RSD-TR-14-85

AUTOMATIC GENERATION OF TRAJECTORY PLANNERS FOR INDUSTRIAL ROBOTS[1]

Kang G. Shin

Neil D. McKay

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109

October 1985

CENTER FOR RESEARCH ON INTEGRATED MANUFACTURING

Robot Systems Division

COLLEGE OF ENGINEERING

THE UNIVERSITY OF MICHIGAN

ANN ARBOR, MICHIGAN 48109-1109

---

[1]This work was supported in part by the NSF Grant No. ECS-8409938 and the US Airforce Contract No. F49620-82-C-0089. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the funding agency.

TABLE OF CONTENTS

1. INTRODUCTION 2

2. TRAJECTORY PLANNING PROBLEM AND ITS SOLUTIONS 4

3. TRAJECTORY PLANNING SYSTEM STRUCTURE 13

3.1. Data Structures for Representing Geometric Paths 13

3.2. Selection of a Trajectory Planning Method 17

3.3. Generation of Dynamic Equations 18

3.4. Generating the Constraint Function 21

3.5. Ancillary Software 22

4. CONCLUDING REMARKS 23

ABSTRACT

The control of industrial robots is usually divided into several sequential stages. Trajectory planning is an important off-line stage which is concerned with the generation of a time history of a robot's joint position, velocity, acceleration, and input torques.

A number of trajectory planning methods have been developed [1], [4]-[6], [10], [11], [13], which usually entail complex computations and algebraic manipulations. Programming this sort of trajectory planners is very complex and error-prone, thereby limiting their applicability. To remove this limitation, we have begun the development of software for automating the trajectory planning, called the *Automatic Trajectory Planner Generator* (ATPG). This paper describes important components of the ATPG: three trajectory planners, data structures for describing geometric paths, generation of the robot's dynamic equations and constraint functions, and ancillary software. A large portion of the ATPG has been completed, and the remaining portion is currently under development.

1. INTRODUCTION

Due to the complexity and nonlinearity of the robot's dynamics, the robot control problem is usually divided into several subproblems, which are first solved individually and then combined. This division can best be explained by Figure 1. From a *task planner* we obtain an ordered sequence of points in Cartesian space which represent a collision-free path if we connect them properly (e.g., by spline functions or straight line segments). The *geometric path generator* (i) transforms this Cartesian points to the corresponding points in joint space, and (ii) using the transformed points in joint space, generates a *geometric path* which is a parameterized curve in joint space (to be discussed in Section 2). The trajectory planner receives these geometric paths as input and determines a time history of position, velocity, acceleration, and input torques which are fed to a *trajectory tracker.* The trajectory tracker drives the robot to follow the desired trajectory specified by the trajectory planner with feedback information on a subset of position, velocity, acceleration, and input torques.

As can be seen from the results presented in our previous papers[4], [10], [11], [13] and others [1], [3], [6], writing trajectory planning programs can be a laborious task. Before actually writing the program, kinematic and dynamic equations must be derived, and actuator constraints must be described. In addition, other mundane details, such as what data structures to use, must be considered. The aim of this paper is to describe the process of trajectory planner

generation in detail, and present sufficient guidelines so that the entire process can be automated.

There are two major reasons for automating the trajectory planner generation process. First, generating a trajectory planner without computer assistance consumes much expensive human labor, even if the task is performed only occasionally. This is due in large part to the perils of hand calculation; deriving dynamic equations by hand is slow and error-prone. Second, writing a trajectory planner is time-consuming. To understand why this second factor is important, consider the use of trajectory planners as robot design aids. If trajectory planners can be generated quickly, then a hypothetical robot design can be tested easily and accurately; the robot can be pushed to its working limits, and the designer can then check to see, for example, if any joints are under-powered. It also makes the effects of design changes easy to evaluate, since a new trajectory planner can be generated quickly, and a new set of tests can be run.

The paper is organized as follows. For completeness we begin in the next section with a brief description of the trajectory planning problem and its solutions. In Section 3, the major components of a trajectory planning system are described. This description includes both functional descriptions of the components and suggestions regarding implementation details such as the choice of data structures. The generation of these components can then be reduced to algorithmic form suitable for computer implementation. The paper concludes with Section 4, where the status of our automatic trajectory planner is given.

2. TRAJECTORY PLANNING PROBLEM AND ITS SOLUTIONS

Before discussing the problem of generating an automatic trajectory planner, we will first describe briefly the trajectory planning problem and its solutions. (See [10], [11], [13] for detailed descriptions.)

For the trajectory planning problem, we have to consider the effects of restricting the manipulator's motion to a fixed collision-free path, which is specified by the geometric path generator in Figure 1. In what follows, the manipulator will be restricted to some geometric path

$$\mathbf{q}^i = f^i(\lambda), \ 0 \leq \lambda \leq \lambda_{\max} \tag{1}$$

where $\mathbf{q}^i$ is the position of the *i-th* joint and λ is a scalar parameter. Since the parameter λ along with the functions f^i completely describes joint positions, it will be referred to as the "position" variable. The *i-th* joint velocity then becomes

$$\mathbf{v}^i = \dot{\mathbf{q}}^i = \frac{df^i}{d\lambda}\frac{d\lambda}{dt} = \frac{df^i}{d\lambda}\dot{\lambda} = \frac{df^i}{d\lambda}\mu \tag{2}$$

where $\mu \equiv \dot{\lambda}$ is the *pseudo-velocity* of the manipulator. Plugging this into the usual dynamic equations

$$\mathbf{u}_i = \mathbf{J}_{ij}(\mathbf{q})\ddot{\mathbf{q}}^j + \mathbf{C}_{ijk}\dot{\mathbf{q}}^j\dot{\mathbf{q}}^k + \mathbf{R}_{ij}\dot{\mathbf{q}}_j + \mathbf{g}_i(\mathbf{q}) \tag{3}$$

