
  temp := L1/ModP;
  P[X] := P[X]*temp;
  P[Y] := P[Y]*temp;
  P[Z] := P[Z]*temp;
end;
{-------------------------------------------------------------------------}
procedure FindP3(var P2, C1, Nhu, Nhl, P3u, P3l : Coord; var R : real;
                 var Opdir : boolean);
var Temp, SqmodP2, ModP2, P2dotC1, ModIC1, SqmodIC1, SqmodIP3, ModIP3 : real;
    I, IJ, IC1, TempC : Coord;
begin
  writeln('P2=',P2[X],P2[Y],P2[Z]);
  SqmodP2 := sqr(P2[X])+sqr(P2[Y])+sqr(P2[Z]);
  ModP2 := sqrt(SqmodP2);
  P2dotC1 := P2[X]*C1[X]+P2[Y]*C1[Y]+P2[Z]*C1[Z];
  Temp := P2dotC1/SqmodP2;
  I[X] := temp*P2[X];
  I[Y] := temp*P2[Y];
  I[Z] := temp*P2[Z];
  Modulus(I,C1,IC1,SqmodIC1,ModIC1);
  SqmodIP3 := SqmodIC1-sqr(R);
  if SqmodIP3 <=0 then
    begin
```

```
    Opdir := false;
    writeln('**** These two spheres intersect each other ****');
    end
  else
    begin
    ModIP3 := sqrt(SqmodIP3);
    Temp := SqmodIP3/SqmodIC1;
    IJ[X] := Temp*IC1[X];
    IJ[Y] := Temp*IC1[Y];
    IJ[Z] := Temp*IC1[Z];
    Temp := ModIP3*R/(SqmodIC1*ModP2);
    Xproduct(IC1,P2,TempC);
    P3u[X] := I[X]+IJ[X]+temp*TempC[X];
    P3u[Y] := I[Y]+IJ[Y]+temp*TempC[Y];
    P3u[Z] := I[Z]+IJ[Z]+temp*TempC[Z];
    P3l[X] := I[X]+IJ[X]-temp*TempC[X];
    P3l[Y] := I[Y]+IJ[Y]-temp*TempC[Y];
    P3l[Z] := I[Z]+IJ[Z]-temp*TempC[Z];
    writeln('P3u=',P3u[X],P3u[Y],P3u[Z]);
    writeln('P3l=',P3l[X],P3l[Y],P3l[Z]);
{ Find the unit vectors normal to the planes }
    Xproduct(P2,P3u,TempC);
    temp := sqrt(sqr(TempC[X])+sqr(TempC[Y])+sqr(TempC[Z]));
    Nhu[X] := TempC[X]/Temp;
    Nhu[Y] := TempC[Y]/Temp;
    Nhu[Z] := TempC[Z]/Temp;
    Xproduct(P2,P3l,TempC);
    temp := sqrt(sqr(TempC[X])+sqr(TempC[Y])+sqr(TempC[Z]));
    Nhl[X] := TempC[X]/Temp;
    Nhl[Y] := TempC[Y]/Temp;
    Nhl[Z] := TempC[Z]/Temp;
    end;
  Keypressed;
end;
{-------------------------------------------------------------------------}
procedure Findplan(var C1,C2 : Coord; var R1,R2 : real; Var P : RangeSp;
                   var Nh : RangeNh; var Opdir : boolean);
var Temp, ModC1C2, ModC1, CosA : real;
    P2, C1C2 : Coord;
    loop1, Konst : integer;
begin
  writeln('FINDPLAN');
  writeln('C1=',C1[X],C1[Y],C1[Z]);
  writeln('C2=',C2[X],C2[Y],C2[Z]);
  Modulus(C1,C2,C1C2,ModC1C2,Temp);
  if Opdir then
    begin                    { Find ClAn and AnCl planes }
    writeln('Opdir');
    Temp := R1/(R1+R2);
    P2[X] := C1[X]+temp*C1C2[X];
    P2[Y] := C1[Y]+temp*C1C2[Y];
    P2[Z] := C1[Z]+temp*C1C2[Z];
    FindP3(P2,C1,Nh[2],Nh[1],P[3],P[1],R1,Opdir);
    Temp := (R1+R2)/R1;
    P[2,X] := P[1,X]+Temp*(P2[X]-P[1,X]);
    P[2,Y] := P[1,Y]+Temp*(P2[Y]-P[1,Y]);
    P[2,Z] := P[1,Z]+Temp*(P2[Z]-P[1,Z]);
    P[4,X] := P[3,X]+Temp*(P2[X]-P[3,X]);
    P[4,Y] := P[3,Y]+Temp*(P2[Y]-P[3,Y]);
    P[4,Z] := P[3,Z]+Temp*(P2[Z]-P[3,Z]);
```

```
    end;
{ Find ClCl and AnAn Planes }
  if Opdir = false then
    begin
    if R1=0 then
      begin
      FindP3(C1,C2,Nh[4],Nh[3],P[8],P[6],R2,Opdir);
      P[5] := C1;
      P[7] := C1;
      end
    else
      begin
      FindP3(C2,C1,Nh[3],Nh[4],P[5],P[7],R1,Opdir);
      P[6] := C2;
      P[8] := C2;
      end;
    end
  else if R1 = R2 then
    begin
    P2 := C1C2;
    FindP3(P2,C1,Nh[4],Nh[3],P[7],P[5],R1,Opdir);
    P[6,X] := C2[X]+P[5,X]-C1[X];
    P[6,Y] := C2[Y]+P[5,Y]-C1[Y];
    P[6,Z] := C2[Z]+P[5,Z]-C1[Z];
    P[8,X] := C2[X]+P[7,X]-C1[X];
    P[8,Y] := C2[Y]+P[7,Y]-C1[Y];
    P[8,Z] := C2[Z]+P[7,Z]-C1[Z];
    end
  else
    begin
    Temp := R1/(R1-R2);
    P2[X] := C1[X]+Temp*C1C2[X];
    P2[Y] := C1[Y]+Temp*C1C2[Y];
    P2[Z] := C1[Z]+Temp*C1C2[Z];
    if R2>R1 then
      begin
      FindP3(P2,C1,Nh[3],Nh[4],P[7],P[5],R1,Opdir);
      Temp := (R2-R1)/R2;
      end
    else
      begin
      FindP3(P2,C1,Nh[4],Nh[3],P[7],P[5],R1,Opdir);
      Temp := (R1-R2)/R1;
      end;
    P[6,X] := P[5,X]+Temp*(P2[X]-P[5,X]);
    P[6,Y] := P[5,Y]+Temp*(P2[Y]-P[5,Y]);
    P[6,Z] := P[5,Z]+Temp*(P2[Z]-P[5,Z]);
    P[8,X] := P[7,X]+Temp*(P2[X]-P[7,X]);
    P[8,Y] := P[7,Y]+Temp*(P2[Y]-P[7,Y]);
    P[8,Z] := P[7,Z]+Temp*(P2[Z]-P[7,Z]);
    end;
  Konst := 2;
{ if Opdir then Konst := 0;
  for loop1 := Konst+1 to 4 do
    writeln('Nh[',loop1,']=',Nh[loop1,X],Nh[loop1,Y],Nh[loop1,Z]);}
  for loop1 := 2*Konst+1 to 8 do
    writeln('P[',loop1,']=',P[loop1,X],P[loop1,Y],P[loop1,Z]);
  keypressed;
end;
{-------------------------------------------------------------------------}
```

```
procedure Outputd3(var dta1 : Rangeint; var dta2 : RangeApp;
                   var Cnt1, Cnt2 : integer);
var loop1, loop2, Onei : integer;
begin
  Onei := 1;
  for loop1 := 1 to Cnt1 do begin
    write('%');
    if Dta1[loop1].Grip = true then
      write(Onei)
    else write(Zeroi);
    for loop2 := 1 to 5 do
      write(Dta1[loop1].R[loop2]);
    writeln; keypressed; end;
  for loop1 := 1 to Cnt2 do begin
    write('%');
    if Dta2[loop1].Grip = true then
      write(Onei)
    else write(Zeroi);
    for loop2 := 1 to 5 do
      write(Dta2[loop1].R[loop2]);
    writeln; keypressed; end;
end;
{-------------------------------------------------------------------------------}
procedure Outputd2(var Step : SMd; var T : RangeR; var Gp : boolean;
                   var Rangetest : boolean);
{ Converts angles to robot coordinates }
var loop1 : integer;
begin
  write('OUTPUTD2 ');
  Step.R[1] := int((T[1]-3.195)*-283.3);
    if Step.R[1] < 0 then Rangetest := true;
  Step.R[2] := int((T[2]-2.72)*-299.2);
    if Step.R[2] < 0 then Rangetest := true;
  Step.R[3] := int((T[3]-0.4046)*316.4);
    if Step.R[3] < 0 then Rangetest := true;
  Step.R[5] := int(44+(T[4]/(Pi/2)-1)*667);
    if Step.R[5] < 0 then Rangetest := true;
    if Step.R[5] > 999 then Step.R[5] := 999;
  Step.R[4] := int(500+400*T[5]);
    if Step.R[4] < 0 then Rangetest := true;
    if Step.R[4] > 999 then Step.R[4] := 999;
  Step.Grip := Gp;
{  for loop1 := 1 to 5 do
    write(Step.R[loop1]);
  writeln(Gp);}
end;
{-------------------------------------------------------------------------------}
procedure Outputd(var Initp, Finalp : RangeApp; var Mvp, Interp : Rangeint;
                  var Countinit, Countfin, Countmvp, Countint : integer);
var loop1, loop2 : integer;
begin
  writeln('OUTPUTD');
  Outputd3(Mvp,Initp,Countmvp,Countinit);
  Outputd3(Interp,Finalp,Countint,Countfin);
  writeln('&');
end;
{-------------------------------------------------------------------------------}
procedure SMdata2(var T : Degree; var Interp : Rangeint; var Countint : integer);
{ Converts angles to robot coordinates and outputs them }
var temp, R1, R2, R3, R4, R5 : real;
```

```
    G, loop1 : integer;
begin
  write('SMDATA2 ');
  Countint := Countint+1;
  if T[1] > Pi then begin
    if T[1] > Pix2-0.01 then
      T[1] := 0
    else if T[1] < Pi+0.01 then
      T[1] := Pi
    else
      writeln('ERROR  - Rotation coordinate out of range');
    end;
  Interp[Countint].R[1] := int((T[1]-3.195)*-283.3);
  if T[2] > Pi/2 then begin
    if T[2] > Pix3o2 then
      T[2] := T[2]-Pix2
    else if T[2] < Pio2+0.01 then
      T[2] := Pio2
    else
      writeln('ERROR  - Elevation coordinate out of range');
    end;
  Interp[Countint].R[2] := int((T[2]-2.72)*-299.2);
  Interp[Countint].R[3] := int((T[3]-0.4046)*316.4);
  Interp[Countint].R[4] := 500;
  Interp[Countint].R[5] := 736;
  Interp[Countint].Grip := true;
  Keypressed;
end;
{-------------------------------------------------------------------------}
procedure SMdata(var Robcoor : Rangerob; var CountRc, Countint : integer;
                 var Interp : Rangeint);
{ convert the step coordinates to a list of robot coordinates which will guide
the robot in the correct path }
var loop1, loop2, OPnum : integer;
    Spaceing1, Spaceing2, Spaceing3, Dif1, Dif2, Dif3,
    temp4, Varience : real;
    Angles1, Angles2, tempD : Degree;
    Gp : boolean;
begin
  write('SMDATA ');
  Countint := 0;
{ write the coordinates of the startpoint }
  SMdata2(Robcoor[0],Interp,Countint);
  for loop1 := 0 to CountRc-1 do begin
{ set angles1 and Angles2 to the begining and end of the step }
    Angles1 := Robcoor[loop1]; Angles2 := Robcoor[loop1+1];
{   writeln('Angles1=',Angles1[1],Angles1[2],Angles1[3]);
    writeln('Angles2=',Angles2[1],Angles2[2],Angles2[3]);}
    if Angles1[2] > 3*Pi/2 then             { get around problem of  }
      Angles1[2] := Angles1[2]-2*Pi;        { interpolation around 0 }
    if Angles2[2] > 3*Pi/2 then
      Angles2[2] := Angles2[2]-2*Pi;
    Dif1 := Angles2[1]-Angles1[1];
    Dif2 := Angles2[2]-Angles1[2];
    Dif3 := Angles2[3]-Angles1[3];
{ find the largest range of angle for the three degrees of freedom }
    if abs(Dif2)>abs(Dif1) then begin
      if abs(Dif3)>abs(Dif2) then
        Varience := abs(Dif3)
      else Varience := abs(Dif2); end
```

```
    else if abs(Dif3)>abs(Dif1) then
      Varience := abs(Dif3)
    else Varience := abs(Dif1);
    temp4 := int(Varience/0.05);
    if temp4>0 then begin
{ calculate the intermediate coordinates }
      Spaceing1 := Dif1/(temp4+1);
      Spaceing2 := Dif2/(temp4+1);
      Spaceing3 := Dif3/(temp4+1);
      tempD[1] := Angles1[1];
      tempD[2] := Angles1[2];
      tempD[3] := Angles1[3];
      while temp4 >0 do begin
        temp4 := temp4-1;
        tempD[1] := tempD[1]+Spaceing1;
        tempD[2] := tempD[2]+Spaceing2;
        tempD[3] := tempD[3]+Spaceing3;
        SMdata2(tempD,Interp,Countint); end;
      end;   { if temp4 }
    SMdata2(Angles2,Interp,Countint);
    end;   { for loop1 }
  Keypressed;
end;
{-------------------------------------------------------------------------}
Procedure FindA(var V, U : Coord; var A : real);
var ModV, VdotU, CosA : real;
begin
{  writeln('FindA');}
  ModV := sqrt(sqr(V[X])+sqr(V[Y])+sqr(V[Z]));
  VdotU := V[X]*U[X]+V[Z]*U[Z];
  CosA := VdotU/ModV;
{  writeln('CosA=',CosA);}
  A := invcos(CosA);
  writeln('A=',A),
{ if (CosA > -1e-5) and (CosA < 1e-5) then
    begin
    if V[Y] > 0 then
      A := Pi/2
    else
      A := 3*Pi/2;
    end
  else
    begin
    A := arctan(sqrt(1/sqr(CosA)-1));
    if V[X] < 0 then
      A := A+Pi
    else if V[Y]<0 then
      A := A+2*Pi;
    end;}
end;
{-------------------------------------------------------------------------}
procedure Forbidden(var P1,P2 : Coord; var Obst : boolean);
begin
  if (P1[Z] < -300) or (P2[Z] < -300) then
    writeln('******** Error Forbidden Coordinate ********');
  end;
{-------------------------------------------------------------------------}
procedure Testp(var N : integer; var Sph : Rangesph; var Rad : Rangerad;
                var S1, S2 : integer; var Nh, P1, P2 : Coord; var Obst : boolean);
```

