MPK: An Open Extensible Motion Planning Kernel

Ian Gipson, Kamal Gupta*
School of Engineering Science
Simon Fraser University
Burnaby, British Columbia V5A LS6
e-mail: gipson@sfu.ca, kamal@cs.sfu.ca

Michael Greenspan
National Research Council of Canada
Ottawa, Ontario, Canada
e-mail: Michael.Greenspan@nrc.ca

Received 27 November 1999; accepted 3 April 2001

The motion planning kernel (MPK) is a software system designed to facilitate development, testing, and comparison of robotic and geometric reasoning algorithms. Examples of such algorithms include automatic path planning, grasping, etc. The system has been designed to be open and extensible, so that new methods can be easily added and compared on the same platform. © 2001 John Wiley & Sons, Inc.

1. INTRODUCTION

The field of robot motion planning comprises a large body of academic research[1,2] and it remains an active area of investigation. Broadly speaking, it refers to the ability of a robot to plan its own motions in the presence of obstacles in the environment. The case when the environment is completely known a priori, is called model-based motion planning and has received the greatest attention. The basic approaches to model-based planning have been well characterized[2] and several practical algorithms have been proposed in the literature.[1] While many consider model-based motion planning to be a maturing field, little if any effort has gone into the comparison of such algorithms, and it remains a difficult task for a practitioner/researcher in the field to decide which algorithm solves a given problem best.

The majority of the research effort in the field has been focused on developing new representations and search methods, and the code artifacts

*To whom all correspondence should be addressed.
Contract grant sponsors: National Research Council of Canada (NRC), National Sciences and Engineering Research Council of Canada (NSERC), and MD Robotics.

tend to be of a home-grown variety, unsuitable for extensive reuse. In addition, although many researchers believe that aspects of this technology could be well utilized in industrial systems, there has been little direct industrial application of motion planning methods.[1] While this is a complex issue, one contributing factor has been a scarcity of effective software tools. Commercially available robot programming packages (IGRIP,[3] ROBCAD,[4] and SILMA[5]) tend to have some collision detection capabilities but hardly any motion planning functionality. ACT[6] does provide some motion planning capability, but it is geared toward a particular technique. In general, while these systems indeed provide robot programming environments, they tend to be of a proprietary nature where users are permitted little if any access to the source code. The underlying algorithm for a particular function (say collision detection) is generally invisible to the user and cannot be changed.

The motion planning kernel (MPK) is aimed at addressing these issues. It is a general software toolkit that facilitates the testing and benchmarking of various motion planning methods, as well as the design and implementation of new methods.

The system has been designed to be used and extended by many researchers. Rather than limiting users only to the functionality that we have explicitly provided, there are convenient ways to incorporate new components into the code. New collision detection algorithms could be added to the software as they are developed, as could new planning algorithms and geometric descriptions. Rather than attempting to anticipate and provide everything that all potential users might need, basic functionality has been provided, along with the ability for users to extend the system themselves. This "design for extensibility" is a primary benefit of the MPK approach.

The MPK contributes to the field of robotics and motion planning a general platform upon which to build programs and test robotic algorithms. It provides users a code framework that allows them to treat various robot specific operations as abstractions, similar to something of an application programming interface (API) level. The MPK combines modern object oriented software design with robotic programming concepts developed over the past 20 years to provide a toolkit that facilitates design, implementation and testing of robotic algorithms.

Additionally, writing code in the MPK framework allows users immediate access to a wide range of sample data that we have created. The MPK allows users to make use of previously created libraries of robots, geometry, and working environments, thereby simplifying the testing of methods and allowing for benchmarking to be performed on an equal footing. This ease of extensibility will allow for the addition of new motion planning algorithms as they are developed. Furthermore, we plan to make the system accessible over the Web, so that researchers/practitioners in dispersed locations can use the MPK for testing and sharing new motion planning algorithms.

The MPK is not yet a fully featured robotic programming environment. The current version primarily facilitates developing and implementing robot algorithms with a geometric flavor, such as motion planning. Features such as grasping of objects, force planning, and proper dynamic simulation are not yet implemented.

The MPK code library is written in object oriented C++. It uses only standard C++ and no platform dependant features, so it can be used by researchers on any standard hardware configurations.

This article is organized as follows. A survey of prior work is presented in Section 2. In Section 3, the major elements of the MPK system and their interrelationships are described in detail. The user interface and planned Web enabled feature is discussed in Section 4, a benchmarking example is shown in Section 5, and the article concludes in Section 6 with a summary and a discussion of future work.