gives the following equations of motion along the the geometric path

$$\dot{\lambda} = \mu \tag{4a}$$

$$\mathbf{u}_i = \mathbf{J}_{ij}(\lambda)\frac{df^j}{d\lambda}\dot{\mu} + \mathbf{J}_{ij}(\lambda)\frac{d^2f^j}{d\lambda^2}\mu^2$$

$$+ \mathbf{C}_{ijk}(\lambda)\frac{df^j}{d\lambda}\frac{df^k}{d\lambda}\mu^2 + \mathbf{R}_{ij}\frac{df^j}{d\lambda}\mu + \mathbf{g}_i(\lambda) \tag{4b}$$

where $\mathbf{u}_i$ is the torque/force applied at the i-th joint, $\mathbf{J}_{ij}$ the $N \times N$ inertia matrix, $\mathbf{C}_{ijk}$ the Coriolis force array, $\mathbf{R}_{ij}$ the matrix representing viscous friction, $\mathbf{g}_i$ the gravitational force, and N the number of the robot's joints. The Einstein summation convention is used here, and all indices run from 1 to N..

It is of course assumed that the coordinates $\mathbf{q}^i$ vary continuously with λ. It is also assumed that the derivatives $\frac{df^i}{d\lambda}$ and $\frac{d^2f^i}{d\lambda^2}$ exist, and that the derivatives $\frac{df^i}{d\lambda}$ are never all zero simultaneously. This ensures that the path never retraces itself as λ goes from 0 to λ_{max}. Such a retrace would force the parameter λ to take a discontinuous jump in order for the point $\mathbf{q}^i$ to move forward continuously.

It should be noted that in practice the geometric paths are given in Cartesian coordinates. While it is in general difficult to convert a curve in Cartesian coordinates to that in joint coordinates, it is relatively easy to perform the conversion for individual points. One can then pick a sufficiently large number of points on the Cartesian path, convert to joint coordinates, and use some sort

of interpolation technique (e.g. cubic splines) to obtain a similar path in joint space (see [6] for an example). Introducing some shorthand notation, let

$$M_i \equiv \mathbf{J}_{ij} \frac{df^j}{d\lambda} ,$$

$$Q_i \equiv \mathbf{J}_{ij} \frac{d^2 f^j}{d\lambda^2} + \mathbf{C}_{ijk} \frac{df^j}{d\lambda} \frac{df^k}{d\lambda} ,$$

$$R_i \equiv \mathbf{R}_{ij} \frac{df^j}{d\lambda} ,$$

$$S_i \equiv \mathbf{g}_i .$$

We can then express Eq. (4b) by:

$$\mathbf{u}_i = M_i \dot{\mu} + Q_i \mu^2 + R_i \mu + S_i . \tag{5}$$

Note that the quantities listed above are functions of λ. For the sake of brevity, the functional dependence is not indicated in what follows.

The goal of automation is to produce goods at as low a cost as possible. In practice, costs may be divided into two groups: fixed and variable. Variable costs depend upon details of the manufacturing process, and include, in the cases where robots are used, that part of the cost of driving a robot which varies with robot motion, and some maintenance costs. Fixed costs are those which remain constant on a per-unit-time basis. Fixed costs include taxes, heating costs, building maintenance, and, in the case of a robot, the portion of the electric power which the robot uses to run its computer controller and other

peripheral devices. If one assumes that the fixed costs dominate, then cost per item produced will be proportional to the time taken to produce the item. In other words, minimum cost is equivalent to minimum production time. A loose statement of the minimum-cost trajectory planning problem is as follows:

> What controls will drive a given robot along a specified curve in joint space with minimum cost, given constraints on initial and final velocities and on control signal magnitudes?

As was seen in Eqs. (1), (2), (4a) and (4b), this form of the problem reduces the complexity of the control problem by introducing a single parameter λ which describes the robot's position. The time derivative of this parameter and the parameter itself completely describe the current state (joint positions and velocities) of the robot. The control problem then becomes essentially a two dimensional minimum-cost control problem with some state and input constraints.

The minimum-cost control problem can be stated as follows.

> Given a curve in the robot's joint space (or some equivalent coordinate system), the robot's dynamic properties, and the robot's actuator characteristics, what set of signals to the actuators will drive the robot from its current state to a desired final state with minimum cost?

To state the above problem more formally, assume that the geometric path is given in the form of a parameterized curve as in Eq. (1). Also assume that the constraints on the actuator torques can be expressed in terms of the state of the system, i.e., in terms of the robot's speed and position, so that

$$\mathbf{u} \in \mathbf{E}(\mathbf{q},\dot{\mathbf{q}}) = \mathbf{E}_1(\lambda,\mu) \tag{6}$$

where $\mathbf{u}=(u_1,u_2,\ldots,u_N)^T$ is a vector of actuator torques/forces, and $\mathbf{E}:\mathbf{R}^N\times\mathbf{R}^N\rightarrow\mathbf{R}^N$ and $\mathbf{E}_1:\mathbf{R}^2\rightarrow\mathbf{R}^N$ are set functions. N is the number of joints the robot has. Given the functions f^i, the set functions $\mathbf{E}$ and $\mathbf{E}_1$, the desired initial and final velocities, and the manipulator dynamic equations (4a) and (4b), the trajectory planning problem is to find the controls $\mathbf{u}(\lambda)$ which minimize the cost functional C given by

$$C=\int_0^{\lambda_{max}}\phi(\mathbf{u}(\lambda),\mathbf{q}(\lambda),\dot{\mathbf{q}}(\lambda))d\lambda. \tag{7}$$

A practical aspect of the trajectory planning problem is that of the description of curves and the actual calculation of actuator torques. Some suitable method of representing curves is required, and all computations involving those curves should be done automatically. In particular, it should be possible, given a robot's dynamic equations, to generate a trajectory planner for that robot. This is especially desirable in view of the fact that the dynamic equations of all but the simplest robots are very complicated, and any manipulation of such equations will be prone to human error.