```
var U, OB, tempC : Coord;
    D, A1, A2, A3, Clear, ModOB, temp, ModOBC : real;
    loop1 : integer;
    Inrange : boolean;

begin
  WRITELN('TESTING THE PATH BETWEEN sphere ',S1,' AND sphere ',S2);
  writeln('Nh=',Nh[X],Nh[Y],Nh[Z]);
  Obst := false; loop1 := 0; Inrange := true;
  Forbidden(P1,P2,Obst);
  if Obst = true then
    loop1 := 100;
{ U is a vector in the plane which is given by (0,1,0) x Nh }
  U[X] := Nh[Z];
  U[Z] := Nh[X];
  repeat
    loop1 := loop1+1;
    if loop1 = S1 then            { make sure loop1 <> S1 or S2 }
      loop1 := loop1+1;
    if loop1 = S2 then
      loop1 := loop1+1;
    if loop1 = S1 then
      loop1 := loop1+1;
    if loop1<=N then
      begin
      writeln('Sphere[',LOOP1,']=',Sph[LOOP1,X],Sph[LOOP1,Y],Sph[LOOP1,Z]);
{ find Distance from sphere centre to plane }
      D := Sph[loop1,X]*Nh[X]+Sph[loop1,Y]*Nh[Y]+Sph[loop1,Z]*Nh[Z];
      D := abs(D);
      writeln('the distance from the sphere center to the plane is',D);
      if D <= Rad[loop1] then
        begin
        OB[X] := Sph[loop1,X]-D*Nh[X];
        OB[Y] := Sph[loop1,Y]-D*Nh[Y];
        OB[Z] := Sph[loop1,Z]-D*Nh[Z];
        ModOB := sqrt(sqr(OB[X])+sqr(OB[Y])+sqr(OB[Z]));
        if ModOB > L1 then
          begin
          temp := L1/ModOB;
          OB[X] := temp*OB[X];
          OB[Y] := temp*OB[Y];
          OB[Z] := temp*OB[Z];
          Modulus(OB,Sph[loop1],tempC,ModOBC,temp);
          if ModOBC > Rad[loop1] then
            Inrange := false;
          end;
        if Inrange then
          begin
          FindA(P1,U,A1);
          FindA(P2,U,A2);
          FindA(OB,U,A3);
          writeln('A1=',A1,' A2=',A2,' A3=',A3);
          if ((A3>=A1) and (A3<=A2)) or ((A3<=A1) and (A3>=A2)) then
            Obst := true
          else if ((A3<A1) and (A1<A2)) or ((A3>A1) and (A1>A2)) then
            begin
            if abs(A3-A1)<Pi/2 then
              begin
              Clear := sqrt(sqr(D)+sqr(ModOB*sin(abs(A3-A1))));
              if Clear<=Rad[loop1] then
```

```
              Obst := True;
            end;
          end
        else
          begin
          if abs(A3-A2)<Pi/2 then
            begin
            Clear := sqrt(sqr(D)+sqr(ModOB*sin(abs(A3-A1))));
            if Clear<=Rad[loop1] then
              Obst := True;
            end;
          end;
        end;
      end;
  until (loop1>=N) or (Obst=True);
  if Obst = true then
    writeln('Obst = true')
  else
    writeln('Obst = false');
end;
{-------------------------------------------------------------------------------}
procedure Listnode(var Sph : RangeSph; var Rad : RangeRad; var Node : Rangenod;
                   var S : integer; var Sdir : dir);
var Nh, P1, P2 : coord;
    Offset : real;
    Rev : integer;
begin
  if Node[S,Sdir].Cost <> 999 then
    begin
    writeln('Sphere[',S,'] ',Sph[S,X],Sph[S,Y],Sph[S,Z],Rad[S]);
    P1 := Node[S,Sdir].P1;
    P2 := Node[S,Sdir].P2;
    Nh := Node[S,Sdir].Nh;
    write('node',S);
    if Sdir = cl then
      write(' clockwise')
    else write(' anticlockwise');
    if Node[S,Sdir].stat = Open then
      writeln(' stat= open')
    else writeln(' stat = closed');
    write('Dist=',Node[S,Sdir].Dist,' Cost=',Node[S,Sdir].Cost);
    write(' Pred=',Node[S,Sdir].Predi);
    if Node[S,Sdir].Predd = cl then
      writeln(' Clockwise')
    else
      writeln(' Anticlockwise');
    writeln('Nh=',Nh[X],Nh[Y],Nh[Z]);
    writeln('P1=',P1[X],P1[Y],P1[Z]);
    writeln('P2=',P2[X],P2[Y],P2[Z]);
  end;
end;
{-------------------------------------------------------------------------------}
procedure Calcdist(var P1, P2 : Coord; var Dist : real);
var ModP1, ModP2, temp, P1dotP2 : real;
{ Calculate the angular distance between points P1 and P2 }
begin
  P1dotP2 := P1[X]*P2[X]+P1[Y]*P2[Y]+P1[Z]*P2[Z];
  ModP1 := sqrt(sqr(P1[X])+sqr(P1[Y])+sqr(P1[Z]));
  ModP2 := sqrt(sqr(P2[X])+sqr(P2[Y])+sqr(P2[Z]));
```

```
  if P1dotP2 = 0 then
    Dist := Pi/2
  else
    begin
    temp := P1dotP2/(ModP1*ModP2);
    if temp >= 1 then
      Dist := 0
    else
      Dist := abs(arctan(sqrt(1/sqr(temp)-1)));
    end;
end;
{-------------------------------------------------------------------------}
procedure Costp(var Node : Rangenod; var S1, S2 : integer; var P1, P2, G, N2h :
                Coord; var R1 : real; var S1dir, S2dir : Dir);
{ find the best route to Node S2 }
var Temp, Nbcost, Dangle, ModP1, ModP2, ModG, Cosa, Costsp,
    P1dotP2, P2dotG, Dist : real;
    N1h : Coord;

begin
  writeln('COSTP');
  ModG := sqrt(sqr(G[X])+sqr(G[Y])+sqr(G[Z]));
  if S1dir=Cl then
    writeln('S1dir=Clock')
  else
    writeln('S1dir=Anti');
  N1h := Node[S1,S1dir].Nh;
  writeln('N1h=',N1h[X],N1h[Y],N1h[Z]);
  writeln('N2h=',N2h[X],N2h[Y],N2h[Z]);
{ find the angle traveled through when going around S1 }
  Cosa := N1h[X]*N2h[X]+N1h[Y]*N2h[Y]+N1h[Z]*N2h[Z];
  if (Cosa<1e-5) and (Cosa>-1e-5) then
    Dangle := Pi/2
  else
    Dangle := arctan(sqrt(1/sqr(Cosa)-1));
  ModP1 := sqrt(sqr(P1[X])+sqr(P1[Y])+sqr(P1[Z]));
  Dangle := Dangle*R1/ModP1;
  writeln('ModP1=',ModP1,' Dangle=',Dangle);
{ find the distance from P2 to G }
  Calcdist(P2,G,Dist);
{ find the cost of the path in a plane }
  Calcdist(P1,P2,Costsp);
  Writeln('Dist=',Dist,' Costsp=',Costsp);
{ find the cost of this path }
  Nbcost := Costsp+Dangle+Dist-Node[S1,S1dir].Dist+Node[S1,S1dir].cost;
  writeln('Costsp=',Costsp,' Dangle=',Dangle,' S1.Dist=',Node[S1,S1dir].Dist);
  writeln('Node[S1,S1dir].cost=',Node[S1,S1dir].cost);
  if Nbcost<=Node[S2,S2dir].cost then
    begin
    Node[S2,S2dir] cost  := Nbcost;
    Node[S2,S2dir].Predi := S1;
    Node[S2,S2dir].Predd := S1dir;
    Node[S2,S2dir].stat  := Open;
    Node[S2,S2dir].dist  := Dist;
    Node[S2,S2dir].Nh := N2h;
    Node[S2,S2dir].P1 := P1;
    Node[S2,S2dir].P2 := P2;
    end;
  Listnode(Sphere,Radius,Node,S2,S2dir);
  writeln('the end of costp');
```

```
        end;
        {-------------------------------------------------------------------------}
        procedure Pathfind(var N : integer; var Sph : Rangesph; var Rad : Rangerad;
                           var Node : Rangenod; var S1, S2 : integer);
        { This procedure generates paths between two spheres }
        var R1 : real;
            Obst, Opdir, Clockpos, AnClpos : boolean;       { Opposite direction }
            G : Coord;
            Clock, Anti : Dir;
            Goalnode : integer;
            Nh : RangeNh;
            Sp : RangeSp;

      begin
        writeln('pathfind between ',S1,' and ',S2);
        Opdir := true;
        Clock := Cl; Anti := An;
        Goalnode := N+1;
        G := Sph[Goalnode];
        R1 := Rad[S1];
        if not((S1=0) and (S2=Goalnode)) then
          begin
          if (Rad[S1]=0) or (Rad[S2]=0) then
            Opdir := false;
          Findplan(Sph[S1],Sph[S2],Rad[S1],Rad[S2],SP,Nh,Opdir);
          Clockpos := false; AnClpos := false;
          if Node[S1,Clock].Cost < 999 then Clockpos := true;
          if Node[S1,Anti].Cost < 999 then AnClpos := true;
          if Clockpos then
            begin
            Testp(N,Sph,Rad,S1,S2,Nh[3],Sp[5],Sp[6],Obst);
            if Obst = false then
              Costp(Node,S1,S2,SP[5],SP[6],G,Nh[3],R1,Clock,Clock);
            end;
         if AnClpos then
            begin
            Testp(N,Sph,Rad,S1,S2,Nh[4],Sp[7],Sp[8],Obst);
            if Obst = false then
              Costp(Node,S1,S2,SP[7],SP[8],G,Nh[4],R1,Anti,Anti);
            end;
        if Opdir then
          begin
          if Clockpos then
          begin
            Testp(N,Sph,Rad,S1,S2,Nh[1],Sp[1],Sp[2],Obst);
            if Obst = false then
              Costp(Node,S1,S2,SP[1],SP[2],G,Nh[1],R1,Clock,Anti);
            end;
         if AnClpos then
            begin
            Testp(N,Sph,Rad,S1,S2,Nh[2],Sp[3],Sp[4],Obst);
            if Obst = false then
              Costp(Node,S1,S2,SP[3],SP[4],G,Nh[2],R1,Anti,Clock);
            end;
      end;
      writeln(' This is the end of findpath ');
    end;
  end;
{-------------------------------------------------------------------------}
procedure OutPdata(var Node : Rangenod; var Goalnode, Elbcnt : integer;
```

```
                    var Elbdata : Rangeelb; var ElbowS, ElbowG : Coord);
{ write out data for the path which ropaf must follow }
var loop1, Sph1, Sph2, tempI, Rev : integer;
    temp1, temp2, Offset, ModP1aP1b, ModP2aP2b, D1, D2 : real;
    P1a, P1b, P2a, P2b, P1aP1b, P2aP2b, tempC : Coord;
    Elbdata1 : Rangeelb;
    Dir1, Dir2 : Dir;
    Dontdoit : boolean;
    Filename : String[12];
    Datafile : File of real;
begin
  writeln('OUTPDATA');
  Dontdoit := false;
  Sph1 := Goalnode; Elbcnt := 0;
  temp1 := Node[Sph1,Cl].Cost;
  temp2 := Node[Sph1,An].Cost;
  Dir1 := Cl;
  if temp2 < temp1 then
    Dir1 := An;
  P1a := Node[Sph1,Dir1].P1;
  P1b := Node[Sph1,Dir1].P2;
  if Node[Sph1,Dir1].Predi>0 then
    begin
    repeat
      writeln('Sph1=',Sph1);
      P2a := P1a;
      P2b := P1b;
      Sph2 := Sph1;
      Dir2 := Dir1;
      Sph1 := Node[Sph2,Dir2].Predi;
      Dir1 := Node[Sph2,Dir2].Predd;
      P1a := Node[Sph1,Dir1].P1;
      P1b := Node[Sph1,Dir1].P2;
      writeln('P1a=',P1a[X],P1a[Y],P1a[Z]);
      writeln('P1b=',P1b[X],P1b[Y],P1b[Z]);
      writeln('P2a=',P2a[X],P2a[Y],P2a[Z]);
      writeln('P2b=',P2b[X],P2b[Y],P2b[Z]);
      Modulus(P1a,P1b,P1aP1b,temp1,ModP1aP1b);
      Modulus(P2a,P2b,P2aP2b,temp1,ModP2aP2b);
      D1 := sqrt(sqr(P2a[X]-P1b[X])+sqr(P2a[Y]-P1b[Y])+sqr(P2a[Z]-P1b[Z]));
      D2 := sqrt(sqr((P1b[X]-P1aP1b[X]/ModP1aP1b)-(P2a[X]+P2aP2b[X]/ModP2aP2b))
            +sqr((P1b[Y]-P1aP1b[Y]/ModP1aP1b)-(P2a[Y]+P2aP2b[Y]/ModP2aP2b))
            +sqr((P1b[Z]-P1aP1b[Z]/ModP1aP1b)-(P2a[Z]+P2aP2b[Z]/ModP2aP2b)));
      tempC[X] := P1b[X]+(D1/(D2-D1))*P1aP1b[X]/ModP1aP1b;
      tempC[Y] := P1b[Y]+(D1/(D2-D1))*P1aP1b[Y]/ModP1aP1b;
      tempC[Z] := P1b[Z]+(D1/(D2-D1))*P1aP1b[Z]/ModP1aP1b;
      temp1 := sqrt(sqr(tempC[X])+sqr(tempC[Y])+sqr(tempC[Z]));
      writeln('D1=',D1,' D2=',D2,' ModP1aP1b=',ModP1aP1b);
      writeln('ModP2aP2b=',ModP2aP2b,' temp1=',temp1);
      writeln('tempC=',tempC[X],tempC[Y],tempC[Z]);
      Length_adjust(tempC,temp1);
      Elbcnt := Elbcnt+1;
      Convert4(tempC,Elbdata1[Elbcnt]);
    until Node[Sph1,Dir1].Predi = 0;
    end;
  for loop1 := 1 to Elbcnt do
    Elbdata[loop1] := Elbdata1[Elbcnt+1-loop1];
  Elbcnt := Elbcnt+1;
  Convert4(ElbowS,Elbdata[0]);
  Convert4(ElbowG,Elbdata[Elbcnt]);
```