2. RELATION TO PRIOR WORK

Development of a software toolkit for robotic programming is not a novel idea. As mentioned earlier, commercial toolkits exist for reasons similar to those stated above. Many have become rather dated, however, as throughout the software industry monolithic, closed form C language libraries have been replaced with more modular, object oriented systems, with C++ being the language of choice for many researchers. In this aspect, the object oriented nature and C++ interface of the MPK is a marked improvement over earlier systems.

The planned Web based interactive portion of the MPK also borrows from existing work. For example the Interactive Benchmark system presented in ref. 7 provides a system for evaluating a few algorithms for nonholonomic motion planning of car-like vehicles. Other similar online systems include *FixtureNET*,[8] a system for evaluating the problem of grasping parts via the Internet. Both of these systems constrain the problem to be solved to a very small subset of the motion planning field. They also severely limit the nature of the problem that can be specified because they constrain all problems to two dimensions, and they do not permit descriptions of new geometries to be uploaded from files. Users are forced to draw parts and obstacles themselves. Modular software packages for control aspects of robotics do exist[9,10]; however these do not address geometric motion planning aspects. In fact these packages would be complementary to the MPK. The University of Minnesota maintains an online library of geometric applications, although none of them are specifically related to robotics.[11] Also, many data structures and algorithms are borrowed from the LEDA library[12] maintained by Algorithmic Solutions.

3. CODE LIBRARY

Conceptually, the MPK code is divided into several interacting modules, which are listed in Table I. This division of the system corresponds to the generally accepted notion that motion planning can be broken down into some sort of search and sampling of the configuration space (C-space) of the robot. The search (or planning) mechanism effectively places the samples, and a collision detector acts as an evaluator, testing each sample for collision. This breakdown is consistent with the formulation presented in ref. 1.

A typical way in which a programmer would use the MPK would be to create a *universe* object and then populate it with robots and obstacles. During the testing stage of a new algorithm, populating the *universe* would be done via some of the preexisting sample data. Typically, only one robot would exist at a time, but nothing prevents the existence of multiple robots. A *collision detection* object would then be instantiated that used this *universe* and performed whatever internal optimizations it needed to make collision checking faster. Once a particular *collision detector* has been implemented, a planner can subsequently be instantiated that uses that *collision detector*, takes a start and goal configuration, and produces a path. Figure 1 illustrates the use-case scenario outlined above.

An example of code that would perform some simple planning tasks is shown in Figure 2. The example shows how easy it is for a programmer to incorporate MPK planning functionality into a program. Also shown is how various planners can be swapped between with ease.

The code example in Figure 2 serves to create a *universe* object and load a robot and some obstacles from a file. Two different motion planners, each using two different collision detection methods, are created. Start and goal configurations are set up for the planners, and planning is then executed. The planners produce paths consisting of a sequence of joint angles, which are output to the console window.

Table I. The modules of MPK.

Module	Function
Universe	Contains information about the robot and environment
Collision detector	Acts as an intermediary between the planner and the universe; responds to collision queries posed by the planner
Planner	Contains the algorithm for path planning that will be implemented using the collision detector
Geometry	Library of data structures and routines used by the universe to represent geometric data

3.1. Universe

The *universe* object, as the name implies, represents the entire physical world: both the robot(s) and other workspace objects. A user must first populate the *universe* before any other tasks can be accomplished. The *universe* in turn is composed of a collection of the abstract base class called *entity*. Both robots and obstacles are derived from *entity* as shown in Figure 3. *Entities* derive their location in

