Path planning by querying persistent stores of trajectory segments*

R. L. Grossman S. Mehta X. Qin

September, 1992

Laboratory for Advanced Computing Technical Report Number LAC 93-R3, University of Illinois at Chicago, September, 1992.

Introduction. In this paper, we introduce an algorithm for path planning (long duration) paths of a dynamical system, given a database, or *store*, containing suitable collections of short duration trajectory segments. We also describe experimental results from a proof-of-concept implementation of the algorithm. The basic idea is to create a persistent object store (1) consisting of short duration trajectory segments and compute the desired path by a suitable query on the store of trajectory segments. The query returns a concatenation of short duration trajectory segments which is close to the desired path. The needed short duration segments are computed by using a divide and conquer algorithm to break up the original path into shorter paths; each shorter path is then matched to a nearby trajectory segment which is part of the persistent object store by using a index function.

The more complicated the flight dynamics of an aircraft the more costly it is to accurately compute and control its flight path. On the other hand, a complex trajectory segment and a simple trajectory segment can be retrieved at equal cost given the proper design of a database, and suitable indicies for accessing the trajectory segments. After some cross over point, it becomes more efficient to retreive previously computed or simulated trajectory segments, than to compute such segments. This observation is our starting point. Often times, trajectories near by a given reference trajectory are also required. Again, given the proper database design, retrieval of near by trajectories can be achieved with little additional cost. We have implemented a proof-of-concept prototype of such a system: experimental results are summarized in Tables 2, 3 and 4.

We expect algorithms designed to take advantage of the ability of object managers to retreive previously stored or computed data will become ever more common due to the dropping cost of secondary storage and the rising cost of computing complex scientific objects on the fly.

*This research is supported in part by NASA grant NAG2-513.

Persistent object stores. At the most basic level, an object manager (1) provide facilities for storing, accessing, and querying objects on the basis of their content, not their physical location. For example, instead of returning all TrajectorySegments in File n on Disk m, a query can return all TrajectorySegments passing through a given region.

Our application requires support for complex objects and complex queries. The data is complex in the sense that objects are defined in terms of other objects. For example, TrajectorySegments are defined are to be a linked list of SpatialPoints, which themselves are defined from other more basic objects; see Figure 1. The queries are complex in the sense a query is essentially a comlicated algorithm accessing data from the store; see Figure 2 for an example. This type of query is more complex than a traditional geometric range query, such as return all TrajectorySegments with the property that their initial point is with a ball of radius one of the origin.

The data is mainly accessed, and much less frequently stored. For this reason a database transaction model (2) is not needed. With these requirements, we designed an object manager to store and access objects created using C++. Querying was done by simply accessing the objects using standard C++.

The store consists of trajectory segments from a control system (3) of the form

$$\dot{x}(t) = F(x(t); u(t)), \quad x(0) = x^0 \in \mathbf{R}^N.$$

Here $t \mapsto u(t)$ is a control, and for fixed u, $x \mapsto F(x; u)$ is a vector field in $\mathbf{R}^N$ defined in a neighborhood of x^0. We assume sufficient smoothness on the vector fields and controls so that the control trajectories are uniquely defined. In our examples we took the controls to be piecewise constant.

By a *trajectory segment* (4), we mean the structure consisting of a linked list of points (x, t), where $x \in \mathbf{R}^N$ is point in configuration space, and t is the time at which the underlying system is in position x, and a linked list of parameter values $u \in \mathbf{R}^M$. This structure is illustrated in Figure 1. By a *path*, we mean a linked list of pairs (x, t), but this time the points (x, t) need not satisfy the dynamics of the underlying system.

Path planning queries. Suppose that the persistent store contains n trajectory segments. With a naive iterative search, trajectory segments can be retreived in time $O(n)$; with B+ trees, segments can be retrieved with cost $\log n$; with hashing, trajectory segments can be retrieved in constant time (5). For the proof-of-concept prototype, we used hashing.

We used a geometric hashing function that mapped TrajectorySegments to the numbers $\{1, 2, 3, \ldots\}$ This was done by dividing the configuration space $\mathbf{R}^N$ into N-dimensional cubes, numbering the cubes $1, 2, 3, \ldots$, and assigning each trajectory segment an index given by the number of the cube containing its initial point. Several indicies were precomputed for each TrajectorySegments by dividing configuration space into a mesh of subsquentially finer N-dimensional cubes. These different indicies were used by different stages in our divide and conquer algorithms.

The path planning query is described in detail in Figure 2. Briefly, its input is a path called the QueryPath, which is broken up into a number of shorter paths. Each such path is compared to all the TrajectorySegments in the store with the same index. If an acceptably close TrajectorySegment is found, it is added to the output path called the FlightPath. Otherwise, the path is broken up into finer paths, and the algorithm is called again recursively with the new shorter path, but this time using a hashing function computed using a finer grid.