One possible approach to the solution of the above trajectory planning problem is to apply one of the standard tools of optimal control theory, Pontryagin's maximum principle. However, this approach requires solving a two-point boundary value problem for a non-linear system of differential equations with non-linear constraints; clearly, this does not lead to a tractable solution. The maximum principle also sheds little or no light on the other auxiliary

problems, such as sensitivity to parameter variations. Moreover, since the trajectory planning problem frequently requires state or mixed state-control constraints, the maximum principle is not usually applicable. Therefore, we have taken a more intuitive but systematic approach to develop the three trajectory planners in [10], [11], [13], whose brief descriptions are given below for completeness. (A similar approach was also proposed independently of ours [1].)

The first method is referred to as the *phase plane method.* It is so called because it makes use of plots of the "pseudo-velocity" $\mu \equiv \lambda$ vs. the position parameter λ. Such a plot, in which a velocity is plotted as a function of position, is generally referred to as a "phase plane plot", hence the name. Actually all three trajectory planners make use of this idea in one way or another, and from here on, the term "trajectory" will be taken to mean "phase trajectory", or λ-μ plot.

The phase plane method is in general applicable only to minimum time problems, i.e.,

$$C = \int_0^{\lambda_{max}} \frac{d\lambda}{\mu} . \tag{7'}$$

Since only minimum-time solutions are to be considered, it is useful to consider how this restriction on the objective function can be used. Obviously, minimizing traversal times is equivalent to maximizing traversal speed. Given this fact, it is easy to see that, at least in the simplest case, the minimum time solution consists of an accelerating and a decelerating part; the robot should accelerate

at its maximum rate, then "put on the brakes" at precisely that time which will bring it to a stop at the destination point. Of course, there will in general be some velocity limits as well as acceleration limits. The velocity limits are imposed by the interaction of velocity-dependent force terms in the dynamic equations and the actuator torque limits; the actuators must generate enough torque to overcome these forces and keep the manipulator on the desired path. If the robot is to avoid these velocity limits, then the trajectory must alternately accelerate and decelerate, and the switching points should be timed so that the trajectory just barely misses exceeding the velocity limits. A more precise description of this method can be found in [10], including a derivation of the velocity limits and an algorithm for generating the optimal trajectories. Other complications are also discussed there.

As an alternative, we have used dynamic programming to solve the trajectory planning problem [11]. Dynamic programming is an impractical method for solving the general path planning problem for an arm with a large number of joints, since there are two state variables per joint, thus requiring a $2N$ dimensional grid. (This is a classic example of the "curse of dimensionality".) However, when the path is given, there are only two state variables (i.e., λ and μ); thus only a 2-dimensional grid is required.

To use dynamic programming, the grid is set up so that the position parameter λ is used as the stage variable. Thus a "column" of the grid corresponds to a fixed value of λ, while a "row" corresponds to a fixed μ value. One starts at the desired final state (the last column of the grid, with the row

corresponding to the desired final μ value) and assigns that state zero cost. All other states with position λ_{max} are given a cost of infinity. Then the usual dynamic programming algorithm can be applied. The algorithm starts at the last column. For each point in the previous column one finds all the accessible points in the current column, determines the minimum cost to go from the previous to the current column, and increments costs accordingly. For each of the previous grid points, the optimal choice of the next grid point is recorded. When the initial state is reached, the optimal trajectory is found by following the pointer chain which starts at the given initial state. In the case at hand, determining which points are accessible from one column to the next is simply a matter of checking to see if the slope of the curve connecting the two points gives a permissible value. (The slope limits can be found from the constraints on the actuator torques (6).) The incremental cost is computed for minimum cost problems, so a running sum can be kept for the total cost.

Dynamic programming has the advantage that it is a well-established and well-understood optimization method. It also gives the control law for *any* point on the curve, and so makes provision for the robot to vary its speed if necessary. (This of course assumes that the robot stays on the desired path.) On the other hand, if it is implemented in the most obvious and straightforward manner, it requires a large array for computations, and if the array size is to be known in advance then an upper bound on the velocity is needed. In practice, there may be artificially imposed velocity bounds, but in general it would be necessary to either calculate velocity bounds in advance or create a new (larger) grid and

start all over if the trajectory left the grid. The computation times also increase rather quickly as the density of the grid, and hence the accuracy of the solution, increases. Some modifications to the algorithm are suggested in [7] which should considerably increase its speed.

It should also be noted that dynamic programming may still be used even if the robot's actuator torque constraints are not independent of one another; this is not the case with the phase plane method. Making the phase plane algorithm work for non-independent actuators would require that the space of actuator torques be searched for an acceleration bound. Dynamic programming only requires that a function be available which returns a yes-or-no answer to the question "if this acceleration is desired, will the required torques be realizable?". The dynamic programming algorithm itself performs the search of the actuator torque space.

A third algorithm, called the *perturbation trajectory improvement algorithm* (PTIA), is developed as a simple but powerful method [13]. This algorithm is in some respects similar to the dynamic programming algorithm, though like the phase plane method it is only applicable to minimum time problems. This algorithm starts with an initial feasible trajectory, and perturbs the trajectory in such a way that the traversal time for the trajectory decreases, while the trajectory remains feasible. This method has most of the advantages of the dynamic programming method, and can be modified to generate minimum-time trajectories when there are limits on the jerk, or the derivative of the acceleration, as well as limits on joint torques.