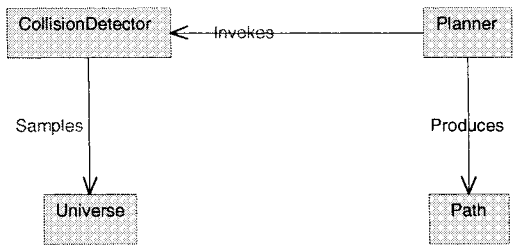

Figure 1. Program flow for a typical planning task.

```
void main()
{
    //load data into the universe object
    Universe theUniverse ;
    theUniverse.LoadRobotFile( "puma robot.txt" ) ;
    theUniverse.LoadObstacleFile( "workcell.wrl" ) ;

    //create some collision detectors
    CD_Vcollide collisionDetector1( theUniverse ) ;
    CD_Solid collisionDetector2( theUniverse ) ;

    //create some planners
    PL_Linear       planner1 ;
    PL_Sequential   planner2 ;

    //set the planners to use the collision detectors
    planner1.SetCollisionDetector(collisionDetector1) ;
    planner2.SetCollisionDetector(collisionDetector2) ;

    //set up the start and goal configurations    Configuration startConfig ;
    startConfig = theUniverse.CurrentConfig() ;
    Configuration goalConfig
    GoalConfig = theUniverse.CurrentConfig() ;
    goalConfig[ 0 ] = 0 ;
    goalConfig[ 1 ] = 90 ;
    planner1.SetStartConfig( startConfig ) ;
    planner1.SetGoalConfig( goalConfig ) ;
    planner2.SetStartConfig( startConfig ) ;
    planner2.SetGoalConfig( goalConfig ) ;

    //perform the planning
    bool p1Success = planner1.Plan() ;
    bool p2Success = planner2.Plan() ;

    //view the results
    if( p1Success == true )
    {
        planner1.GetPath().Output() ;
    }

    if( p2Success == true )
    {
        planner2.GetPath().Output() ;
    }
}
```

Figure 2. Complete planning example.

the three dimensional physical world from the position and orientation of the *frame* they are defined with respect to. As such, an *entity* possesses both a kinematic and a geometric component. Since every *entity* is defined with respect to a frame, if that frame moves, so does the *entity*. A frame represents a transformation frame in three-dimensional (3D) space. Refer to Figure 4 for a standard universal modeling language (UML) diagram outlining this relationship. In both diagrams, arrows with white triangles represent object inheritance, while arrows with diamonds represent object aggregation. Filled diamonds represent containment by value, while hollow diamonds represent containment by reference. A robot is composed of links described in Denavit–Hartenberg notation. *Link* objects are responsible for adjusting the values contained in a *frame* object.

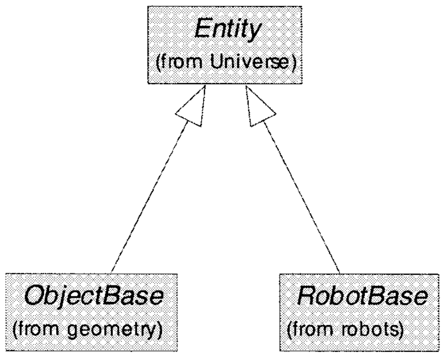

Figure 3. Both robots and obstacles derive from class entity.

Parts, obstacles, the work cell, and robots all inherit from the *entity* base class within the *universe*. Parts will be entities that the robot interacts with; items will be entities that the robot is supposed to grasp or assemble. Obstacles will be those entities that are immutable from the robot's per-

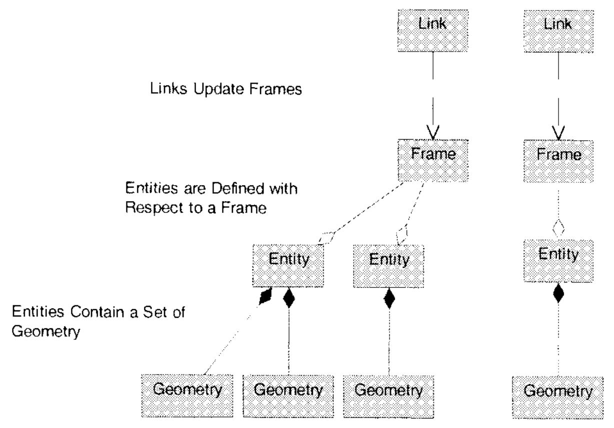

Figure 4. Relationship between links, frames, and entities.

spective. What is considered a part in one instance can be an obstacle in a different instance. Consider, for example, an assembly problem. The robot will grasp the first part and place it in the proper location. That part now becomes an obstacle when the robot is attempting to place the second part.

3.2. Geometry

To maintain a description of the robot and the environment, geometric data structures and functions are included in the MPK. Some of the objects associated with geometry derive from the class *entity* and as such can be added to the *universe* as obstacles or added to a link as link geometry. Other objects, like the *VRMLreader*, are accessory objects used to load geometry from files.

It is important to remember that the MPK is not meant to represent a fully featured computer-aided design (CAD) modeler, and as such the geometry portion has been kept deliberately simple, and users will not be able to interactively design complex geometry. The understanding is that quite a lot of good software exists for designing geometry (SolidWorks,[13] 3D Studio Max[14]). The philosophy of the MPK is to allow users to import geometry from a wide variety of CAD formats.