```
  Keypressed;
end;
{-------------------------------------------------------------------------}
procedure Routep2(var Sph : Rangesph ; var Rad : Rangerad; var N,
                  Goalnode : integer; var Sp : RangeSp; var Node : Rangenod);
{ FIND A PATH TO THE GOALPOINT }

var Loop1, Opennode : integer;
    Obst, Opdir : boolean;
    Noncost : real;
    Nh : RangeNh;                          { ClAn, AnCl, ClCl, AnAn }
    Opendir, Nopendir, Loopvar : Dir;

begin
{ check whether a clear path exists }
  Opdir := false; Opennode := 0;
  Findplan(Sph[0],Sph[Goalnode],Rad[0],Rad[Goalnode],Sp,Nh,Opdir);
  Testp(N,Sph,Rad,Opennode,Goalnode,Nh[3],Sp[5],Sp[6],Obst);
  if Obst = false then begin
    Opennode := Goalnode;
    Node[Goalnode,Cl].Predi := 0;
    Node[Goalnode,Cl].Cost := 0; end;
{ find a path around obstacles }
  while Opennode < Goalnode do
    begin
    loop1 := 0;
    repeat { EXPAND OPENNODE }
      Loop1 := Loop1+1;
      IF Loop1 = Opennode
        then Loop1 := Loop1+1;
      IF Loop1 <= Goalnode then
        begin
        Pathfind(N,Sph,Rad,Node,Opennode,Loop1);
        writeln('this is routep again');
        end;
    until Loop1 >=Goalnode;

    writeln('choose the next opennode');
    Node[Opennode,Cl].Stat := Closed;       { FIRST CLOSE THE OPENNODE }
    Node[Opennode,An].Stat := Closed;
    Noncost := 999;     { NEW OPENNODE COST }
    for loop1 := 1 TO N+1 DO
      begin
      for Loopvar := Cl to An do
        begin
        if (Node[loop1,loopvar].Stat = Open)
        and (Node[loop1,Loopvar].cost<=Noncost) then
          begin
          Noncost := Node[loop1,Loopvar].cost;
          Opennode := loop1;
          Opendir := loopvar;
          end;
        end;
      end;
    write('The new opennode is ',Opennode,' the direction is ');
    if Opendir = Cl
      then writeln('Clockwise');
    if opendir = An
      then writeln('Anticolckwise');
    end;   { while }
```

```
  Keypressed;
end;
{-------------------------------------------------------------------------}
procedure Routep1(var Sphere : Rangesph; var Elbdata : Rangeelb;
                  var Radius : Rangerad; var N1, Elbcnt : integer;
                  var Startpoint, Goalpoint : Coord);

var Loop1, Goalnode, N, Rev : integer;
    Zero, ElbowS, ElbowG : Coord;
    Loopvar, Clock, Anti : Dir;
    Node : Rangenod;
    Sp : RangeSp;
    temp1, temp2, Offset : real;
    tempD : Degree;
    Sph : Rangesph;
    Rad : Rangerad;

 begin
   writeln('THIS IS ROUTEPATH');
   Clock := Cl; Anti := An;
   N := N1;
   Zero[X] := 0; Zero[Y] := 0; Zero[Z] := 0;

 { Convert Obstacles }
   Sph := Sphere; Rad := Radius;
   Goalnode := N+1;
 { Calculate the elbow (end of the upper arm) startpoint and goalpoint }
   Rev := 1; Offset := 51;
   ElbowS := Startpoint; ElbowG := Goalpoint;
   Convtobs(ElbowS,Offset,Rev); Convert(ElbowS,Zeroi,tempD); Convert2(tempD,ElbowS);
   Convtobs(ElbowG,Offset,Rev); Convert(ElbowG,Zeroi,tempD); Convert2(tempD,ElbowG);
   Sph[0] := ElbowS; Sph[Goalnode] := ElbowG;
   writeln('ElbowS=',ElbowS[X],ElbowS[Y],ElbowS[Z]);
   writeln('ElbowG=',ElbowG[X],ElbowG[Y],ElbowG[Z]);
   Rad[0] := 0; Rad[Goalnode] := 0;
   Node[0,Clock].P2 := Sph[0]; Node[0,Anti].P2 := Sph[0];
   Node[Goalnode,Clock].P1 := Sph[Goalnode];
   Node[Goalnode,Anti].P1 := Sph[Goalnode];

   { INITIALISE TOTALC VALUES }
   for Loop1 := 1 to Goalnode do
     begin
     for Loopvar := Cl to An do
       begin
       Node[Loop1,Loopvar].Cost := 999;
       Node[Loop1,Loopvar].Stat := Open;
       end;
     end;
    Node[0,Clock].stat := Closed; Node[0,Anti].stat := Closed;
    Node[0,Clock].cost := 0; Node[0,Anti].cost := 0;
    Node[0,Clock].Nh := Zero; Node[0,Anti].Nh := Zero;
    Calcdist(ElbowS,ElbowG,Node[0,Clock].dist);
    Node[0,Anti].dist := Node[0,Clock].dist;
    Routep2(Sph,Rad,N,Goalnode,Sp,Node);
    OutPdata(Node,Goalnode,Elbcnt,Elbdata,ElbowS,ElbowG);
    Keypressed;
  end;
  {-------------------------------------------------------------------------}
  procedure FindT2(var T1, T2 : real; var E1,E2,En : Coord);
  { Find T2 which is between E1 and E2, and find the new elbow coords En }
```

```
var ModEn, temp, Dy, Dx, Adj, Px, Py, Pz, Lambda, TanT1 : real;
begin
  write('FindT2 ');
  Dy := E2[Y]-E1[Y];
  Dx := E2[X]-E1[X];
  {writeln('Dx=',Dx,' Dy=',Dy,' T1=',T1);}
  temp := cos(T1);                              { find T1 }
  if (temp < 1e-4) and (temp > -1e-4) then
    TanT1 := 1e10
  else
    TanT1 := sin(T1)/temp;
  if (Dx=0) and (Dy=0) then Lambda := 0
  else Lambda := (TanT1*E1[X]-E1[Y])/(Dy-TanT1*Dx);
  {writeln('TanT1=',TanT1,' Lambda=',Lambda);}
  En[X] := E1[X]+Lambda*Dx;
  En[Y] := E1[Y]+Lambda*Dy;
  En[Z] := E1[Z]+Lambda*(E2[Z]-E1[Z]);
  temp := sqr(En[X])+sqr(En[Y]);
  Adj := sqrt(temp);
  Invtan(Adj,En[Z],T2);
  ModEn := sqrt(temp+sqr(En[Z]));
  Length_adjust(En,ModEn);
  {writeln('En=',En[X],En[Y],En[Z]);}
  Keypressed;
end;
{-------------------------------------------------------------------------------}
procedure Find_Point(var S, G, C, Pj : Coord);
{This finds the point Pj on the line SG closest to C}
var Sc_Sg, ModSg, temp, r : real;
    Sc, Sg : Coord;

begin
  write('FIND_POINT ');                 { find the point P on Sq such }
  Sc[X] := C[X]-S[X];                   { that Cp and Sp meet at a right angels}
  Sc[Y] := C[Y]-S[Y];
  Sc[Z] := C[Z]-S[Z];
  Modulus(S,G,Sg,Temp,ModSg);
  Sc_Sg := Sc[X]*Sg[X]+Sc[Y]*Sg[Y]+Sc[Z]*Sg[Z];
  r := Sc_Sg/ModSg;
  if r >= ModSg then            { is the point at the tip of the forearm? }
    Pj := G
  else if r <= 0 then                   { is the point at the elbow }
    Pj := S
  else begin                            { the point is in between }
    temp := r/ModSg;
    Pj[X] := S[X]+temp*Sg[X];
    Pj[Y] := S[Y]+temp*Sg[Y];
    Pj[Z] := S[Z]+temp*Sg[Z]; end;      { hello says Sarah }
  {writeln('Pj[X]=',Pj[X],' Pj[Y]=',Pj[Y],' Pj[Z]=',Pj[Z]);}
  keypressed,
end;
{-------------------------------------------------------------------------------}
procedure Avoidobs(var C, Pjf, CPjf, Se, Ge : coord; var Sf : real;
                   var S, G : Degree; var Closestpart : integer);
{ find new T2 and T3 values which avoid the obstacle }
var N : Degree;
    Pn, En, PnEn, E, T : Coord;
    XYdist, temp, EndotPnEn, ModPnEn, ModOPnxy, a : real;
    Box : integer;
begin
```

```
  writeln('AVOIDOBS');
  Box := 0;
  Pn[X] := C[X]+Sf*CPjf[X];
  Pn[Y] := C[Y]+Sf*CPjf[Y];
  Pn[Z] := C[Z]+Sf*CPjf[Z];

  { customise to application }
  writeln('Pjf=',Pjf[X],Pjf[Y],Pjf[Z]);
  if ((Pjf[X]>200)and(Pjf[X]<600)) then begin
    if Pjf[Y] > 400 then
      begin
      writeln('avoiding box1');
      Box := 1; end;
    end
  else if ((Pjf[X]>-350)and(Pjf[X]<-50)) then begin
    writeln('2');
    if Pjf[Y] > 450 then
      begin
      writeln('Avoiding box2');
      Box := 2; end;
    end;

  Invtan(Pn[X],Pn[Y],N[1]);
  case Closestpart of
1:  writeln('avoiding a collision with the forearm');
2:  begin
    writeln('Avoiding a collision with the gripper motor');
    ModOPnxy := sqrt(sqr(Pn[X])+sqr(Pn[Y]));
    N[1] := N[1]-invsin(22/ModOPnxy);
    Pn[X] := Pn[X]+22*sin(N[1]);
    Pn[Y] := Pn[Y]-22*cos(N[1]);
    end;
3:  begin
    writeln('Avoiding a collision with the part');
    ModOPnxy := sqrt(sqr(Pn[X])+sqr(Pn[Y]));
    N[1] := N[1]-invsin(44/ModOPnxy);
    a := Pio2-N[1];
    Pn[X] := Pn[X]+44*cos(a);
    Pn[Y] := Pn[Y]-44*sin(a);
    end;
  end; { case }

  case Box of
1:  begin
    XYdist := sqrt(sqr(Pn[X])+sqr(Pn[Y]));
    if XYdist > 720 then begin
      Pn[X] := 300; Pn[Z] := 500; end;
    end;
2:  begin
    XYdist := sqrt(sqr(Pn[X])+sqr(Pn[Y]));
    if XYdist > 630 then begin
      Pn[X] := -100; Pn[Z] := 500; end;
    end;
  end; { case }
  FindT2(N[1],N[2],Se,Ge,En);
  Modulus(Pn,En,PnEn,temp,ModPnEn);
  EndotPnEn := En[X]*PnEn[X]+En[Y]*PnEn[Y]+En[Z]*PnEn[Z];
  N[3] := Invcos(EndotPnEn/(ModPnEn*L1));
  Convert2(N,E);
  Convert3(N,E,T);
```

```
  G := N; Ge := E;
  writeln('SF=',SF);
  writeln('Cpjf=',Cpjf[X],Cpjf[Y],Cpjf[Z]);
  writeln('Pn=',Pn[X],Pn[Y],Pn[Z]);
  writeln('N=',N[1],N[2],N[3]);
  writeln('The cartesian coordinates of the robot are');
  writeln('Elbow:',E[X],E[Y],E[Z]);
  writeln('Tip  :',T[X],T[Y],T[Z]);
end;
{-------------------------------------------------------------------------}
procedure Rangefind(var Ta,Tb,Ts,Tl : real; var Outofrange : boolean);
{Find the smallest and largest values of T1 for which a collision is possible}
var Temp : real;
begin
  write('RANGEFIND ');
  {writeln('Ta=',Ta,' Tb=',Tb,' Ts=',Ts,' Tl=',Tl);}
  Outofrange := false;
  if Tb < Ta then begin
    temp := Ta;
    Ta := Tb;
    Tb := temp; end;
  If (Ta > Ts) and (Ta < Tl) then Ts := Ta;
  If (Tb < Tl) and (Tb > Ts) then Tl := Tb;
  If (Tb < Ts) then Outofrange := true;
  If (Ta > Tl) then Outofrange := true;
  keypressed;
  writeln('Ts=',Ts,' Tl=',Tl);
end;
{-------------------------------------------------------------------------}
procedure Findpos(var T : Degree; var Tip, Pos : Coord; var Numpart : integer);
var a, Sina, Cosa, SinT1, CosT1 : real;
begin
  write('FINDPOS ');
  case Numpart of
2:  begin
    Pos[X] := Tip[X]-22*sin(T[1]);
    Pos[Y] := Tip[Y]+22*cos(T[1]);
    Pos[Z] := Tip[Z];
    end;
3:  begin
    a := (T[2]+T[3])-Pi;
    Cosa := cos(a); Sina := sin(a);
    CosT1 := cos(T[1]); SinT1 := sin(T[1]);
    Pos[X] := Tip[X]+120*Cosa*CosT1-13*Sina*CosT1-44*SinT1;
    Pos[Y] := Tip[Y]+120*Cosa*SinT1-13*Sina*SinT1+44*cosT1;
    Pos[Z] := Tip[Z]+120*Sina+13*Cosa;
    end;
  end;
  {writeln('Pos=',Pos[X],Pos[Y],Pos[Z]);}
end;
{-------------------------------------------------------------------------}
procedure Testfa2(var C, Pjf, CPjf, Se, Ge : Coord; var Rad, Sf, T1s, T1l : real;
            var S, G : Degree; var Obst : boolean;
            var Closestpart : integer);
{ Finds the closest point (Pjf), on the surface swept by the robot forearm, to
the obstacle sphere }
type RangePrad = array[1..3] of real;
var Psf, Plf, CPjftemp, Pjftemp, TsEc, TlEc, TsTc, TlTc,
  tempc, Sn, Poss, Posl : Coord;
  T1n, T2n, T3n, T1c, T3c, Range,
```