3. TRAJECTORY PLANNING SYSTEM STRUCTURE

A trajectory planner is of no use in isolation; it must be part of a larger system. In a practical robot control system, the trajectory planner receives a path description from a geometric path planner, which generates the geometric path description from task descriptions. After the trajectory planner has assigned timing information to this path, it is passed on to a tracker which drives the robot in real time. However, if the trajectory planner is being used as a design aid, as suggested above, the geometric path planner, the path tracker, and the robot will not actually be present. In this case, a driver for generating geometric paths and a stub for analyzing the trajectory planner output are required. Such a system is shown schematically in Figure 2.

The precise character of the trajectory planner, the driver routine, and the stub are determined largely by the choice of representations for the input and output data and by the particular type of trajectory planner chosen. These subjects are discussed in the following subsections.

3.1. Data Structures for Representing Geometric Paths

Choosing the type of data structure used to represent geometric paths is probably the single most important decision to be made in designing a path planning system. It influences not just the design of the trajectory planner, but the design of the geometric path planner and the tracking system as well. This being the case, the geometric path representation must be considered very carefully. Several options are considered here, and their implications for the trajec-

tory planning system design are investigated.

Three geometric path descriptions are discussed here: expression trees, splines, and arrays of points. An expression tree is a linked structure in which the internal nodes of the tree represent operators, the subtrees lying below a given node represent the operands for that node, and the leaves are irreducible expressions such as constants or variables. Such trees are equivalent to algebraic expressions. Splines are sequences of curves which are connected end-to-end. These curves usually are chosen so that they have some particular number of continuous derivatives, even at the points where the curve segments meet. For example, if cubic polynomials are used to connect the intermediate points, then it is possible to have continuous first derivatives. The second derivatives exist, but in general are discontinuous. Finally, a curve can be represented as a sequence of points. The points must be chosen so that any reasonably smooth curve which connects all the points will not deviate too much from the actual desired path.

Expression trees have the advantage that their manipulation as algebraic expressions is easy. Such manipulations mimic the processes carried out by humans, and the results are complete algebraic expressions. Such manipulations are required for deriving the robot's dynamic equations, so a formula manipulation system of some sort will be required anyway. However, expression trees have the disadvantage that introducing new types of curves for the robot to traverse may require that new types of operators be created. Also, the paths which robots are expected to traverse often are composed of connected pieces

rather than single analytic curves. Some sort of conditional operator, such as a Heaviside step, is then required, and this introduces problems when the curve is differentiated; the resulting analytic expression may contain delta functions.

Cubic (or other) splines can be thought of as a restricted case of an expression tree. The expression tree will consist of the sum of many conditional expressions, where each conditional expression consists of two step functions multiplied by a cubic polynomial. For instance, the segment of a spline which passes from point 5 to point 6 might be represented as

$$s(\lambda-\lambda_5)s(\lambda_6-\lambda)p_5(\lambda), \tag{8}$$

where

$$s(\phi)=\begin{cases}0 & \text{if } \phi<0\\ 1 & \text{if } \phi\geq 0\end{cases} \tag{9}$$

and $p_5(\lambda)$ is a cubic polynomial in λ. The problems discussed above for expression trees also apply to cubic splines.

A simple array of points does not lend itself to symbolic manipulation of any sort. However, most of the calculations which the trajectory planner must carry out can be done numerically, so this is not a big disadvantage. The major disadvantage of this method is that the path is not really completely specified; the motion between the interpolation points is undetermined. However, if the density of interpolation points is high enough, this will not matter in practice. Indeed, the perturbation and dynamic programming trajectory planners discre-

tize the trajectory planning problem anyway, and the differential equations that need to be solved when using the phase-plane method must in practice be solved using discrete, approximate methods. The representation is very simple, and for that reason was used in most of the examples presented in [10], [11], [13]. It is also a very easy structure to attach other information to; if the geometric path is represented as an array of points, and the points are represented as a structure or record, then time, torque, and velocity information can be attached to each point by simply adding new fields to the record. Attaching this information to a spline or an expression tree is much more difficult. For these reasons, an array of points appears to be a good choice of data structures.

The data structure used in the examples in this paper is shown in Figure 3a. It is a record, and it contains information which applies to the path as a whole: the maximum value of the parameter λ, the number of joints the robot has, and the number of interpolation points on the curve. In addition, this record contains pointers to several arrays, including an array of λ-values for each point on the curve. Finally, it contains a pointer to an array of "joint paths", where each joint path is an array of joint data values, one array element per point. It is in these arrays that the information pertaining to the individual joints, such as the dynamic coefficients M_i, Q_i, R_i, and S_i in Eq. (5), the parametric derivatives $\frac{d\mathbf{q}^i}{d\lambda}$ and $\frac{d^2\mathbf{q}^i}{d\lambda^2}$, and the joint torques $\mathbf{u}_i$ and motor voltages $\mathbf{V}_i$, are stored. A typical structure for these values (the $\mathbf{D}_{ij}$ in

Figure 3a) is shown in Figure 3b.

The trajectory planning process can be expressed entirely in terms of operations on this data structure. The input to the trajectory planner is a sequence of points, and the ultimate goal of the trajectory planner is to fill in the values of μ, $\dot{\mu}$, t, etc. at each point on the curve. The intermediate steps are to compute the values of the parametric derivatives $\frac{d\mathbf{q}^i}{d\lambda}$ and $\frac{d^2\mathbf{q}^i}{d\lambda^2}$, calculate the dynamic coefficients M_i, Q_i, R_i, and S_i, and finally apply a trajectory planning algorithm which generates μ, $\dot{\mu}$, t, and $\mathbf{u}_i$ for $1 \leq i \leq N$.