The current version of the MPK supports the following object types and will be expanded as time and demand for new objects permit:

1. Polyhedral surface meshes,
2. Spheres,
3. Line segments, and
4. Groups of the above named objects.

Future additions to the MPK may include solid models, as opposed to boundary representations, spherical decompositions of objects, and quadtrees and octrees as described in ref. 15.

3.3. Collision Detection

A planner's main interface to the environment is provided via a collision detector object. This interface allows the planner to access information about how the robot and the environment interact, generally termed a "collision query." Collision queries can be widely different, depending on the particular needs of a planner; however they generally are not overly complex. Several basic queries already implemented are:

- Point probe collision query;
- Line probe collision query (two types).

The point probe query specifies one complete configuration of the robot (i.e., a point in configuration space) and returns a true or false indication of whether that configuration is collision-free. The line probe query is actually an extremely simple local planner. It forms a straight line between two points in configuration space, and determines if this path can be traversed without collisions. The justification for including these particular queries is that they are the most basic and most widely used. More complex collision queries will be added in future revisions.

Collision detection is a huge field in itself, and the body of preexisting work is very large.[15] The most desirable manner of supporting collision detection is to permit interfacing with as many existing packages as possible and to allow the programmer or the user of the interactive system to choose the one that he or she feels is appropriate. The MPK can be used to evaluate the utility of different collision detection schemes in much the same way as it is used to evaluate the performance of motion planning algorithms.

The problem with allowing multiple existing collision detection code libraries to be used with the MPK is that they have a variety of different interfaces. Distilling the information from different collision detectors is a problem in itself, so to simplify the task of planner developer, a standard method of interfacing with the collision detection libraries was settled upon. For each distinct interface to a collision detection library that is possible, an interface class is implemented. An interface is an abstract class that cannot be instantiated, but exposes several functions that must be implemented in any class that inherits from it. This mechanism serves to isolate the planner from the implementation of the collision detection library. The planner only uses the interface class, and thus any class that inherits from that interface could also be used by that planner.

In Figure 5, a planner that requires interface1 can use either the I_collide[16] or the V_collide[17] libraries, while a planner that needs interface2 could only use V_collide. A planner requiring interface3 would be unable to use either of the collision detection libraries shown. The low-level collision detection objects shown in the diagram represent wrapper classes created for each of the third party libraries.

Collision detectors that are currently integrated into the system include:

1. A homegrown, simple collision detector,
2. V_collide, and
3. Solid.[18]

The simple homegrown collision detector is a crude scheme, intended to act as a baseline for comparing other algorithms. It performs a brute force intersection test between every pair of geometric objects that can collide in the system. There is no optimization performed as there is in the other systems. Future iterations of the MPK will include many other more sophisticated collision detection schemes including the one in ref. 19.

As mentioned earlier, extensibility is key in the MPK. One area in which researchers may extend the system is the addition of different collision detectors. To create a new collision detector, a researcher would have to create a class *NewCollisionDetector*. This class would derive from whatever interfaces were appropriate. The researcher would then alter the behavior of the constructor for this class—this is the function to which the *universe* object is passed. It is here that the researcher would perform much of the algorithmic work of the new collision detector. In addition, the researcher would have to implement the query functions that are to be supported, so that they take advantage of the optimizations.

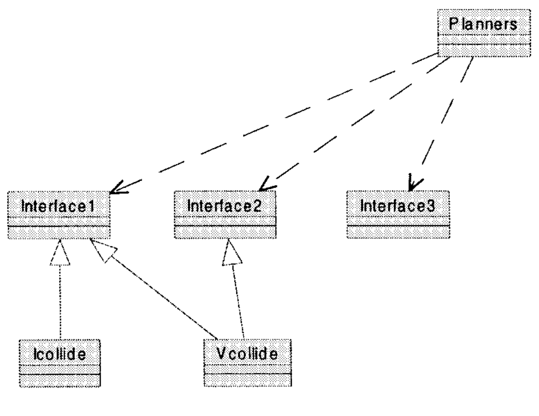

Figure 5. Relationship between planners, interfaces, and collision detectors.

3.4. Planners

The MPK system may be used to evaluate the performance of a wide array of motion planning algorithms. Programmers will develop their own algorithms within the MPK framework and add them to the system. However, the interactive MPK must have several ''baseline'' algorithms as a starting point as shown in Table II. As such we have implemented certain algorithms and plan to implement others, often with the help of other developers.

Implicit in the notion of a planner within the MPK framework is the notion of a task. The only task that a planner is currently capable of responding to is moving to a specific, fully specified goal configuration. The inclusion of more complex tasks is something that is being actively investigated. Examples include grasping planning, partially specified goal configurations, inverse kinematics, both point-to-point[25] and along a specific tool frame path, and manipulation planning,[26] etc.