```
  V3, temp, Intmax, Int, R1, R2, ModCPjf : real;
  Ts, TI : Degree;
  Partnum, Numpart : integer;
  Partrad : RangePrad;
begin
  Partrad[1] := 45; Partrad[2] := 89; Partrad[3] := 40;
  Numpart := 3;
  write('TESTFA2 ');                    { TsEc = smallest T1 elbow coordinates }
  Obst := false;                        { TIEc = largest  ..   ..       ..     }
                                        { TsTc = smallest ..  tip       ..     }
  Ts[1] := T1s;                         { TITc = largest  ..   ..       ..     }
  TI[1] := T1I;
  if S[1]=G[1] then
    begin
    Ts[2] := S[2]; TI[2] := G[2];
    R1 := 0; R2 := 1;
    end
  else
    begin
    R1 := (Ts[1]-S[1])/(G[1]-S[1]);     { assume angular interpolation of T3 }
    R2 := (TI[1]-S[1])/(G[1]-S[1]);
    end;
  FindT2(Ts[1],Ts[2],Se,Ge,TsEc);
  FindT2(TI[1],TI[2],Se,Ge,TIEc);
  V3 := G[3]-S[3];
  Ts[3] := S[3]+R1*V3;
  TI[3] := S[3]+R2*V3;
  Convert3(Ts,TsEc,TsTc);
  Find_Point(TsEc,TsTc,C,Psf); { find the closest points to the object on the robot }
  Convert3(TI,TIEc,TITc);
  Find_Point(TIEc,TITc,C,PIf); { arms at the positions either side of the obstacle  }
  Find_Point(Psf,PIf,C,Pjf);
  Modulus(C,Pjf,CPjf,temp,ModCPjf);
  Intmax := ModCPjf-(Rad+Safety+Partrad[1]);
  writeln('The forearm has an interference of ',Intmax);
  if Intmax <= 0 then begin
    SF := (5*safety+Rad+Partrad[1])/ModCPjf;
    G := TI;
    Closestpart := 1;
{   writeln('Ts=',Ts[1],Ts[2],Ts[3]);
    writeln('TI=',TI[1],TI[2],TI[3]);
    writeln('the closest point on the forearm initially is');
    writeln('Psf=',Psf[X],Psf[Y],Psf[Z]);
    writeln('the closest point on the forearm finally is');
    writeln('PIf=',PIf[X],PIf[Y],PIf[Z]);
    writeln('The closest point on the forearm to the obstacle is');}
    writeln('Pjf=',Pjf[X],Pjf[Y],Pjf[Z]);
    end;
  { set the goal coords to those at the end of an intersecting sphere }
    for Partnum := 2 to Numpart do
      begin
      FindPos(Ts,TsTc,Poss,Partnum);
      FindPos(TI,TITc,PosI,Partnum);
      Find_point(Poss,PosI,C,Pjftemp);
      Modulus(C,Pjftemp,CPjftemp,temp,ModCPjf);
      Int := ModCPjf-(Partrad[Partnum]+Rad+Safety);
      writeln('Part ',Partnum,' has an interference of ',Int);
      if (Int < 0) and (Int < Intmax) then begin
        Intmax := Int;
        Pjf := Pjftemp;
```

```
      CPjf := CPjftemp;
      Closestpart := Partnum;
      SF := (5*safety+Partrad[Partnum]+Rad)/ModCPjf;
      writeln('The closest point on the gripper is');
      writeln(Pjftemp[X],Pjftemp[Y],Pjftemp[Z]);
      end;
    end;
  if Intmax <= 0 then begin
    Obst := true; end;
  writeln('Obst2=',Obst);
  keypressed;
end;
{-------------------------------------------------------------------------}
procedure Testfa(var Sph : Rangesph; var Rad : Rangerad; var N : integer;
                 var S, G : Degree;  var Se, Ge : Coord;
                 var Obst : boolean; var SLvalT1 : RangeSL);

{ Testpath tests to see if there are any obstacles in the robots path between
the set of coordinates given to it by S and G. It returns the point Pc which
is avoiding the closest obstacle  }

var loop1, Sphnum, Closestpart, Part : integer;
    C, CPjf, Pjf, ClosestPjf, ClosestCPjf, Tempc : Coord;
    T1s, T1l, Ta, Tb, SF, ModSpjf, Temp, ClosestT1, DelT1 : real;
    Obst2, Outofrange : boolean;

begin
  writeln('TESTFA testing a path between ');
  writeln('     ',S[1],S[2],S[3]);
  writeln('and ',G[1],G[2],G[3]);
  Obst := false; ClosestT1 := 999;
  loop1 := 1;
  C := Sph[loop1];
  while loop1 <= N do begin
    writeln('testing for a collision with sphere',C[x],C[y],C[z]);
    Obst2 := false;
    T1s := SLvalT1[loop1,1];
    T1l := SLvalT1[loop1,2];
    Ta := S[1]; Tb := G[1];
    Rangefind(Ta,Tb,T1s,T1l,Outofrange);
    if not Outofrange then
      Testfa2(C,Pjf,CPjf,Se,Ge,Rad[loop1],SF,T1s,T1l,S,G,Obst2,Part);
    if Obst2 then
      begin
      invtan(Pjf[X],Pjf[Y],temp);
      DelT1 := abs(temp-S[1]);
      if DelT1<ClosestT1 then
        begin
        Obst := true;
        ClosestT1 := DelT1;
        ClosestCPjf := CPjf;
        ClosestPart := Part;
        ClosestPjf := Pjf;
        Sphnum := loop1;
        end;
      end;
    loop1 := loop1+1;
    C := Sph[loop1];
    end; { while }
  if Obst then
```

```
    begin
    Avoidobs(Sph[Sphnum],ClosestPjf,ClosestCPjf,Se,Ge,SF,S,G,Closestpart);
    end;
  keypressed;
end;

{-------------------------------------------------------------------------}
procedure Stepgoal(var Elbdata : Rangeelb; var Stepnum, G : integer;
                   var Pcurrent, Go : Degree; var T3sg : real);
{ Calculate T3sg (step goal) }
var loop1 : integer;
    temp, Totalr, StepPcent : real;
begin
  write('STEPGOAL ');
  { calculate the total rotation of T1 and T2 }
  Totalr := abs(Elbdata[Stepnum+1,1]-Pcurrent[1])+
            abs(Elbdata[Stepnum+1,2]-Pcurrent[2]);
  temp := Totalr;
  for loop1 := Stepnum+2 to G do
    Totalr := Totalr+abs(Elbdata[loop1,1]-Elbdata[loop1-1,1])
                    +abs(Elbdata[loop1,2]-Elbdata[loop1-1,2]);
  { calculate the % of the total path in this step }
  if temp = 0 then StepPcent := 0
  else StepPcent := temp/Totalr;
  T3sg := (Go[3]-Pcurrent[3])*StepPcent+Pcurrent[3];
  {writeln('G=',G,' Stepnum=',Stepnum);
  writeln('the total rotation involved is ',Totalr);
  writeln('StepPcent=',StepPcent,' temp=',temp);
  writeln('T3 goal for the end of the step is ',T3sg);}
  keypressed;
end;
{-------------------------------------------------------------------------}
procedure Fapath2(var Sph : Rangesph; var Elbdata : Rangeelb; var Rad : Rangerad;
                  var N, E, CountRc : integer; var Robcoor : Rangerob;
                  var SLvalT1 : RangeSL; var St, Go : Degree);

var Pcurrent, Pg : Degree;
    PcurrEC, PgEC : Coord;          { elbow coordinates }
    Obst, Obst2 : boolean;
    T3sg : real;
    loop1, loop2, Stepnum : integer;
begin
  write('FAPATH2');
  RobCoor[0] := St;
  CountRC := 0;
{ T3 at the begining of a step is T3begin }
  Pcurrent := St;
  Convert2(St,PcurrEC);
  writeln('PcurrEC=',PcurrEC[X],PcurrEC[Y],PcurrEC[Z]);
  for Stepnum := 0 to E-1 do
    begin
    writeln('Stepnum=',Stepnum);
    repeat
      CountRc := CountRc+1;
      Stepgoal(Elbdata,Stepnum,E,Pcurrent,Go,T3sg);
      Pg[1] := Elbdata[Stepnum+1,1]; Pg[2] := Elbdata[Stepnum+1,2];
      Pg[3] := T3sg;
      Convert2(Pg,PgEC);
      writeln('PgEC=',PgEC[X],PgEC[Y],PgEC[Z]);
      Testfa(Sph, Rad, N, Pcurrent, Pg, PcurrEC, PgEC, Obst, SLvalT1);
```

```
      if Obst = true then
        repeat
          Testfa(Sph, Rad, N, Pcurrent, Pg, PcurrEC, PgEC, Obst2, SLvalT1);
          writeln('P=',Pg[1],Pg[2],Pg[3]);
          writeln('Obst2=',Obst2);
        until Obst2 = false;
      Pcurrent := Pg;
      PcurrEC := PgEC;
      Robcoor[CountRC] := Pg;
    until Obst = false;
    end;
  Keypressed;
end;
{-------------------------------------------------------------------------}
procedure Fapath(var Sph : Rangesph; var Elbdta : Rangeelb; var Rad : Rangerad;
    var N, E : integer; var Startpoint, Goalpoint : Coord;
      var SLvalT1 : RangeSL; var Interp : RangeInt; var Countint : integer);
{ Calculates the path for the forearm given the path of the upper arm }
var loop1, CountRobc, Rev : integer;
    tempC : Coord;
    St, Go : Degree;
    Offset : real;
    Robcoor : Rangerob;
begin
  writeln('------------------------------ FAPATH ------------------------------');
  writeln;
  writeln('Startpoint=',Startpoint[X],Startpoint[Y],Startpoint[Z]);
  writeln('Goalpoint= ',Goalpoint[X],Goalpoint[Y],Goalpoint[Z]);
  for loop1 := 0 to E do
    writeln('Elbdta[',loop1,']= ',Elbdta[loop1,1],Elbdta[loop1,2]);

{ Calculate St and Go }
  Rev := 1; Offset := 51;
  tempC := Startpoint; Convtobs(tempC,Offset,Rev); Convert(tempC,Zeroi,St);
  tempC := Goalpoint;  Convtobs(tempC,Offset,Rev); Convert(tempC,Zeroi,Go);

  Fapath2(Sph,Elbdata,Rad,N,E,CountRobc,Robcoor,SLvalT1,St,Go);
  for loop1 := 0 to CountRobc do
    writeln('Robcoor[',loop1,']= ',Robcoor[loop1,1],
                Robcoor[loop1,2],Robcoor[loop1,3]);
  Countint := 0;
  SMdata(Robcoor,CountRobc,Countint,Interp);
  keypressed;
end;
{-------------------------------------------------------------------------}
procedure Readdata(var Sphere : Rangesph; var Radius : Rangerad;
                   var N : integer);
{ This reads obstacle data from disk }

var FILENAME : string[12];
    DATA     : file of real;
    Offset : real;
    loop1, Rev, G : integer;
  begin
    Filename := 'Sphere.dta';
    writeln('This is the sphere data');
    assign(DATA,FILENAME); reset(DATA);
    N := 0;
    while not EOF(DATA) do
      begin
```

```
    N := N+1;
    read(DATA,Sphere[N,x],Sphere[N,y],Sphere[N,z],Radius[N]);
    writeln(Sphere[N,x],Sphere[N,y],Sphere[N,z],Radius[N]);
    end;   (* while *)
  close(DATA);
  keypressed;
end;
{-------------------------------------------------------------------------}
procedure Expobs(var Sph : RangeSph; var Rad : Rangerad;
                         var N : integer);
{ Expand obstacle set in order to take account of the elbow thickness }
var SinA, CosA, A, E, ModOC, SqModOC, temp1, temp2 : real;
    loop1, loop2 : integer;
    tempC1, tempC2 : Coord;
begin
  E := 58;
  writeln('EXPOBS the expanded obstacles are :-');
{ calculate whether the sphere obstructs the range of the elbow }
  for loop1 := 1 to N do begin
    SqmodOC := sqr(Sph[loop1,X])+sqr(Sph[loop1,Y])+sqr(Sph[loop1,Z]);
    ModOC := sqrt(sqmodOC);
    temp2 := sqrt(sqr(L1)+sqr(E));
{ if the sphere is out of range then move all spheres down one }
    if (ModOC-Rad[loop1])>temp2 then begin
      writeln('This Sphere is out of range so ignore');
      for loop2 := loop1 to N-1 do begin
        Sph[loop2] := Sph[loop2+1];
        Rad[loop2] := Rad[loop2+1]; end;
      N := N-1; loop1 := loop1-1; end
    else begin
      temp1 := sqrt(sqr(Rad[loop1])+SqmodOC);
      writeln('ModOC=',ModOC,' temp1=',temp1,' L1=',L1);
      if (abs(ModOC-L1)<Rad[loop1]) or (abs(L1-temp1)<Rad[loop1]) then begin
        writeln('expanding this obstacle');
{ move all the spheres up one }
        tempC2 := Sph[loop1+1]; temp2 := Rad[loop1+1];
        for loop2 := loop1+1 to N do begin
          tempC1 := tempC2; temp1 := temp2;
          tempC2 := Sph[loop2+1]; temp2 := Rad[loop2+1];
          Sph[loop2+1] := tempC1; Rad[loop2+1] := temp1; end;
        N := N+1;
{ calculate new Obstacle }
        Invtan(L1,E,A);
        SinA := sin(A); CosA := cos(A);
        Sph[loop1+1,X] := Sph[loop1,X]*cosA+Sph[loop1,Y]*sinA;
        Sph[loop1+1,Y] := -Sph[loop1,X]*sinA+Sph[loop1,Y]*cosA;
        Sph[loop1+1,Z] := Sph[loop1,Z];
        loop1 := loop1+1; end;
      end;          { else }
    end;            { for, loop1 }
  for loop1 := 1 to N do
    writeln('Sph[',loop1,']=',Sph[loop1,X],Sph[loop1,Y],Sph[loop1,Z]);
  Keypressed;
end;
{-------------------------------------------------------------------------}
procedure RangeCalc(var Sph : Rangesph; var Rad : Rangerad;
                    var N : integer; var SLvalT1 : RangeSL);
{ Calculate the range of T1 in which the robot can collide with the obstacle }
var loop1 : integer;
    Rot, D, Dist1, Dist2, DelTheta1, DelTheta2 : real;
```