3.2. Selection of a Trajectory Planning Method

The choice of geometric path representations influences the structure of the entire path planning system, from the geometric path planner down through the tracking system. The choice of trajectory planning techniques has more localized effects, but the effects on the implementation of the trajectory planner, which are of primary concern here, are major. Assuming that minimum-time trajectories are desired, the perturbation method should be chosen rather than dynamic programming, since it is considerably faster and can handle more general torque constraints. The phase plane method is the fastest of the three techniques described in Section 2, but cannot be generalized to handle all the kinds of torque constraints that the perturbation method can handle. The torque constraints also must be incorporated into the phase plane algorithm in several places, making changes to the torque limits more difficult in a phase-plane trajectory planner than in a perturbation trajectory planner.

The perturbation method isolates the torque constraints into a single constraint function, thereby isolating the torque constraint checks from the rest of the trajectory planner.

Because of the ease of construction of perturbation trajectory planners, and because of the natural way that they break up into modules, they would appear to be good choices for a computer-generated trajectory planner. The major components of such a trajectory planner are shown in Figure 4. The components are a derivative generator, a dynamic coefficient generator, and the trajectory planning algorithm itself. Note that the constraint function "plugs into", or links to, the trajectory planning algorithm only, and that the trajectory planning algorithm itself is the same regardless of the torque constraints. The trajectory planning algorithm also is the same for any robot, regardless of its dynamics; everything that the trajectory planning algorithm needs to know about the robot's dynamics is distilled into the dynamic coefficients M_i, Q_i, R_i, and S_i. The derivative generator can be a simple numerical procedure and is independent of the robot characteristics also, so that only the constraint function and the dynamic coefficient generator need ever be changed.

3.3. Generation of Dynamic Equations

So far, very little has been said about the actual generation of the dynamic equations for robots. This is the most time-consuming, labor-intensive and error-prone part of the trajectory planner generation process, and

so is the most important part to automate. Computer generation of dynamic equations may be time-consuming, but it is certain to be several orders of magnitude faster than hand calculation, and certainly will be more accurate.

Though it sometimes is possible to derive dynamic equations of simple robots relatively quickly in *ad hoc* ways, such tricks are difficult to apply systematically. However, systematic methods do exist; see [9]. These systematic methods, combined with a system for doing symbolic algebra, such as MACSYMA [8] and REDUCE [3], and a symbolic differentiator make the generation of dynamic equations fairly straightforward. While the resulting equations may not be in simplest form, they will certainly be correct.

It is important to note that having the dynamic equations in simplest form may not always be necessary. In particular, this is the case if the trajectory planning system is to be used primarily as a tool for robot design. In that situation, the trajectory planner will probably be run several times and then thrown away as design changes are made. It is much more important that the trajectory planner be generated quickly than that it run quickly. (Essentially the same reasoning is used to justify the existence of slow optimizing and fast non-optimizing compilers for conventional computer languages; there usually is no sense in spending ten minutes to optimize a small program that will be run only once or twice.)

A schematic of the robot dynamics generation process is shown in Figure 5. The user first describes the robot's kinematics; this can be done by specifying the number of joints, whether the joint is revolute or prismatic, the

various joint offsets and twists, and the link lengths. The resulting transformation matrices can be computed as in [9]. Once the forward kinematics of the robot are known, the user may describe the link inertias. A simple version of a dynamic equation generator may just prompt the user for the link pseudo-inertias; a more sophisticated version would compute the inertias from CAD models of the robot links. Given the kinematics and the pseudo-inertias, the inertia matrix, the Coriolis coefficient array and the gravitational forces can be calculated using the formulas in [9]. It is for these computations that a symbolic differentiator is needed, since the kinematic transform matrices must be differentiated with respect to the joint variables.

The output of the dynamic equation generator will be arrays of expression trees representing the inertia matrix, the Coriolis coefficient array, and the gravitational force terms. These must be translated from arrays of expression trees to some form suitable for compilation by whatever high-level language compiler is chosen as an implementation language. If the target language has pointers to procedures, then an appropriate representation of, for example, the inertia matrix would be a two-dimensional array of pointers to procedures; each element of the array would be set to point to a function which would compute the appropriate inertia matrix coefficient, given the robot's current position. This requires only that the expression tree representations be translated into appropriate computer code, a very simple task.

3.4. Generating the Constraint Function

Generating the constraint function for a dynamic programming or perturbation trajectory planner poses some problems which are not amenable to general solution. Actuator torque constraints may in general be quite complicated; for example, in a cable-driven robot arm, moving one drive cable may move several joints, cables will stretch, and the cables must all be kept under positive tension. Generating constraint functions for completely arbitrary actuators is obviously very difficult, if not impossible. However, not every actuator which is theoretically possible, nor even every actuator which is practically realizable, need be considered. By restricting the trajectory planner generator to a few very common actuator types, most practical robot designs can be handled.

One type of actuator which is very common is the D.C. torque motor. The torque bounds for such a device are determined by the saturation limits of the motor itself and by the properties of the amplifier which drives the motor. In addition, there are limits on the time derivative of the torque which are imposed by the motor inductance and the drive amplifier voltage limits, but these constraints can often be ignored in practice. Since only a few parameters are needed to describe the motor torque limits, generating constraint functions for this case is a fairly simple process.

Other types of actuators, such as hydraulic servoes, may also be described in a simple manner provided that effects such as delays due to the finite speed of propagation of pressure waves are ignored. As long as these effects are

small, the resulting constraint function should produce a useful trajectory planner.

3.5. Ancillary Software

As indicated in Figure 2, the trajectory planner is not the only component of a path planning system; if the trajectory planner is to be used as a design tool, a driver for generating test data and a stub for analyzing trajectory planner output are both required.