The code structure of most motion planning algorithms is consistent with the object oriented nature of the MPK. A planner is an object that derives from the *PlannerBase* abstract base class. This parent class forces each planner object to implement several member functions. Chief among the

Table II. Planners implemented within the MPK.

Planner implemented	Comments
Ariadne's clew algorithm[20]	Implemented
Sequential planner[21]	Implemented
Linear planner (local planner)	Implemented
RPP[22]	Planned
Probabilistic roadmap[23]	Implemented
SANDROS[24]	Planned
A* planner	Implemented

member functions is the *Plan*( ) function that instructs the planner to examine its current task, and develop a sequence of motions to complete it. As an example, code is presented to illustrate the ease of developing a planner algorithm from within the MPK framework. Figure 6 contains the entire code for the linear planner algorithm, a simple local planner that plans a path between two completely specified configurations by discretizing a line segment between them in C-space into a finite number of points. If each of the points is collision free, then a path is said to exist along that line segment.

The code in Figure 6 represents the contents of the *Plan*( ) method in the object that implements the linear planner. This was the only method that needed to be implemented to write this new planner. All the construction and deconstruction, setting of start and goal configurations, etc. is done by the parent class.

4. USER INTERFACE AND WEB BASED SYSTEM

The planned primary architecture of the interactive MPK system consists of a front end client communicating with a server handling complex processing (Fig. 7). This could consist of a Web based client

```
path.AppendPoint( startConfig ) ;

const long int steps = 1000 ;
path.Clear() ;

//test all points from start to goal
Configuration current = startConfig ;
Configuration offset = goalConfig-startConfig;
for( long int i = 0; i <= steps; i++ )
{
        if
        (
collisionDetector->IsInterfering( current ) )
        {
                return false ;
        }
        path.AppendPoint( current ) ;
        current += ( offset * (1.0/steps ) ) ;
        }
return true ;
```

Figure 6. Code example for the linear planner.

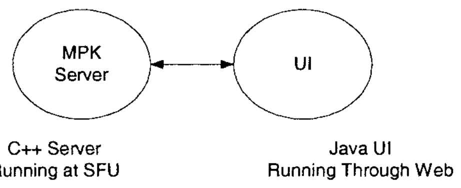

Figure 7. Example of the client server system architecture.

communicating with a powerful server, or the client and server could be linked together and run on the same machine. The framework is meant to be flexible. The front end will allow the user to design a robot and environment and to set start and goal configurations. Essentially it will allow the client user to specify a problem to be solved by a planner through the MPK system. Once the problem has been fully specified, it will be sent to the server for processing. When the server has completed this task, it will return the results.

For one client, we plan to use a Java front end because it is a cross-platform, cross-browser solution, providing a large base of people who can test out the MPK. C+ + was chosen for the server side implementation due to the large number of third party libraries that exist in C/C+ + that could be added to the server. A PC platform was chosen because this type of machine is becoming more and more common in research and in industrial applications and is a cost effective option. Communication between client and server is performed via TCP/IP sockets, and display of the robot in the Web based client applet uses Java3D.

The typical scenario in which a user would operate this application is shown in Figure 8. Designing the robot and the environment will be performed by selecting options from the menu or loading files. Specifying robot configurations for the planning task is done interactively using sliders and viewing the robot in a window. Results of planning are displayed to the user either as an animation or as a sequence of intermediate shadow images of the robot. As of now, a user interface has been imple-

1 Design a robot
2 Design an environment composed of obstacles
3 Formulate a planning task
4 Choose algorithms, and run a test
5 Evaluate the results

Figure 8. User workflow for the Web based application.

mented in C++ and integrated with the MPK library to provide a MS Windows application that mimics the planned Web based system without using the Internet. Figure 9 illustrates this user interface.

The example in Figures 10–12 shows a simple scenario in which a user designed a task for a manipulator, placed an obstacle in the way, and examined the results. The user then added an additional obstacle. Note the convenient intermediate shadow images.

5. BENCHMARKING DIFFERENT PLANNERS: AN ILLUSTRATION

To illustrate the benefits of the MPK as applied to benchmarking, we created a simple example problem and ran various planning algorithms on it. The statistic of interest was the total time taken to complete the plan. The algorithms that we used for the test include Ariadne's clew (ACA), a variant of the probabilistic road map algorithm known as Lazy PRM[27], and A*. The example problem was quite simple. It consisted of moving a 4 degree of freedom planar arm among cylindrical obstacles. 3D examples of motion planning can often be difficult to visualize and reproduce in print. This example was chosen because it is simple enough that it can be easily visualized in two dimensions.