```
    C : Coord;
begin
  writeln('RANGECALC');
  for loop1 := 1 to N do
    begin
    C := Sph[loop1];
    Invtan(Sph[Loop1,X],Sph[Loop1,Y],Rot);
    if Rot > Pix3o2 then Rot := Pix2-Rot;
    D := sqrt(sqr(C[x])+sqr(C[y]));
    Dist1 := Rad[loop1]+88;
    Dist2 := Rad[loop1]+38;
    DelTheta1 := Invsin(Dist1/D);
    DelTheta2 := Invsin(Dist2/D);
    SLvalT1[loop1,1] := rot-DelTheta1;
    SLvalT1[loop1,2] := rot+DelTheta2;
    writeln('SLvalT1[',loop1,']',SLvalT1[loop1,1],SLvalT1[loop1,2]);
    end;
  keypressed;
end;
{-------------------------------------------------------------------------}
procedure Delay1(var A : integer);
var loop1, loop2 : integer;
    R1 : real;
begin
  for loop2 := 1 to A do
    begin
    for loop1 := 1 to 66 do
      R1 := sin(pi/2);
    end;
  keypressed;
end;
{-------------------------------------------------------------------------}
procedure Randcoord(var Workpiece : RangeWkP; var WkPieceN, Area : integer);
var Clr : boolean;
    P : Coord;
begin
  write('RANDCOORD ');
  WkpieceN := WkpieceN+1;
  if WkpieceN=6 then WkpieceN := 1;
  Area := Area+1; if Area = 4 then Area := 1;
  repeat
    case Area of
1:    begin
      P[X] := 250+Random(200);
      P[Y] := 250+Random(200);
      P[Z] := -530; end;
2:    begin
      P[X] := -250-Random(200);
      P[Y] := 250+Random(200);
      P[Z] := -530; end;
3:    begin
      P[X] := -100+Random(200);
      P[Y] := 550+Random(100);
      P[Z] := -60; end;
      end; { case }
    Clr := true;
    for loop1 := 1 to 5 do begin
      if (abs(P[X]-Workpiece[loop1,X]) < 45) and
                        (abs(P[Y]-Workpiece[loop1,Y]) < 115) then
        Clr := false;
```

```
      writeln(Workpiece[loop1,X],Workpiece[loop1,Y],Workpiece[loop1,Z]);
      end;
  until Clr;
{ fix values for a test }
  if Area = 3 then Area := 1;
  if Area = 1 then begin
    P[X] := 350; P[Y] := 350; P[Z] := -530; end
  else begin P[X] := -250; P[Y] := 250; P[Z] := -530; end;
  WkpieceN := -1;
{ end of test code }
  Workpiece[WkpieceN] := P;
  writeln('Area=',Area,' WkpieceN=',WkpieceN);
  writeln('P=',P[X],P[Y],P[Z]);
end;
{-------------------------------------------------------------------------------}
procedure Pick(var St, Go : Coord; var Sr, Gr : real;
            var Initialp, Finalp : RangeApp; var Countinit, Countfin : integer);

var loop1, loop2, Rev : integer;
    Offset : real;
    S, G, tempC : Coord;
    Rangetest, Gp : boolean;
    T : RangeT;
    R : array[1..8] of RangeR;
begin
  write('PICK ');
  writeln('St=',S[X],S[Y],S[Z]);
  writeln('Go=',G[X],G[Y],G[Z]);
  Offset := 51; Rev := 1;
  S := St; Convtobs(S,Offset,Rev);
  G := Go; Convtobs(G,Offset,Rev);
  Convert(S,Zeroi,T[4]);
  S[Z] := S[Z]+10;
  Convert(S,Zeroi,T[3]);
  S[Z] := S[Z]+40;
  Convert(S,Zeroi,T[2]);
  S[Z] := S[Z]+50;
  Convert(S,Zeroi,T[1]);
  Convert(G,Zeroi,T[8]);
  G[Z] := G[Z]+10;
  Convert(G,Zeroi,T[7]);
  G[Z] := G[Z]+40;
  Convert(G,Zeroi,T[6]);
  G[Z] := G[Z]+50;
  Convert(G,Zeroi,T[5]);
  for loop1 := 1 to 8 do
    begin
    for loop2 := 1 to 3 do begin
      {writeln('T[',loop1,',',loop2,'] ',T[loop1,loop2]);}
      R[loop1,loop2] := T[loop1,loop2]; end;
    R[loop1,4] := (Pix3o2-T[loop1,2])-T[loop1,3];
    end;
  for loop1 := 1 to 4 do
    R[loop1,5] := (Pio2-T[loop1,1])+Sr;
  for loop1 := 5 to 8 do
    R[loop1,5] := (Pio2-T[loop1,1])+Gr;
  Gp := false;
  outputd2(Initialp[1],R[1],Gp,Rangetest);
  outputd2(Initialp[2],R[2],Gp,Rangetest);
  outputd2(Initialp[3],R[3],Gp,Rangetest);
```

```
  outputd2(Initialp[4],R[4],Gp,Rangetest);
  Gp := true;
  outputd2(Initialp[5],R[4],Gp,Rangetest);
  outputd2(Initialp[6],R[3],Gp,Rangetest);
  outputd2(Initialp[7],R[2],Gp,Rangetest);
  outputd2(Initialp[8],R[1],Gp,Rangetest);
  outputd2(Finalp[1],R[5],Gp,Rangetest);
  outputd2(Finalp[2],R[6],Gp,Rangetest);
  outputd2(Finalp[3],R[7],Gp,Rangetest);
  outputd2(Finalp[4],R[8],Gp,Rangetest);
  Gp := false;
  outputd2(Finalp[5],R[8],Gp,Rangetest);
  outputd2(Finalp[6],R[7],Gp,Rangetest);
  outputd2(Finalp[7],R[6],Gp,Rangetest);
  outputd2(Finalp[8],R[5],Gp,Rangetest);
  Countinit := 8; Countfin := 8;
  keypressed;
end;
{-------------------------------------------------------------------------------}
procedure Calctip(var Gc : Coord; var Tc : Coord);
var A, T1, Tg, ModGcxy : real;
begin
  write('CALCTIP ');
  ModGcxy := sqrt(sqr(Gc[X])+sqr(Gc[Y]));
  invtan(Gc[X],Gc[Y],Tg);
  A := invsin(7/ModGcxy);
  T1 := Tg+A;
  Tc[X] := Gc[X]+44*sin(T1)-13*cos(T1);
  Tc[Y] := Gc[Y]-13*sin(T1)-44*cos(T1);
  Tc[Z] := Gc[Z]+120;
  writeln('Tc=',Tc[X],Tc[Y],Tc[Z]);
end;
{-------------------------------------------------------------------------------}
begin
  clrscr;
  writeln('              ----------   Mainrpf   ----------');
  Readdata(Sphere,Radius,N);
  Goal[0] := Goal[1];
  Sph1 := Sphere; Sph2 := Sphere;
  Rad1 := Radius;
  N1 := N; N2 := N;
  Expobs(Sph1,Rad1,N1);

{ Transform Sphere data to obstacles as seen for the upper arm }
  writeln(' The Converted Obstacles for the upper arm are');
  Rev := 1; Offset := 105;
  for loop1 := 1 to N1 do begin
    Convtobs(Sph1[loop1],Offset,Rev);
    Rad1[loop1] := Rad1[loop1]+58;
    writeln('Sph1[',loop1,']=',Sph1[loop1,X],Sph1[loop1,Y],Sph1[loop1,Z],
            ' Rad1=',Rad1[loop1]); end;

{ Transform Sphere data to obstacles as seen for the forearm }
  Offset := 51;
  writeln(' The Converted Obstacles for the forearm are');
  for loop1 := 1 to N2 do begin
    Rad2[loop1] := Radius[loop1];
    Convtobs(Sph2[loop1],Offset,Rev);
    writeln('Sph2[',loop1,']=',Sph2[loop1,X],Sph2[loop1,Y],Sph2[loop1,Z],
            ' Rad2=',Rad2[loop1]); end;
```

```
  Rangecalc(Sph2,Rad2,N2,SLvalT1);

  tempC[X] := 300;
  tempC[Y] := 200;
  tempC[Z] := -530;
  for loop1 := 1 to 5 do begin
    tempC[X] := tempC[X]+50;
    Workpiece[loop1] := tempC;
    tempC[Z] := -430;
    Calctip(tempC,WkpConv[loop1]); end;
  TipS[X] := 400; TipS[Y] := 200; TipS[Z] := -100;
  Sr := 0; Gr := 0;
  loop1 := 20;
  WkpieceN := 1; Area := 1;
  repeat
  Randcoord(Workpiece,WkpieceN,Area);
    Calctip(Workpiece[WkpieceN],TipG);
    Routep1(Sph1,Elbdata,Rad1,N1,E,TipS,WkpConv[WkpieceN]);
    Fapath(Sph2,Elbdata,Rad2,N2,E,TipS,WkpConv[WkpieceN],SLvalT1,MvP,Countmvp);
    tempC := WkpConv[WkpieceN]; tempC[Z] := WkpConv[WkpieceN,Z]-100;
    Pick(tempC,TipG,Sr,Gr,Initialp,Finalp,Countinit,Countfin);
    TipG[Z] := TipG[Z]+100;
    Routep1(Sph1,Elbdata,Rad1,N1,E,WkpConv[WkpieceN],TipG);
    Fapath(Sph2,Elbdata,Rad2,N2,E,WkpConv[WkpieceN],TipG,SLvalT1,Interp,Countint);
    WkpConv[WkpieceN] := TipG;
    TipS := TipG;
    readln;
    Outputd(Initialp,Finalp,Mvp,Interp,Countinit,Countfin,Countmvp,Countint);
    {Delay(loop1);}
    keypressed;
  until false;
end.
```

APPENDIX D

Storedata program listing

```
PROGRAM STOREDTA;

Const L1 = 386;
      L2 = 376;

TYPE RANGE    = 0..21;
     COORDs   = (X,Y,Z);
     Coord    = array[Coords] of real;
     RANGESPH = ARRAY[RANGE,COORDs] OF REAL;
     RANGERad = ARRAY[RANGE] OF REAL;
     Degree = array[1..3] of real;
     Elb = array[1..2] of real;

VAR  N, G       : INTEGER;
     RUNNING,Ok: BOOLEAN;
     Sph, Goal : RANGESPH;
     Rad       : RANGERad;
     loop1, Ans, Num, Rev : integer;
     Offset : real;
     tempC : Coord;
     tempD : Degree;

{-------------------------------------------------------------------------}
procedure Keypressed;
var B:Byte;
begin
  B := Port[$0DA];
  B:=B and 02;
  if B=2 then read
end;
{-------------------------------------------------------------------------}
procedure Readdata(var Sph, G : Rangesph; var Rad : Rangerad;
                   var N, G : integer);
{ This reads obstacle data, and Goalpoint data from disk }

var FILENAME : string[12];
    DATA     : file of real;
    Offset : real;
    Rev : integer;

  begin
    writeln('reading data from file Sphere.dta');
    Filename := 'Sphere.dta';
    assign(DATA,FILENAME); reset(DATA);
    N := -1; G := -1;
    while not EOF(DATA) do
      begin
      N := N+1;
      read(DATA,Sph[N,x],Sph[N,y],Sph[N,z],Rad[N]);
```

```
      writeln(Sph[N,x],Sph[N,y],Sph[N,z],Rad[N]);
      end;   (* while *)
    Close(DATA);
    writeln('reading data from file Goal.dta');
    Filename := 'Goal.dta';
    assign(DATA,FILENAME); reset(DATA);
    while not EOF(DATA) do
      begin
      G := G+1;
      read(DATA,Goal[G,x],Goal[G,y],Goal[G,z]);
      writeln(Goal[G,x],Goal[G,y],Goal[G,z]);
      end;   (* while *)
  close(DATA);
end;

(*-------------------------------------------------------------------------*)
procedure WRITEGOAL(var GOAL : RANGESPH; var G : integer);

var FILENAME : string[12];
    I        : integer;
    DATAFILE : file of real;

begin
  Filename := 'Goal.Dta';
  assign(DATAFILE,FILENAME);
  rewrite(DATAFILE);
  for I := 0 to G do
    begin
    write(DATAFILE,GOAL[I,X],GOAL[I,Y],GOAL[I,Z]);
    end;
  close(DATAFILE);
end;
{-------------------------------------------------------------------------}
procedure WRITEDATA(var SPHERE : RANGESPH; var RADIUS : RANGERad;
                    var N : integer);
{ write data in sphere and Radius to a file called ' Sphere.Dta '. }
var FILENAME : string[12];
    I        : integer;
    DATAFILE : file of real;
begin
  Filename := 'Sphere.Dta';
  assign(DATAFILE,FILENAME);
  rewrite(DATAFILE);
  for I := 0 to N do
    write(DATAFILE,SPHERE[I,X],SPHERE[I,Y],SPHERE[I,Z],RADIUS[I]);
  close(DATAFILE);
end;
{-------------------------------------------------------------------------}

begin
  Clrscr; writeln('*************** STORE DATA ******************');
  N := -1; G := -1;
  RUNNING := TRUE;
  WHILE RUNNING DO
    begin
    Clrscr; writeln('*************** STORE DATA ******************');
    writeln('Choose one of the following options');
    writeln('1. Enter the coordinates of an obstacle sphere (in mm)');
    writeln('2. List the sphere coordinates');
    writeln('3. Edit a sphere');
```