Implementation. A software tool called PTool was written to map memory resident C++ objects (transient objects) into disk resident ones (persistent objects). This was done by using the Unix operating system call mmap (6), which is part of the interface between the operating system and the underlying virtual memory system. It is then easy to define sets of transient and persistent objects.

Persistent TrajectorySegments were generated using a fourth order Runge Kutta algorithm, for a variety of initial conditions and parameter values, and stored in persistent sets using PTool. For the experiments described below, the initial conditions and parameter values were varied uniformly throughout the region of interest. A query proceeds by opening the database, loading a persistent set, and then accessing the persistent TrajectorySegments as usual using C++.

Experimental results. Five different persistent stores were created and queried for this study. The differential equations describing the dynamics are given in Table 1. The dimensions of the system range three to six, the size of the stores from 14 MBs to 186 MBs, and the number of TrajectorySegments from 121,000 to 1,000,000. See Table 2 for details. For each system, the algorithm described in Figure 2 was applied to three different QueryPaths. The time required to complete the three queries for System 1 is given in Table 3. The average times to complete the three queries is described in Table 4 for each of the systems. In all cases, the average accuracy was simply a function of the denseness in which the TrajectoryStore was populated: the denser the store was populated, the more accurate was each query.

Conclusion. The path planning algorithm described produces a concatenation of TrajectorySegments which approximate an arbitrary QueryPath to an accuracy determined by the density in which the TrajectoryStore is populated. The store was populated with TrajectorySegments obtained by sampling the initial conditions and parameter values uniformly. Decreasing the step size of the sampling, increased the over all accuracy of the algorithm, as expected.

The cost to populate a store is constant for each TrajectorySegment; the higher the number of TrajectorySegments, the higher the cost of population. The cost of query consists of two parts: a preselection in which all TrajectorySegments with the proper index are retrieved; and a selection in which of the TrajectorySegments retrieved, the one with closest to the QueryPath is selected. Since hashing is used, the cost of preselection doesn't vary that much between queries or between systems. The cost of the selection does vary a bit, depending upon the dimension and the geometry. Over all though, the cost of the

query is largely independent of the complexity of the QueryPath or of the system, unlike other types of path planning algorithms. For example, whether the QueryPath is chosen to avoid obstacles does not effect the cost of the query. Of course, for the query to succeed the population must of provided the necessary TrajectorySegments to the store.

The accuracy of the algorithm is a function of the denseness in which the store is populated, as mentioned. As the dimension of the system increases, the cost of populating the store increasings dramatically. This is because the cost of population is a function of the volume of phase space, which is an exponential function of dimension of phase space. On the other hand, it is straightforward to parallelize the population of the store, since each TrajectorySegment, and more generally, each region of phase space may be populated in parallel. Similarly, it is straightforward to parallelize the query, since here again, each segment in the QueryPath may be queried in parallel. In other words, for both the population and the query, we may use data centered parallelism to speed up the computation.

References.

1. See, for example, A. Dearle, G. M. Shaw, and S. B. Zdonik, *Implementing Persistent Object Bases: Principles and Practice,* Morgan Kaufmann, San Mateo, California, 1991.

2. H. Korth and A. Silberschatz, *Database System Concepts,* McGraw-Hill, New York, 1986 is a general reference on databases, Chapter 11.

3. A. Isidori, *Nonlinear Control Systems: An Introduction,* Springer-Verlag, Berlin, 1985.

4. J. Guckenheimer and P. Holmes, *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields,* Springer-Verlag, New York, 1986.

5. For general indexing and hashing methods, see Korth and Silberschatz, op. cit, Chapter 8.

6. Memory mapped input/output is described in W. Richard Stevens, *Advanced Programming in the UNIX Environment,* Addison-Wesly Publishing Company, Reading, Massachusetts, 1992, pp. 407–413.

7. System 1 is taken from J. Hauser, S. Sastry, and G. Meyer, "Nonlinear controller design for flight control systems," Electronics Research Laboratory, University of California at Berkeley, No. UCB/ERL M88/76, 1988; the other systems are taken from R. M. Murray and S. S. Sastry, "Nonholonomic motion planning: steering using sinusoids," Electronics Research Laboratory, University of California at Berkeley, No. UCB/ERL M91/45, 1991.

```
class SpatialPoint {
public:
    float  time;
    float *tuple;
    SpatialPt *next;
};

class ParameterNode {
public:
    float  param;
    ParamNode *next;
};

class TrajectorySegment {
public:
    int  dim;
    ParameterNode *firstP;
    SpatialPoint *firstS;
```

Figure 1: Defining the TrajectorySegment class.

The input to the algorithm is the desired flight path QP. The output is a linked list FP of TrajectorySegments (the flight path) which is close to QP.