Figure 13 illustrates the example used for benchmarking. The start configuration is in the upper right quadrant, and the goal configuration is in the lower left quadrant. The manipulator arm has four links and is constrained to the plane of the page. Collisions with the circular obstacles are not permitted.

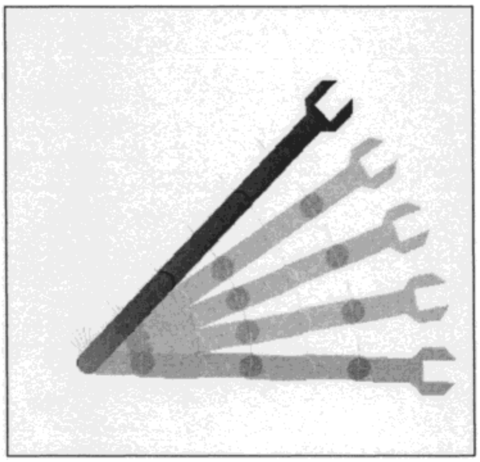
Figure 10. User has specified a start and a goal.

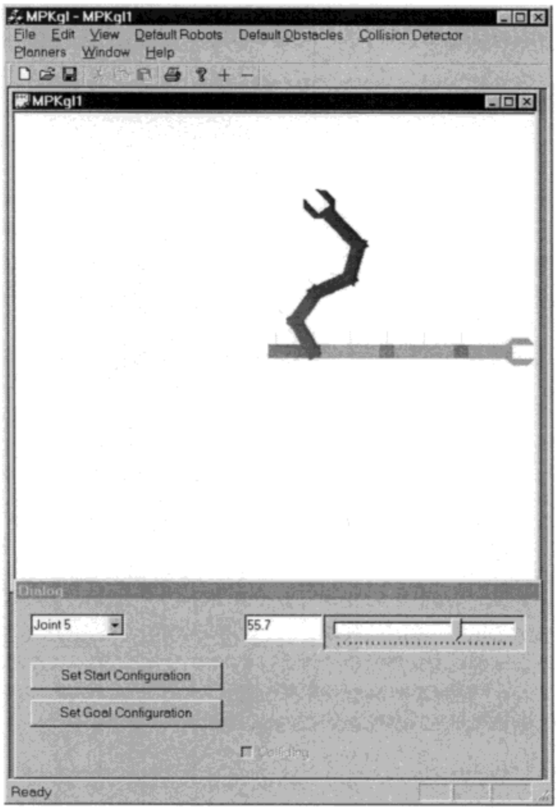

Figure 9. Movement of joints and selection of start and goal configurations.

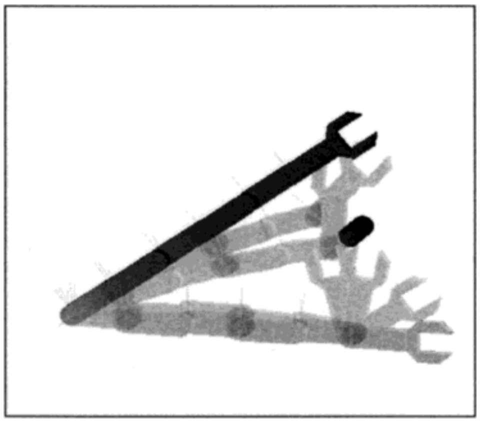
Figure 11. User added an obstacle that the robot must avoid.

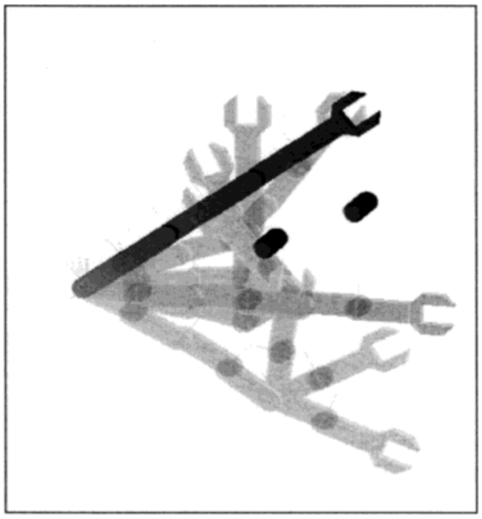

Figure 12. User added an additional obstacle to make the task progressively harder.