```
    writeln('4. Write sphere data to a file');
    writeln('5. Enter a goalpoint');
    writeln('6. List goalpoints');
    writeln('7. Edit a goalpoint');
    writeln('8. Write goalpoint data to a file');
    writeln('9. Readdata');
    writeln('10. Stop');
    READLN(ANS);
    CASE ANS OF
    1:BEGIN
      N:=N+1;
      writeln('Enter the coordinates of the sphere center (',N,')');
      readln(Sph[N,x],Sph[N,y],Sph[N,z]);
      writeln('Enter the radius');
      readln(Rad[N]);
      end;
    2:begin
      for loop1 := 0 to N do
        writeln(Sph[loop1,x],Sph[loop1,y],
        Sph[loop1,z],Rad[loop1]);
      readln;
      end;
    3:begin
      writeln('What number of sphere');
      readln(Num);
      writeln('Enter the coordinates of the sphere center (',Num,')');
      readln(Sph[Num,x],Sph[Num,y],Sph[Num,z]);
      writeln('Enter the radius');
      readln(Rad[Num]);
      end;
    4:Writedata(Sph, Rad, N);
    5:begin
      G := G+1;
      writeln('Enter the coordinates of goalpoint(',G,')');
      readln(GOAL[G,X],GOAL[G,Y],GOAL[G,Z]);
      end;
    6:begin
      for LOOP1 := 0 to G do
        writeln(GOAL[LOOP1,X],GOAL[LOOP1,Y],GOAL[LOOP1,Z]);
      readln;
      END;
    7:begin
      writeln('What number of goalpoint');
      readln(Num);
      writeln(Goal[Num,x],Goal[Num,y],Goal[Num,z]);
      readln(Goal[Num,x],Goal[Num,y],Goal[Num,z]);
      end;
    8:WRITEGOAL(GOAL,G);
    9:begin Readdata(Sph,Goal,Rad,N,G); read; end;
    10:Running:=false;
    END;      (*WHILE*)
  end;        (*case*)
END.
```

APPENDIX E

Espace program listing

```
program Espace;

const L1 : integer = 385;
      L2 : integer = 376;
      Off1 : integer = 105;
      Off2 : integer = 54;
      Off3 : integer = 22;
      Upprad : integer = 58;
      Forrad : integer = 45;
      Griprad : integer = 89;
      Pix3o2 : real = 4.71238898;
      Pio2 : real = 1.570796326;
      Pix2 : real = 6.283185307;
      forearmtest : boolean = true;
      upperarmtest : boolean = false;
      first : boolean = true;
      second : boolean = false;

type Range = 1..10;
     Coords = (x,y,z);
     Coord = array[Coords] of real;
     Rangesph = array[Range] of Coord;
     Rangerad = array[Range] of real;
     Degree = array[1..3] of real;
     rangelim = array[1..3] of integer;
     rangetrig = array[0..36] of real;
     info1 = record
       code : byte;
       end;
{                  code :    bit              0                1
                             1           position clear      obstacle
                             2           position clear    new obstacle
                             4              untested       forearm tested
                             8              untested       upper arm tested
                             16             offlist           onlist
                             32
                             64
                             128                                        }

      info2 = record
        tcost : real;
        x, y, z : byte;
        end;

      Rangenode = array[0..36,0..18,5..36] of info1;
      Rangelist = array[1..1000] of info2;
      rangelist2 = array [1..1000] of info2;

var Node : Rangenode;
    List : Rangelist;
```

```
    sinval, cosval : rangetrig;
    tempD, T : Degree;
    lowlim, highlim : rangelim;
    T1, T2, T3 : integer;
    B1, E1, E2, Tip1, Tip2, P, tempC : Coord;
    T1s, T1l, Dt1temp, T1temp, temp, A, B, ModB1C,
    Dt, OCxy, OC, ModP, sinT1, cosT1, Phy, countvol : real;
    Numonlist, T1Gvs, T1Gvl,
    loop1, loop2, loop3, loopsph, N : integer;
    Sphere : Rangesph;
    Radius : Rangerad;
    testtype, expansiontype : boolean;
{-------------------------------------------------------------------------}
procedure Keypressed;
var B:Byte;
begin
  B := Port[$0DA];
  B:=B and 02;
  if B=2 then read
end;
{-------------------------------------------------------------------------}
procedure Modulus(var A,B,AB : Coord; var Sqmod, Modul : real);

  begin
    AB[X] := B[X]-A[X];
    AB[Y] := B[Y]-A[Y];
    AB[Z] := B[Z]-A[Z];
    Sqmod := sqr(AB[X])+sqr(AB[Y])+sqr(AB[Z]);
    Modul := sqrt(Sqmod);
  end;
{-------------------------------------------------------------------------}
function Invsin(Sina : real) : real;
var temp : real;
begin
  if ((Sina < 1e-4) and (Sina > -1e-4)) then
    temp := Sina
  else begin
    temp := arctan(sqrt(1/(1/sqr(Sina)-1)));
    if Sina<0 then temp := -temp; end;
  Invsin := temp;
end;
{-------------------------------------------------------------------------}
function Invcos(var Cosa : real) : real;
var temp : real;
begin
  if ((Cosa < 1e-4) and (Cosa > -1e-4)) then
    temp := Pi/2-Cosa
  else
    temp := arctan(sqrt(1/sqr(Cosa)-1));
  if cosa < 0 then temp := Pi-temp;
  Invcos := temp;
end;
{-------------------------------------------------------------------------}
procedure Invtan(var x, y, ans : real);
begin
  if abs(x)<2e-6 then                { This calculates the angle of rotation }
    begin                            { of a line from 0,0 to x,y. If x<2e-6  }
    if abs(y)<2e-6 then              { and y<2e-6 then angle = 0.            }
      ans := 0                       { X is assumed = 0 if x<2e-6.           }
    else if y>0 then
```

```
      ans := Pi/2
    else ans := 1.5*pi;
    end
  else if x>0 then
    begin
    if y>=0 then
      ans := arctan(y/x)
    else ans := 2*Pi+arctan(y/x)
    end
  else ans := Pi+arctan(y/x);
end;
{-------------------------------------------------------------------------}
procedure Convert(var P : Coord; var T : Degree);
var L3, ModXY, SqmodXY, temp, ModEP, D, SqL3, CosT3, SinB, B : real;
    TempC, E : Coord;
begin                                  { This procedure calculates the robot   }
  Invtan(P[x],P[y],T[1]);              { coordinates for when the tip of the   }
  SqmodXY := sqr(P[x])+sqr(P[y]);      { robot arm is at P                     }
  ModXY := sqrt(SqmodXY);              { T3 is always less than Pi.            }
  Invtan(ModXY,P[z],D);
  SqL3 := SqmodXY+sqr(P[z]);
  L3 := sqrt(SqL3);
  if L3 > L1+L2 then begin
    writeln('ERROR Goalpoint out of range');
    if ((D < pio2) and (D > 0)) then begin
      T[2] := D;
      T[3] := Pi; end
    else begin
      T[2] := 0;
      E[x] := L1*cos(T[1]);
      E[y] := L1*sin(T[1]);
      E[z] := 0;
      modulus(E,P,tempC,temp,ModEP);
      T[3] := Pi+invsin(P[z]/ModEP);
      end;
    end
  else begin
    CosT3 := (L1*1.0*L1+L2*1.0*L2-SqL3)/(2.0*L1*L2);
    T[3] := Invcos(CosT3);
    SinB := (L2)*sin(T[3])/sqrt(SqL3);
    B := Invsin(SinB);
    T[2] := B+D;
    if ((T[2] < 0) or (T[2] > Pix3o2)) then begin
      T[2] := 0;
      E[x] := L1*cos(T[1]);
      E[y] := L1*sin(T[1]);
      E[z] := 0;
      modulus(E,P,tempC,temp,ModEP);
      T[3] := Pi+invsin(P[z]/ModEP);
      end
    else begin
      if T[1] > Pi then begin
        if (T[1] > Pix2-0.01) and (T[1] < Pix2+0.01) then
          T[1] := 0
        else if T[1] < Pi+0.01 then
          T[1] := Pi
        else writeln('********** ERROR T1 out of range *************');
        end;
      if T[2] > Pio2 then begin
        if T[2] < Pio2+0.01 then
```

```
          T[2] := Pio2
        else writeln('ERROR5');
        end;
      end;
    end;
  writeln('Converted','T[1]=',T[1],' T[2]=',T[2],' T[3]=',T[3]);
  {keypressed;}
end;
{-------------------------------------------------------------------------}
procedure Convtobs(var C : Coord; var T : Degree);
{ this converts the real coordinates of obstacles to the coordinates as }
{ the robot sees them }
var SqmodOE, ModOE, temp, Beta, A, L1xy, Offset : real;
begin
  Offset := 29;
  SqmodOE := sqr(C[X])+sqr(C[Y]);
  ModOE := sqrt(SqModOE);
  L1xy := sqrt(SqmodOE-sqr(Offset));
  Invtan(C[X],C[Y],Beta);
  Invtan(Offset,L1xy,temp);
  A := Beta-temp;
  C[X] := C[X]-Offset*cos(A);
  C[Y] := C[Y]-Offset*sin(A);
  Convert(C,T);
end;
{-------------------------------------------------------------------------}
procedure Readdata(var Sphere : Rangesph; var Radius : Rangerad;
                   var N : integer);
{ This reads obstacle data from disk }

var FILENAME : string[12];
    DATA     : file of real;
    Offset : real;
    loop1, Rev, G : integer;
  begin
    Filename := 'Sphere.dta';
    writeln('This is the sphere data');
    assign(DATA,FILENAME); reset(DATA);
    N := 0;
    while not EOF(DATA) do
      begin
      N := N+1;
      read(DATA,Sphere[N,x],Sphere[N,y],Sphere[N,z],Radius[N]);
      writeln(Sphere[N,x],Sphere[N,y],Sphere[N,z],Radius[N]);
      end;   { while }
    close(DATA);
    {keypressed;}
  end;
{-------------------------------------------------------------------------}
procedure Find_Point(var S, G, C, Pj : Coord);
{This finds the point Pj on the line SG closest to C}
var Sc_Sg, ModSg, temp, r : real;
    Sc, Sg : Coord;

begin
  {write('FIND_POINT '); }              { find the point P on Sq such }
  Sc[X] := C[X]-S[X];                 { that Cp and Sp meet at a right angels}
  Sc[Y] := C[Y]-S[Y];
  Sc[Z] := C[Z]-S[Z];
  Modulus(S,G,Sg,Temp,ModSg);
```

```
  Sc_Sg := Sc[X]*Sg[X]+Sc[Y]*Sg[Y]+Sc[Z]*Sg[Z];
  r := Sc_Sg/ModSg;
  if r >= ModSg then            { is the point at the tip of the forearm? }
    Pj := G
  else if r <= 0 then                { is the point at the elbow }
    Pj := S
  else begin                          { the point is in between }
    temp := r/ModSg;
    Pj[X] := S[X]+temp*Sg[X];
    Pj[Y] := S[Y]+temp*Sg[Y];
    Pj[Z] := S[Z]+temp*Sg[Z]; end;      { hello says Sarah }
  {writeln('Pj[X]=',Pj[X],' Pj[Y]=',Pj[Y],' Pj[Z]=',Pj[Z]);}
  {keypressed;}
end;
{-------------------------------------------------------------------------}
procedure Pullofflist(var lx, ly, lz : integer; var List : rangelist;
                      var Numonlist : integer);
begin
  if numonlist<1 then writeln('******* ERROR IN pullofflist ***********');
  lx := list[numonlist].x;
  ly := list[numonlist].y;
  lz := list[numonlist].z;
  numonlist := numonlist-1;
end;
{-------------------------------------------------------------------------}
procedure putonlist2(var List : rangelist; var Numonlist : integer;
                     var lx, ly, lz : integer);
begin
  write(numonlist,' ');
  numonlist := numonlist+1;
  List[numonlist].x := lx;
  List[numonlist].y := ly;
  List[numonlist].z := lz;
  if numonlist > 999 then writeln('*********  list size to big ***********');
end;
{-------------------------------------------------------------------------}
procedure testpos(var C : Coord; var rad : real; var lx, ly, lz : integer;
                  var Node : rangenode; var List : rangelist;
                  var Numonlist, loopsph : integer;
                  var testtype, expansiontype : boolean;
                  var lowlim, highlim : rangelim; var sinval, cosval : rangetrig);
var forearmuntested, upperarmuntested, postest : boolean;
    B1, E1, E2, Tip1, Tip2, P, tempC : Coord;
    T1s, T1l, Dt1temp, T1temp, temp, cosphy, sinphy, OCxy, OC, ModP : real;
    T1Gvs, T1Gvl, loop1, phy  : integer;

begin
  forearmuntested := true; upperarmuntested := true;
  if (Node[lx,ly,lz].code and 4) = 4 then forearmuntested := false;
  if (Node[lx,ly,lz].code and 8) = 8 then upperarmuntested := false;
  if (forearmuntested and (testtype = forearmtest)) or
     (upperarmuntested and (testtype = upperarmtest)) then
    begin
    if (expansiontype=first) or ((node[lx,ly,lz].code and 1)=0) then
      begin
      postest := false;

{ find important points on the real robot }
      sinT1 := sinval[lx]; cosT1 := cosval[lx];
      B1[x] := Off1*sinval[lx];              { B1 = base of the upper arm }
```

```
      B1[y] := -Off1*cosval[lx];
      B1[z] := 0;
      E1[x] := B1[x]+L1*cosval[lx]*cosval[ly];    { E1 = top of the upper arm }
      E1[y] := B1[y]+L1*sinval[lx]*cosval[ly];
      E1[z] := B1[z]+L1*sinval[ly];

      if testtype = forearmtest then begin
{ set to forearm tested }
        Node[lx,ly,lz].code := Node[lx,ly,lz].code or 4;
        E2[x] := E1[x]-off2*sinval[lx];          { E2 = base of the forearm }
        E2[y] := E1[y]+off2*cosval[lx];
        E2[z] := E1[z];
        phy := ly+lz-36;
        if phy<0 then begin
          phy := -phy;
          sinphy := -sinval[phy];
          cosphy := cosval[phy];
          end
        else begin
          sinphy := sinval[phy];
          cosphy := cosval[phy]; end;
        Tip1[x] := E2[x]+L2*cosval[lx]*cosphy;   { Tip1 = top of the forearm }
        Tip1[y] := E2[y]+L2*sinval[lx]*cosphy;
        Tip1[z] := E2[z]+L2*sinphy;
        Tip2[x] := Tip1[x]-off3*sinval[lx];       { Tip2 = centre of the gripper }
        Tip2[y] := Tip1[y]+off3*cosval[lx];       {      motor sphere            }
        Tip2[z] := Tip1[z];
     {writeln('B1=',B1[x],',',B1[y],',',B1[z]);
     writeln('E1=',E1[x],',',E1[y],',',E1[z]);
     writeln('E2=',E2[x],',',E2[y],',',E2[z]);
     writeln('Tip1=',Tip1[x],',',Tip1[y],',',Tip1[z]);
     writeln('Tip2=',Tip2[x],',',Tip2[y],',',Tip2[z]);}