P. Populate. In a precomputatoin, short trajectory segments for a variety of control values are numerically computed and then used to populate the database. This is done using PTool, which maps memory resident structures into persistent disk resident structures.

I. Assign indicies. For each TrajectorySegment, indicies for a sequence of finer and finer mesh sizes are computed. This indicies are part of the TrajectorySegment object. This step is also a precomputation.

P. Get the query path. Let QP denote the query path. Let $Q = QP$.

B. Break up query segment Q. Break up the query segment Q into p query segments Q_1 ..., Q_p and place them on a stack. Compute an index for each segment Q_i.

M. Match query segments. If the stack is empty, go to Step R; otherwise, remove a query segment Q from the stack and retreive all TrajectorySegments from the persistent store with the same index as Q. If a TrajectorySegment T is found which is close enough to the query segment Q, go to step A; otherwise, go to Step B.

A. Add TrajectorySegment T. Add TrajectorySegment T to the flight path FP. Goto Step M.

R. Return flight path. Return the FlightPath FP.

Figure 2: The query algorithm to compute a flight path consisting of TrajectorySegments, given a desired flight path.

System 1	$\dot{x}_1$	$= x_2$
	$\dot{x}_2$	$= v_1$
	$\dot{y}_1$	$= y_2$
	$\dot{y}_2$	$= v_2$
	$\dot{\theta}_1$	$= \theta_2$
	$\dot{\theta}_2$	$= \frac{1}{\varepsilon}(\sin\theta_1 + v_1\cos\theta_1 + v_2\sin\theta_1)$
System 2	$\dot{x}$	$= \cos\theta u_1$
	$\dot{y}$	$= \sin\theta u_1$
	$\dot{\varphi}$	$= u_2$
	$\dot{\theta}$	$= \frac{1}{l}\tan\varphi u_1$
System 3	$\dot{x}_1$	$= x_2$
	$\dot{x}_2$	$= x_3$
	$\dot{x}_3$	$= x_4$
	$\dot{x}_4$	$= x_5$
	$\dot{x}_5$	$= u_0 + u_1 t + u_2 t^2 + u_3 t^3 + u_4 t^4$
System 4	$\dot{x}_1$	$= x_2$
	$\dot{x}_2$	$= x_3$
	$\dot{x}_3$	$= u_0 + u_1 t + u_2 t^2$
	$\dot{y}_1$	$= y_2$
	$\dot{y}_2$	$= y_3$
	$\dot{y}_3$	$= v_0 + v_1 t + v_2 t^2$
System 5	$\dot{\varphi}$	$= u_1$
	$\dot{l}$	$= u_2$
	$\dot{\theta}$	$= \frac{-m_1(l+1)^2}{1+m_1(l+1)^2}$

Table 1: These equations represent a variety of nonlinearities which arise in modeling physcial systems (7). System 1 models an aircraft with a non-minimum phase nonlinearity. System 2 models a kinematic car. Systems 3 and 4 are chains of integrators. System 5 models a hopping robot.

System	Dim.	Store size	TrajectorySegments	Length
1	6	186 MBs	1,058,400	5 points
2	4	107 MBs	792,000	5 points
3	5	89 MBs	576,000	5 points
4	6	170 MBs	960,400	5 points
5	3	14 MBs	121,500	5 points

Table 2: Five different persistent stores were studied. As the dimension of the system increases, more TrajectorySegments are required in order to return flight paths with roughly the same accuracy. To obtain an average error between the desired QueryPath and the returned TrajectorySegment of 0.1, $120,000$ TrajectorySegments are needed. for a system of dimension 3. For a system of dimension 6, $1,058,000$ TrajectorySegments are needed.

Query	Length	Preselection	Selection	Accuracy
query 1	90	21/43	127/1	0.192441
query 2	90	20/40	301/3	0.191127
query 3	90	22/45	117/1	0.186922

Table 3: The time required to complete a query for System 1 and the accuracy obtained are given for three different query paths. The preselection time measures the amount of time required to fetch all trajectories with the required index; the selection time measures the amount of time to select from among these trajectories the one with the best fit. Times are in seconds in the form user time/system time. The length of the QueryPaths is measured by the number of points in the path. The QueryPaths and the TrajectorySegments were all contained in the rectangular domain $[1.0, 2.4] \times [1.0, 2.0] \times [1.0, 2.4] \times [1.0, 2.0] \times [1.0, 2.8] \times [1.0, 2.5]$.

System	Preselection	Selection	Accuracy
1	21/43	260/2	0.1875
2	21/35	2/0.8875	0.1548
3	15/30	111/2	0.45
4	24/52	219/4	0.3784
5	2.68/4	50/0.875	0.1132

Table 4: The average time, average accuracy and average relative accuracy for the indicated systems. The average is over three different query paths.