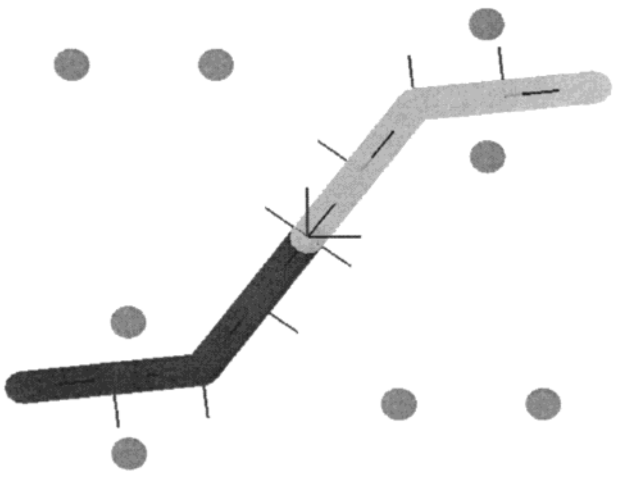

Figure 13. Four link robot benchmarking example.

Both ACA and PRM are randomized (nature and extent of randomization differ) algorithms. To properly benchmark these algorithms, many runs were performed. Figure 14 and also Table III show the minimum time and maximum time, represented by the endpoints of the vertical lines, and 33rd and 66th percentiles, represented by the top and bottom of the rectangle.

It appears that for this example problem, ACA is the fastest algorithm, although it also has the widest variation in completion times (Table III). PRM is slightly slower, but has less variation in completion times. A* is a deterministic algorithm which appears to perform worse than both the randomized schemes.

This analysis is meant to illustrate the comparative benchmarking abilities of the MPK and not as an in-depth analysis of the aforementioned algo-

Table III. Elapsed times for various planners.

Planner	Average time	Minimum time	Maximum time	Standard deviation
A*	87	87	87	0
PRM	34.86	31	37	2.54
ACA	24.25	6	66	19.69

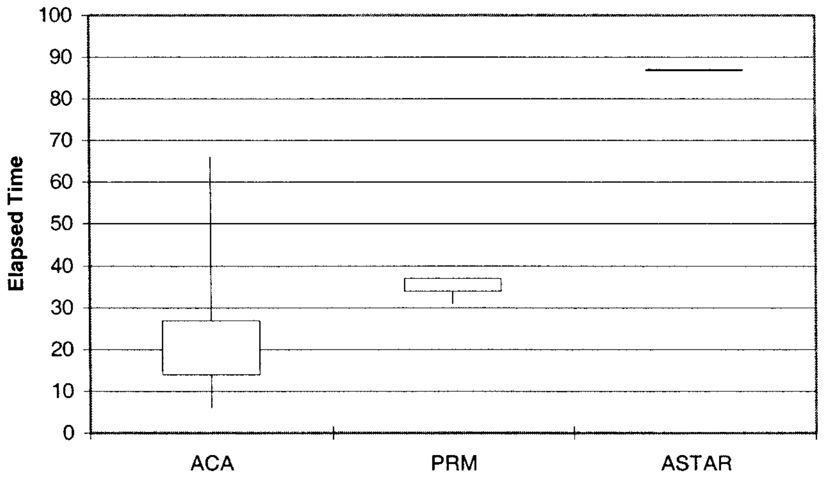

Figure 14. Elapsed times for various planners.

rithms. All the above algorithms have individual tuning parameters that were not optimized before the tests began. All times were measured on an Intel Pentium III-500 MHz processor with 128 MB of RAM.

6. CONCLUSION

The MPK system is a software toolkit to facilitate the development, testing, and comparison of robotic algorithms, particularly those with a geometric flavor such as automatic path planning, grasping, etc. It includes objects to simplify kinematic computations, geometric objects, and collision detection, which are some of the most time consuming areas of designing robotic software. It is a system designed for extensibility, i.e., incorporating algorithms designed and developed at different places in one integrated environment. The planned Web enabled front end to the system will allow researchers to experiment with different motion planning algorithms online on various different robots and environments.

In summary, the main points we would like to emphasize about the MPK are as follows:

- Ability to easily load descriptions of robots and geometries.
- Ability to choose from a variety of planning algorithms.
- Ability to choose from a variety of collision detection algorithms.
- Easy to plug in code for new algorithms—planning and collision detecting.
- Ability to add new planners and collision detectors to the Web interface as they are developed—a good way to evaluate others' work.
- Implementation of a MS Windows application that mimics the Web based application without using the Internet. This can aid researchers in their debugging effort by providing a convenient test bed in which to experiment while coding.