 { decide whether the robot intersects the sphere }
        find_point(E2,Tip1,C,P);
        modulus(P,C,tempC,temp,ModP);
        if (ModP < (rad+Forrad)) then postest := true;
        Modulus(Tip2,C,tempC,temp,ModP);
        if (ModP < (rad+Griprad)) then postest := true;
        end

 { upper arm test }
      else begin
 { set to upper arm tested }
        Node[lx,ly,lz].code := Node[lx,ly,lz].code or 8;
        find_point(B1,E1,C,P);
        modulus(P,C,tempC,temp,ModP);
        if (ModP < (rad+Upprad)) then postest := true;
        if postest = true then
          for loop1 := lowlim[3] to highlim[3] do
          Node[lx,ly,loop1].code := Node[lx,ly,loop1].code or 2;
        end;

       if postest = true then begin
         writeln(lx,',',ly,',',lz);
         putonlist2(list,numonlist,lx,ly,lz);
         Node[lx,ly,lz].code := node[lx,ly,lz].code or 2; end;
       end;
     end;
   {keypressed;}
```

```
end;
{-------------------------------------------------------------------------}
procedure expand(var C : Coord; var rad : real; var l1, l2, l3 : integer;
                 var list : rangelist; var numonlist, loopsph : integer;
                 var node : rangenode; var lowlim, highlim : rangelim;
                 var testtype, expansiontype : boolean;
                 var sinval, cosval : rangetrig);
var e1, e2, e3 : integer;
    exptype : integer;
begin
  exptype := 1;
  if expansiontype = first then
    exptype := 2;
  e1 := l1-exptype;
  e2 := l2;
  e3 := l3;
  if e1>=lowlim[1] then
    testpos(C,rad,e1,e2,e3,node,list,numonlist,loopsph,testtype,expansiontype,
            lowlim,highlim,sinval,cosval);
  e1 := e1+2*exptype;
  if e1<=highlim[1] then
    testpos(C,rad,e1,e2,e3,node,list,numonlist,loopsph,testtype,expansiontype,
            lowlim,highlim,sinval,cosval);
  e1 := e1-exptype;
  e2 := e2-exptype;
  if e2>=lowlim[2] then
    testpos(C,rad,e1,e2,e3,node,list,numonlist,loopsph,testtype,expansiontype,
            lowlim,highlim,sinval,cosval);
  e2 := e2+2*exptype;
  if e2<=highlim[2] then
    testpos(C,rad,e1,e2,e3,node,list,numonlist,loopsph,testtype,expansiontype,
            lowlim,highlim,sinval,cosval);
  e2 := e2-exptype;
  if testtype = forearmtest then begin
    e3 := e3-exptype;
    if e3>=lowlim[3] then
      testpos(C,rad,e1,e2,e3,node,list,numonlist,loopsph,testtype,
              expansiontype,lowlim,highlim,sinval,cosval);
    e3 := e3+2*exptype;
    if e3<=highlim[3] then
      testpos(C,rad,e1,e2,e3,node,list,numonlist,loopsph,testtype,
              expansiontype,lowlim,highlim,sinval,cosval);
    end;
  {keypressed;}
end;
{-------------------------------------------------------------------------}
procedure WRITEDATA(var node : rangenode; var lowlim, highlim : rangelim);
{ write data in sphere and Radius to a file called ' Sphere.Dta '. }
var FILENAME : string[12];
    loop1, loop2, loop3 : integer;
    b1, b2 : byte;
    DATAFILE : file of byte;
begin
  b1 := 0;
  b2 := 64;
  Filename := 'Node.Dta';
  assign(DATAFILE,FILENAME);
  rewrite(DATAFILE);
  for loop1 := lowlim[1] to highlim[1] do begin
    for loop2 := lowlim[2] to highlim[2] do begin
```

```
      for loop3 := lowlim[3] to highlim[3] do begin
        if node[loop1,loop2,loop3].code and 1 = 1 then
          write(DATAFILE,b1)
        else write(DATAFILE,b2);
        end;
      end;
    end;
  close(DATAFILE);
end;
{-------------------------------------------------------------------------}
procedure fill(var C : Coord; var rad : real;
               var list : rangelist; var numonlist, loopsph : integer;
               var node : rangenode; var lowlim, highlim : rangelim;
               var testtype : boolean; var sinval, cosval : rangetrig);
var loop1, loop2, loop3, T1, T2, T3, numonlist2 : integer;
    tempi, zlow1, zlow2, zlow3, zhigh1, zhigh2, zhigh3 : integer;
    i1, i2, i3, l1, l2, l3 : integer;
    list2 : rangelist2;
    bottomrange, expansiontype, edge : boolean;
begin
  zlow1 := 255; zlow2 := 255; zlow3 := 255;
  zhigh1 := 0; zhigh2 := 0; zhigh3 := 0;

  writeln('find bottom range');
  bottomrange := false;
  i1 := lowlim[1]-1;
  repeat
    i2 := lowlim[2]-1;
    i1 := i1+1;
    repeat
      i3 := lowlim[3]-1;
      i2 := i2+1;
      repeat
        i3 := i3+1;
        if node[i1,i2,i3].code and 2 = 2 then
          bottomrange := true;
      until (bottomrange or (i3 = highlim[3]));
    until (bottomrange or (i2 = highlim[2]));
  until (bottomrange or (i1 = highlim[1]));
  writeln('the first point is ',i1,',',i2,',',i3);

  writeln('develope list');
  numonlist2 := 0;
  zlow1 := i1;
  repeat
    repeat
      repeat
        if node[i1,i2,i3].code and 2 = 2 then begin
          if (i2 < zlow2) then
            zlow2 := i2;
          if (i3 < zlow3) then
            zlow3 := i3;
          if (i1 > zhigh1) then
            zhigh1 := i1;
          if (i2 > zhigh2) then
            zhigh2 := i2;
          if (i3 > zhigh3) then
            zhigh3 := i3;
          if numonlist2 > 995 then
            writeln('************* list length is to small **************');
```

```
        numonlist2 := numonlist2+1;
        list2[numonlist2].x := i1;
        list2[numonlist2].y := i2;
        list2[numonlist2].z := i3;
        end;
      i3 := i3+1;
    until i3>=highlim[3]+1;
    i2 := i2+1;
    i3 := lowlim[3];
  until i2>=highlim[2]+1;
  i1 := i1+2;
  i2 :=  lowlim[2];
until i1>=highlim[1]+1;

writeln('fill');
writeln('numonlist2=',numonlist2);
for loop1 := 1 to numonlist2 do begin
  l1 := list2[loop1].x;
  l2 := list2[loop1].y;
  l3 := list2[loop1].z;
  if node[l1+2,l2,l3].code and 2 = 2 then    { +x         }
    node[l1+1,l2,l3].code := node[l1+1,l2,l3].code or 14;   { 14 = tested and blocked }
  if node[l1,l2+2,l3].code and 2 = 2 then    { +y         }
    node[l1,l2+1,l3].code := node[l1,l2+1,l3].code or 14;
  if node[l1,l2,l3+2].code and 2 = 2 then    { +z         }
    node[l1,l2,l3+1].code := node[l1,l2,l3+1].code or 14;
  if node[l1+2,l2+2,l3].code and 2 = 2 then  { +x +y       }
    node[l1+1,l2+1,l3].code := node[l1+1,l2+1,l3].code or 14;
  if node[l1+2,l2-2,l3].code and 2 = 2 then  { +x -y       }
    node[l1+1,l2-1,l3].code := node[l1+1,l2-1,l3].code or 14;
  if node[l1+2,l2,l3+2].code and 2 = 2 then  { +x +z       }
    node[l1+1,l2,l3+1].code := node[l1+1,l2,l3+1].code or 14;
  if node[l1+2,l2,l3-2].code and 2 = 2 then  { +x -z       }
    node[l1+1,l2,l3-1].code := node[l1+1,l2,l3-1].code or 14;
  if node[l1,l2+2,l3+2].code and 2 = 2 then  { +y +z       }
    node[l1,l2+1,l3+1].code := node[l1,l2+1,l3+1].code or 14;
  if node[l1,l2+2,l3-2].code and 2 = 2 then  { +y -z       }
    node[l1,l2+1,l3-1].code := node[l1,l2+1,l3-11].code or 14;
  if node[l1+2,l2+2,l3+2].code and 2 = 2 then { +x +y +z }
    node[l1+1,l2+1,l3+1].code := node[l1+1,l2+1,l3+1].code or 14;
  if node[l1+2,l2+2,l3-2].code and 2 = 2 then { +x +y -z }
    node[l1+1,l2+1,l3-1].code := node[l1+1,l2+1,l3-1].code or 14;
  if node[l1+2,l2-2,l3+2].code and 2 = 2 then { +x -y +z }
    node[l1+1,l2-1,l3+1].code := node[l1+1,l2-1,l3+1].code or 14;
  if node[l1+2,l2-2,l3-2].code and 2 = 2 then { +x -y -z }
    node[l1+1,l2-1,l3-1].code := node[l1+1,l2-1,l3-1].code or 14;
end;

{ get centre points out of list }
  numonlist := 0;
  for loop1 := 1 to numonlist2 do begin
    edge := false;
    l1 := list2[loop1].x; l2 := list2[loop1].y; l3 := list2[loop1].z;
    if node[l1+2,l2,l3].code and 3 = 0 then edge := true;
    if node[l1-2,l2,l3].code and 3 = 0 then edge := true;
    if node[l1,l2+2,l3].code and 3 = 0 then edge := true;
    if node[l1,l2-2,l3].code and 3 = 0 then edge := true;
    if node[l1,l2,l3+2].code and 3 = 0 then edge := true;
    if node[l1,l2,l3-2].code and 3 = 0 then edge := true;
    if edge then begin
```

```
    numonlist := numonlist+1;
    list[numonlist] := list2[loop1]; end;
  end;
for loop1 := 1 to numonlist do
  list2[loop1] := list[loop1];
numonlist2 := numonlist;

writeln('find edge values');
writeln('upperarm');
expansiontype := second;
testtype := upperarmtest;
repeat
  pullofflist(T1,T2,T3,list,numonlist);
  expand(Sphere[loopsph],Radius[loopsph],T1,T2,T3,list,numonlist,loopsph,
         node,lowlim,highlim,testtype,expansiontype,sinval,cosval);
until numonlist = 0;

writeln('forearm');
testtype := forearmtest;
for loop1 := 1 to numonlist2 do
  list[loop1] := list2[loop1];
numonlist := numonlist2;
tempi := 0;
repeat
  tempi := tempi+1;
  pullofflist(T1,T2,T3,list,numonlist);
  expand(Sphere[loopsph],Radius[loopsph],T1,T2,T3,list,numonlist,loopsph,
         node,lowlim,highlim,testtype,expansiontype,sinval,cosval);
until numonlist = 0;
tempi := tempi-numonlist2;
writeln('number of edge points found =',tempi);

writeln('setting clear and untested status for next obstacle');
zlow1 := zlow1-2;
if zlow1 < lowlim[1] then zlow1 := lowlim[1];
zlow2 := zlow2-2;
if zlow2 < lowlim[2] then zlow2 := lowlim[2];
zlow3 := zlow3-2;
if zlow3 < lowlim[3] then zlow3 := lowlim[3];
zhigh1 := zhigh1+2;
if zhigh1 > highlim[1] then zhigh1 := highlim[1];
zhigh2 := zhigh2+2;
if zhigh2 > highlim[2] then zhigh2 := highlim[2];
zhigh3 := zhigh3+2;
if zhigh3 > highlim[3] then zhigh3 := highlim[3];

{ set new obstacles to clear untested }
  for loop1 := zlow1 to zhigh1 do begin
    for loop2 := zlow2 to zhigh2 do begin
      for loop3 := zlow3 to zhigh3 do begin
        if Node[loop1,loop2,loop3].code and 2 = 2 then
          Node[loop1,loop2,loop3].code := 1
        else Node[loop1,loop2,loop3].code := Node[loop1,loop2,loop3].code and 1;
        end; end; end;
  writeln('end of fill');
end;
{-------------------------------------------------------------------------}
begin
  DT := Pi/36;
{ set up cos and sin values }
```

```
  for loop1 := 0 to 36 do begin
    temp := loop1*DT;
    sinval[loop1] := sin(temp);
    cosval[loop1] := cos(temp);
    end;

  lowlim[1] := 0; lowlim[2] := 0; lowlim[3] := 5;
  highlim[1] := 36; highlim[2] := 18; highlim[3] := 36;
  Readdata(Sphere,Radius,N);
  for loop1 := lowlim[1] to highlim[1] do begin
    for loop2 := lowlim[2] to highlim[2] do begin
      for loop3 := lowlim[3] to highlim[3] do begin
        Node[loop1,loop2,loop3].code := 0;
        end; end; end;
{ set the limits of T1, T2, T3 and the increment DT }
{ T1 :  0  to 180, increment of 5 degrees }
{ T2 :  0  to  90, increment of 5 degrees }
{ T3 :  25 to 180, increment of 5 degrees }
{    Nodes of the graph come every 5 degrees ie 5, 15, 25 etc.
The size of the graph is 37x19x32 = 22496    }

  writeln('start algorithm');
  readln;
  for loopsph := 1 to N do
    begin
    writeln('Graphing sphere number ',loopsph);
    writeln(sphere[loopsph,x],sphere[loopsph,y],sphere[loopsph,z],
            radius[loopsph]);
    tempC := sphere[loopsph];
    expansiontype := first;