The stub can be a very simple routine. Most of the time a graphical representation of joint speeds and forces suffices, and the generation of such output can be performed in a manner which does not depend on the kinematics or dynamics of the robot. The driver routine, on the other hand, may depend rather heavily upon the robot's kinematics. If the designer wants to evaluate the performance of the robot as it moves along a Cartesian straight line, then the inverse kinematics of the robot are required. This causes some problems, since the inverse kinematics of robot arms cannot in general be computed in closed form, and because the inverse kinematic solutions are not always unique. However, many robots fall into a relatively small set of kinematic configurations, so that the inverse kinematics can be computed for a generic member of each kinematic class. Then the designer need only plug the precise link dimensions into a standard formula. (See [2] for an example.) While this approach may not work for certain exotic robot designs, it probably will cover the vast majority of robots seen in practice.

Another desirable feature of the driver routine is the generation of geodesics in inertia space. The traversal times for these curves are near-optimal [12], and so will give the robot designer an estimate of the robot's maximum capabilities. Since the differential equations of inertial geodesics are directly obtainable from the robot's dynamic equations, much of the work of writing a geodesic generator will have been done already. The equations need only be incorporated into an appropriate differential equation solver.

The proposed trajectory planner generation system is shown schematically in Figure 6. The user provides kinematics, link inertias, and actuator characteristics, and the planner generator produces a trajectory planner, a stub, and a driver. The driver allows selection of straight line paths in joint space, Cartesian straight lines, and geodesics.

4. CONCLUDING REMARKS

Most of the work performed to date on the automatic generation of trajectory planners consists of the construction of a system for manipulation of algebraic expressions and a symbolic differentiator. These routines consist of about 1400 lines of C, with an additional 1300 lines of code for testing the differentiator. Though these utilities only constitute a portion of the trajectory planner generating system, they are very important components. The largest single component of the trajectory planner generator will most likely be the dynamic equation generator, of which the algebraic manipulation system is an integral part.

Also completed is the stub for generating trajectory planner output, which consists of about 250 lines of code for the main routine and about 2700 lines for various graphical plots.

Some of the rest of the code for the trajectory planner consists of fixed modules; these portions can be taken from the code already written for the numerical examples in [10], [11], [13], and used either directly or with minor modifications. The example for the phase plane plot method in [10] has been coded with about 350 lines of C, that for the dynamic programming method in [11] with about 400 lines, and that for the perturbed trajectory improvement algorithm in [13] with only 50 lines for the main routine and about 150 lines for generating the constraint function.

Though much work remains to be done,[1] it is clear from the discussion presented here that the major problems involved in producing an automatic trajectory planner generator involve writing some large but well-defined pieces of code; there are no major conceptual or theoretical problems to be solved.

REFERENCES

[1] J. E. Bobrow, S. Dubowsky, and J. S. Gibson, "On the Optimal Control of Robotic Manipulators with Actuator Constraints," *Proc. of American Control Conference,* pp. 782-787, June 1983.

[1] such as completion of the three trajectory planners, inclusion of the other trajectory planners, CAD modeling of the robot links, etc.

[2] S. J. Derby, "Kinematic Elasto-Dynamic Analysis and Computer Graphic Simulation of General Purpose Robot Manipulators," Ph.D. thesis, Rensselaer Polytechnic Institute, Troy, New York, August 1981.

[3] A. C. Hearn, "REDUCE User's Manual," *Report,* UCP-19, University of Utah, 1973.

[4] B. K. Kim and K. G. Shin, "Minimum-Time Path Planning for Robot Arms with Their Dynamics Included," *IEEE Trans. on System, Man, and Cybernetics,* vol. SMC-15, no. 2, pp. 213-223, March/April 1985.

[5] J. Y. S. Luh and C. S. Lin, "Optimal Path Planning for Mechanical Manipulators," *ASME Journal of Dynamic Systems, Measurements, and Control,* vol. 102, pp. 142-151, June 1981.

[6] C. S. Lin, P. R. Chang, and J. Y. S. Luh, "Formulation and Optimization of Cubic Polynomial Joint Trajectories for Mechanical Manipulators," *IEEE Trans. on Automatic Control,* vol. AC-28, no. 12, pp. 1066-1074, December 1983.

[7] N. D. McKay, "Minimum-Cost Control of Robotic Manipulators with Geometric Path Constraints," Ph.D. thesis, The University of Michigan, Ann Arbor, MI, September 1985.

[8] J. Moses, "Algebraic Simplification: A Guide for the Perplexed," *Proceedings of the second ACM Symposium on Symbolic and Algebraic Manipulation,* New York, pp. 282-304, 1971.

[9] R. P. C. Paul, *Robot manipulators: Mathematics, programming, and control,* MIT Press, Cambridge, Mass., 1981.

[10] K. G. Shin and N. D. McKay, "Robot Path Planning Using Dynamic Programming," *Proc. of 23rd CDC,* December 1984. (Also to appear in *IEEE Trans. on Automatic Control.*)

[11] K. G. Shin and N. D. McKay, "Minimum-Time Control of Robotic Manipulators with Geometric Path Constraints," *IEEE Trans. on Automatic Control,* vol. AC-30, no. 6, pp. 531-541, June 1985.