Several current and past students and researchers in the SFU Robotics laboratory contributed to the development of the MPK. We would like to thank Juan Manuel Ahuactzin, Francisco Javier Blanco Rodriguez, Gillian Lo, Yong Yu, Morten Petersen, and Shane Schneider for all their assistance.

REFERENCES

1. K.K. Gupta, "Overview and state of the art," Practical motion planning in robotics: current approaches and future directions, Wiley, New York, 1998, pp. 3–8.
2. J. Latombe, Robotic motion planning, Klewer Academic Publishers, Norwall, MA, 1990.
3. J.S. Mogal, IGRIP—a graphics simulation program for workcell layout and off-line programming, Robots 10 Conf Proc, 1988, pp. 65–77.
4. http://www.tecnomatix.com/Products/ROBCAD.asp.
5. http://www.silma.com.
6. E. Mazer et al., ACT: a robot programming environment, Proc IEEE Int Conf Robotics Automat, April 1991, pp. 1427–1432.
7. S. Piccinocchi, Interactive benchmark for planning algorithms on the Web, Proc 1997 IEEE Int Conf Robotics Automat, April 1997, pp. 399–405.
8. http://teamster.usc.edu/fixture/.
9. P.I. Corke, A robotics toolbox for MATLAB, IEEE Robotics Automat Mag 3 (1996), 24–32.
10. R. Gourdau, Object-oriented programming for robotic manipulator simulation, IEEE Robotics Automat Mag 4 (1997), 21–29.
11. http://www.geom.umn.edu/apps/.
12. K. Mehlhorn and S. Naher, LEDA: a platform for combinatorial and geometric computing, Commun ACM 38 (1995), 96–102.
13. http://www.solidworks.com/.
14. http://www.ktx.com/3dsmaxr2/.
15. Collision detection and geometric complexity, Part 3, Practical motion planning in robotics, A.P. Del Pobil, K. Gupta (Editors), Wiley, New York, 1998, pp. 201–274.
16. J. Cohen, M. Lin, D. Manocha, and K. Ponamgi, ICOLLIDE: an interactive and exact collision detection system for large scale environments, Interactive 3D Graphics Conf, Monterey, April 1995, pp. 189–196.
17. T.C. Hudson, M.C. Lin, J. Cohen, S. Gottschalk, and D. Manocha, VCOLLIDE: accelerated collision detection for VRML, Proc 2nd Annu Symp Virtual Reality Modeling Language, Monterey, 1997.
18. http://www.win.tue.nl/cs/tt/gino/solid/.
19. M. Greenspan and N. Burtnyk, Obstacle count independent real-time collision avoidance, ICRA96: Proc 1996 IEEE Int Conf Robotics and Automat, Minneapolis, April 1996, pp. 22–29.
20. J.M. Ahuactzin and K. Gupta, The kinematic roadmap: a motion planning based global approach for inverse kinematics of redundant robots, IEEE Trans Robotics Automat RA-15 (1999), 653–670.
21. K.K. Gupta and X. Zhu, Practical global motion planning for many degrees of freedom: a novel approach within sequential framework, J Robotic Syst 12 (1995), 105–117.
22. J. Barraquand and J.C. Lathombe, Robot motion planning: a distributed representation approach, Int J Robotics Res 10 (1991), 628–679.
23. L. Kavraki, P. Svestka, J.C. Latombe, and M. Overmas, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, IEEE Trans Robotics Automat 12 (1996), 556–580.

24. Y. Hwang, P. Xavier, P. Chen, and P. Watterberg, ''Motion planning with SANDROS and the configuration space toolkit,'' Practical motion planning in robotics: current approaches and future directions, Wiley, New York, 1998, pp. 55–77.
25. J. Ahuactzin, K. Gupta, and E. Mazer, Manipulation planning for redundant robots: a practical approach, Int J Robotics Res 17, (1998), 731–747.
26. P. Bessiere, E. Mazer, and J. Ahuactzin, The Ariadne's clew algorithm: global planning with local methods, Algorithmic Foundations of Robotics, First Workshop Algorithmic Foundations of Robotics, Wellesley, MA, 1994, pp. 39–47.
27. R. Bohlin and L.E. Kavraki, Path planning using lazy PRM, Proc Int Conf Robotics Automat, 2000, vol. 1, pp. 521–528.