{ find the forearm positions which hit the obstacle }
    testtype := forearmtest;
    writeln('Finding blocked forearm positions');
{ find the robot position when the center of the gripper is at the center of
  the sphere }
    convtobs(tempC,T);
{ convert T to nearest coordinates }
    T1 := trunc(0.5+T[1]/DT);
    T2 := trunc(0.5+T[2]/DT);
    T3 := trunc(0.5+T[3]/DT);
{ write out value in degrees }
    tempD[1] := T1*5; tempD[2] := T2*5; tempD[3] := T3*5;
    writeln('The first point is ',tempD[1],',',tempD[2],',',tempD[3]);
    numonlist := 0;
    testpos(Sphere[loopsph],Radius[loopsph],T1,T2,T3,node,list,
       numonlist,loopsph,testtype,expansiontype,lowlim,highlim,sinval,cosval);
    if numonlist = 1 then begin
      repeat
        pulloffist(T1,T2,T3,list,numonlist);
        expand(Sphere[loopsph],Radius[loopsph],T1,T2,T3,list,numonlist,loopsph,
               node,lowlim,highlim,testtype,expansiontype,sinval,cosval);
      until numonlist = 0; end
    else
      writeln('this obstacle is out of range');

{ find the upper arm positions which hit the obstacle }
    testtype := upperarmtest;
    writeln('Finding blocked upper arm positions');
{ is the sphere in range? }
```

```
    temp := sqr(sphere[loopsph,x])+sqr(sphere[loopsph,y]);
    OC := sqrt(temp+sqr(sphere[loopsph,z]));
    if (OC-Radius[loopsph]) < 451 then begin
      OCxy := sqrt(temp);
      invtan(sphere[loopsph,x],sphere[loopsph,y],B);
      temp := Off1;
      invtan(temp,OCxy,A);
      T1 := trunc(0.5+(pio2+B-A)/DT);
      sinT1 := sin(T1); cosT1 := cos(T1);
      B1[x] := Off1*sinT1;                { B1 = base of the upper arm }
      B1[y] := -Off1*cosT1;
      B1[z] := 0;
      modulus(B1,sphere[loopsph],tempC,temp,ModB1C);
      temp := invsin(sphere[loopsph,z]/ModB1C);
      T2 := trunc(0.5+temp/DT);
      T3 := 10;
      writeln('T1=',T1,' T2=',T2);
      numonlist := 0;
      testpos(Sphere[loopsph],Radius[loopsph],T1,T2,T3,node,list,
            numonlist,loopsph,testtype,expansiontype,lowlim,highlim,sinval,cosval);
      if numonlist = 1 then begin
        repeat
          pulloffliist(T1,T2,T3,list,numonlist);
          expand(Sphere[loopsph],Radius[loopsph],T1,T2,T3,list,numonlist,loopsph,
               node,lowlim,highlim,testtype,expansiontype,sinval,cosval);
        until numonlist = 0; end
      else
        writeln('this obstacle is out of range');
      end
    else writeln('sphere out of range of upper arm');

    fill(Sphere[loopsph],Radius[loopsph],list,numonlist,loopsph,
          node,lowlim,highlim,testtype,sinval,cosval);
{ find the position of the upper arm }
    end;          { loopsph }
  writeln('STOP THE TIME');
  readln;
  countvol := 0;
  for loop1 := lowlim[1] to highlim[1] do begin
    for loop2 := lowlim[2] to highlim[2] do begin
      for loop3 := lowlim[3] to highlim[3] do begin
        if (Node[loop1,loop2,loop3].code and 1) = 1 then
          countvol := countvol+1;
        end; end; end;
  writeln('number of points is ',countvol);
  readln;
  writeln('*************   writeing data **************');
  Writedata(Node,lowlim,highlim);
end.
```

APPENDIX F

Graphsch program listing

```
program Graphsch;
type info1 = record
       code : byte;
       costg : byte;
       end;
{                code :    bit              0                1
                           1                              x predecessor
                           2                              y predecessor
                           4                              z predecessor
                           8          -ve predessor       +ve predecessor
                           16
                           32
                           64          obstacle           position clear
                           128         offlist            onlist
                  costg :  set to 1000 at start                          }

      info2 = record
        tcost : real;
        x, y, z : byte;
        end;

      Degree = array[1..3] of integer;
      Rangenode = array[0..36,0..18,5..36] of info1;
      Rangelist = array[1..300] of info2;

var s1, s2, s3, g1, g2, g3, loop1 : integer;
    node : rangenode;
    list : rangelist;
    imp, ok : boolean;
    lowlim, highlim : Degree;
{-------------------------------------------------------------------------}
procedure Keypressed;
var B:Byte;
begin
  B := Port[$0DA];
  B:=B and 02;
  if B=2 then read
end;
{-------------------------------------------------------------------------}
procedure Pullofflist(var x, y, z, numonlist : integer; var List : Rangelist);
var loop1 : integer;
    temp : real;
begin
  if numonlist = 0 then begin
    writeln('***** ERROR numonlist = 0 ********');
    temp := 1/0; end;
{  writeln('Pullofflist',x,',',y,',',z);}
  loop1 := 0;
  repeat
```

```
    loop1 := loop1+1;
  until ((x=List[loop1].x)and(y=List[loop1].y)and(z=List[loop1].z));
  while loop1 < numonlist do begin
    List[loop1] := List[loop1+1];
    loop1 := loop1+1; end;
  numonlist := numonlist-1;
  keypressed;
end;
{-------------------------------------------------------------------------}
procedure Putonlist(var x, y, z, numonlist, costg : integer; var costh : real;
                    var list : Rangelist);
var loop1, loop2 : integer;
    cost : real;
begin
  {writeln('Putonlist',x,',',y,',',z);}
  loop1 := 0;
  cost := costg+costh;
{ find the position on list for the new node }
  repeat
    loop1 := loop1+1;
  until ((list[loop1].tcost>cost)or(loop1>numonlist));
{ move all the others down then insert new node }
  numonlist := numonlist+1;
  loop2 := numonlist;
  while loop2 > loop1 do begin
    list[loop2] := list[loop2-1];
    loop2 := loop2-1;
    keypressed;
    end;              { while }
  list[loop1].x := x;
  list[loop1].y := y;
  list[loop1].z := z;
  list[loop1].tcost := cost;
  if numonlist>295 then writeln('***** LIST LENGTH INSUFFICIENT *****');  (* *)
  keypressed;
end;
{-------------------------------------------------------------------------}
procedure Expand2(var copen, cnext, x, y, z, gx, gy, gz, ncostg, numonlist :
                  integer; var list : Rangelist; var node : Rangenode);
var ncosth : real;
begin
  if (copen xor cnext) <> 8 then begin
    if node[x,y,z].code and 64 = 64 then begin
      if ncostg<node[x,y,z].costg then begin
        ncosth := sqr(gx-x)+sqr(gy-y)+sqr(gz-z);          (* *)
        node[x,y,z].costg := ncostg;
        if node[x,y,z].code and 128 = 128 then          { if on list }
          pulloffllist(x,y,z,numonlist,list);
        putonlist(x,y,z,numonlist,ncostg,ncosth,list);
        node[x,y,z].code := cnext+64+128;
        end; end; end;
  keypressed;
end;
{-------------------------------------------------------------------------}
procedure Expand(var opennodex, opennodey, opennodez, gx, gy, gz, numonlist : integer;
                 var list : Rangelist; var lowlim, highlim : degree;
                 var node : Rangenode);
var copen, cnext, x, y, z, ncostg : integer;
begin
  copen := node[opennodex,opennodey,opennodez].code and 15;
```

```
  x := opennodex-1;
  y := opennodey;
  z := opennodez;
  ncostg := node[opennodex,opennodey,opennodez].costg+1;
  cnext := 1;
  if x>=lowlim[1] then Expand2(copen,cnext,x,y,z,gx,gy,gz,ncostg,numonlist,list,node);
  x := x+2;
  cnext := 9;
  if x<=highlim[1] then Expand2(copen,cnext,x,y,z,gx,gy,gz,ncostg,numonlist,list,node);
  x := x-1; y := y-1;
  cnext := 2;
  if y>=lowlim[2] then Expand2(copen,cnext,x,y,z,gx,gy,gz,ncostg,numonlist,list,node);
  y := y+2;
  cnext := 10;
  if y<=highlim[2] then Expand2(copen,cnext,x,y,z,gx,gy,gz,ncostg,numonlist,list,node);
  y := y-1; z := z-1;
  cnext := 4;
  if z>=lowlim[3] then Expand2(copen,cnext,x,y,z,gx,gy,gz,ncostg,numonlist,list,node);
  z := z+2;
  cnext := 12;
  if z<=highlim[3] then Expand2(copen,cnext,x,y,z,gx,gy,gz,ncostg,numonlist,list,node);
  keypressed;                                    (* *)
end;
{-------------------------------------------------------------------------------}
procedure listpath(var sx, sy, sz, gx, gy, gz :integer;
                   var node : Rangenode; var list : rangelist);
var tcode, dir, loop1, loop2, listbot  : integer;
    R : array[1..5] of real;
    Dt : real;
begin
  writeln(gx,',',gy,',',gz);
  readln;
  listbot := 300;
  list[300].x := gx; list[300].y := gy; list[300].z := gz;
  Dt := Pi/36;
  repeat
    listbot := listbot-1;
    tcode := Node[gx,gy,gz].code and 15;
    if tcode > 8 then dir := -1
    else dir := 1;
    tcode := tcode and 7;
    if tcode = 1 then
      gx := gx+dir
    else if tcode = 2 then
      gy := gy+dir
    else gz := gz+dir;
    writeln(gx,',',gy,',',gz);
    list[listbot].x := gx; list[listbot].y := gy; list[listbot].z := gz;
    keypressed;
  until ((gx=sx)and(gy=sy)and(gz=sz));
  for loop1 := listbot to 300 do begin
    for loop2 := 1 to 10000 do;
    R[1] := (list[loop1].x)*Dt;
    R[2] := (list[loop1].y)*Dt;
    R[3] := (list[loop1].z)*Dt;
    R[1] := int((R[1]-3.195)*-283.3);
    R[2] := int((R[2]-2.72)*-299.2);
    R[3] := int((R[3]-0.4046)*316.4);
    R[4] := 500;
    R[5] := 736;
```

```
    write('%1');
    for loop2 := 1 to 4 do begin
      write(R[loop2]); end;
    writeln(R[5]);
    Keypressed;
    end;
  writeln('&');
end;
{-------------------------------------------------------------------------------}
procedure Readdata(var Node : Rangenode; var lowlim, highlim : Degree);
{ This reads Node data from disk }

var FILENAME : string[12];
    DATA     : file of byte;
    loop1, loop2, loop3 : integer;
  begin
    Filename := 'Node.dta';
    writeln('This is the sphere data');
    assign(DATA,FILENAME); reset(DATA);
    for loop1 := lowlim[1] to highlim[1] do begin
      for loop2 := lowlim[2] to highlim[2] do begin
        for loop3 := lowlim[3] to highlim[3] do begin
          read(DATA,Node[loop1,loop2,loop3].code);
          {writeln('node,',loop1,',',loop2,',',loop3,'=',Node[loop1,loop2,loop3].code);}
          Node[loop1,loop2,loop3].costg := 1000;
          keypressed;
          end;
        end;
      end;
    close(DATA);
    keypressed;
  end;
{-------------------------------------------------------------------------------}
procedure Gsearch(var sx, sy, sz, gx, gy, gz : integer;
                  var lowlim, highlim : degree;
                  var node : rangenode; var list : rangelist);
var loop1, loop2, loop3, opnx, opny, opnz, numonlist : integer;
begin
  opnx := sx; opny := sy; opnz := sz;
  Node[opnx,opny,opnz].costg := 0;
  node[opnx,opny,opnz].code := 192;
  numonlist := 1;
  list[1].x := sx; list[1].y := sy; list[1].z := sz; list[1].tcost := 15;
{ main program }
  clrscr;
  writeln('Graph Search');
  repeat
    Pullofflist(opnx,opny,opnz,numonlist,list);
{ set node to offlist }
    node[opnx,opny,opnz].code := node[opnx,opny,opnz] code and 127;
    expand(opnx,opny,opnz,gx,gy,gz,numonlist,list,lowlim,highlim,node);
    if Numonlist < 1 then writeln('******* ERROR no new opennode ********');
    opnx := list[1].x;
    opny := list[1].y;
    opnz := list[1].z;
    writeln('                    opennode ',opnx,',',opny,',',opnz);
  until ((opnx=gx)and(opny=gy)and(opnz=gz));                    (* *)
  Listpath(sx,sy,sz,gx,gy,gz,node,list);
  writeln('finished');
end;
```

```
{-------------------------------------------------------------------------}
begin
{ initialise variables }
  highlim[1] := 36; highlim[2] := 18 ; highlim[3] := 36;
  lowlim[1] := 0; lowlim[2] := 0; lowlim[3] := 5;
  repeat
  Readdata(Node,lowlim,highlim);
  for loop1 := 1 to 300 do
    list[loop1].tcost := 10000;
  writeln('Coord limits are (',lowlim[1],',',lowlim[2],',',lowlim[3],') to (',
          highlim[1],',',highlim[2],',',highlim[3],')');
  repeat
  repeat
  ok := true;
   writeln('enter cartesian coordinates of the start position');
   readln(s1,s2,s3);
   if (s1<lowlim[1])or(s1>highlim[1])or(s2<lowlim[2])or(s2>highlim[2])or
      (s3<lowlim[3])or(s3>highlim[3]) then ok := false;
  until ok;
  repeat
  ok := true;
    writeln('enter the cartesian coordinates of the goal position');
    readln(g1,g2,g3);
    if (g1<lowlim[1])or(g1>highlim[1])or(g2<lowlim[2])or(g2>highlim[2])or
      (g3<lowlim[3])or(g3>highlim[3]) then ok := false;
  until ok;
  imp := false;
    if node[s1,s2,s3].code = 0 then imp := true;
    if node[g1,g2,g3].code = 0 then imp := true;
    if imp then
      writeln('this path is impossible')
  until not imp;
  node[s1,s2,s3].costg := 0;
  gsearch(s1,s2,s3,g1,g2,g3,lowlim,highlim,node,list);
  until false;
end.
```