[12] K. G. Shin and N. D. McKay "Selection of Near-Minimum Time Geometric Paths for Robotic Manipulators,"

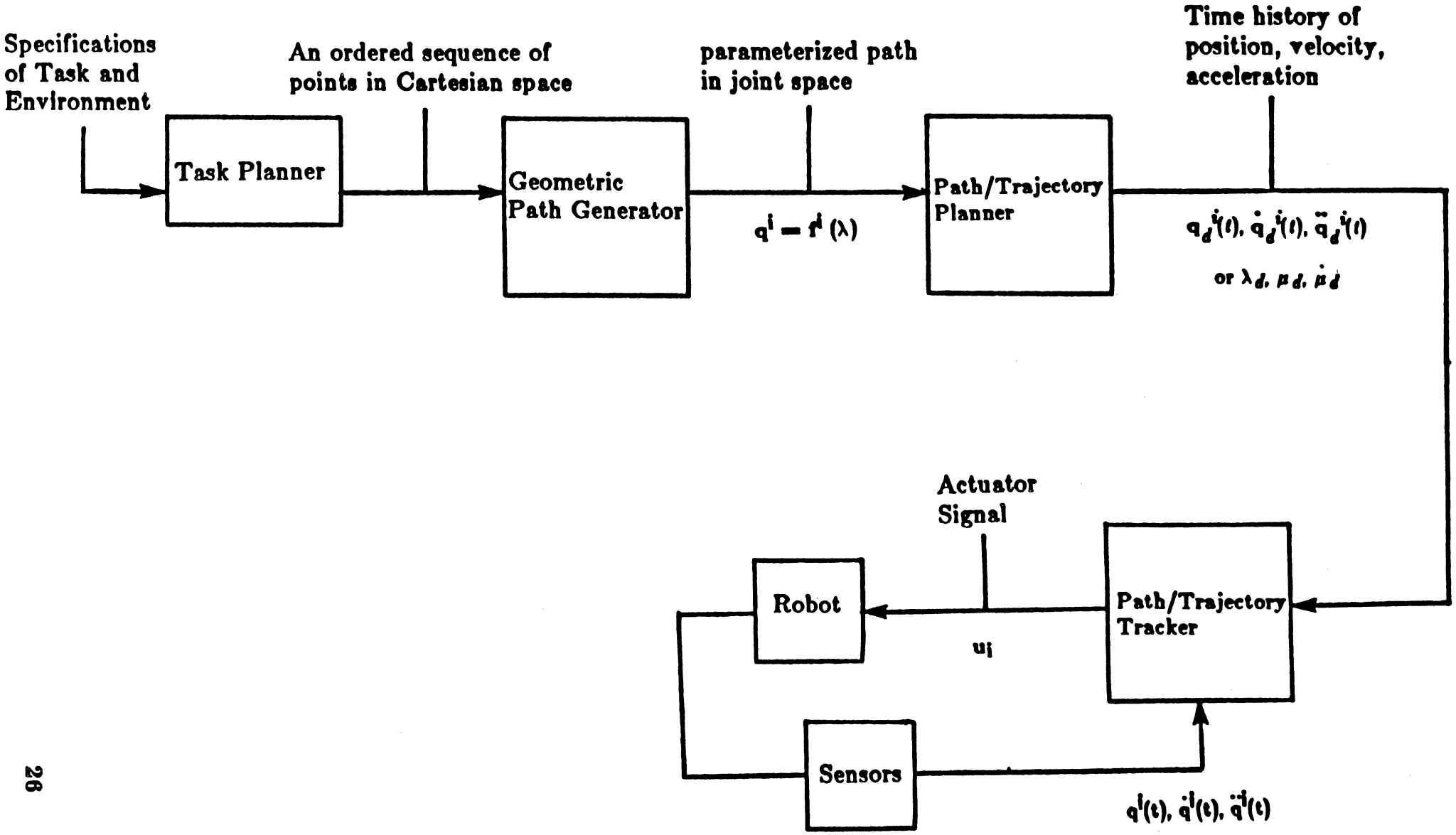


Figure 1. Functional Block Diagram of Manipulator Position Control.

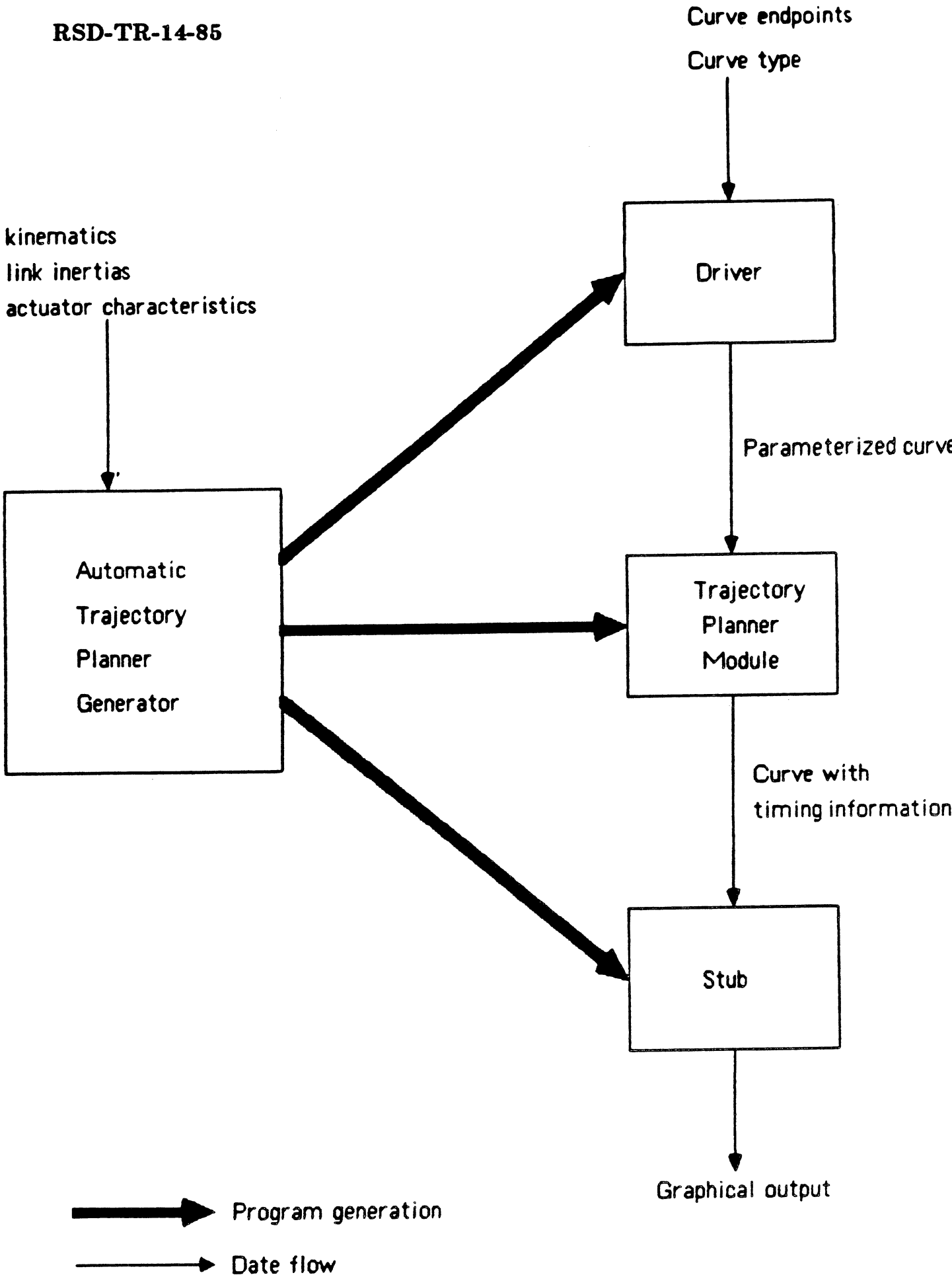
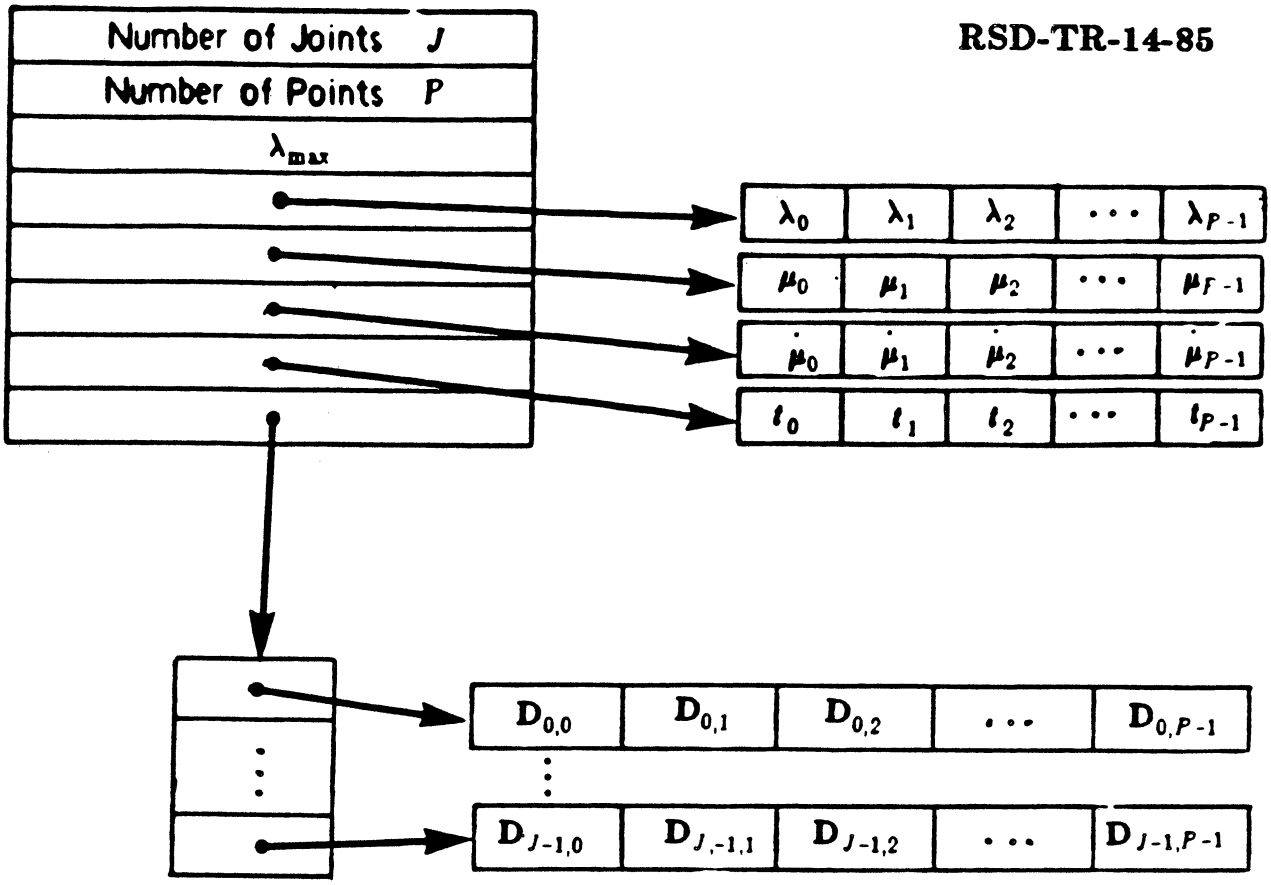


Figure 2. Overview of ATPG system



a) Path data structure

q_m^i	$\frac{dq_m^i}{d\lambda}$	$\frac{d^2q_m^i}{d\lambda}$	
$M_i(\lambda_m)$	$Q_i(\lambda_m)$	$R_i(\lambda_m)$	$S_i(\lambda_m)$
u_m^i	V_m^i		

b) The structure $D_{i,m}$

Figure 3. Robot path data structures

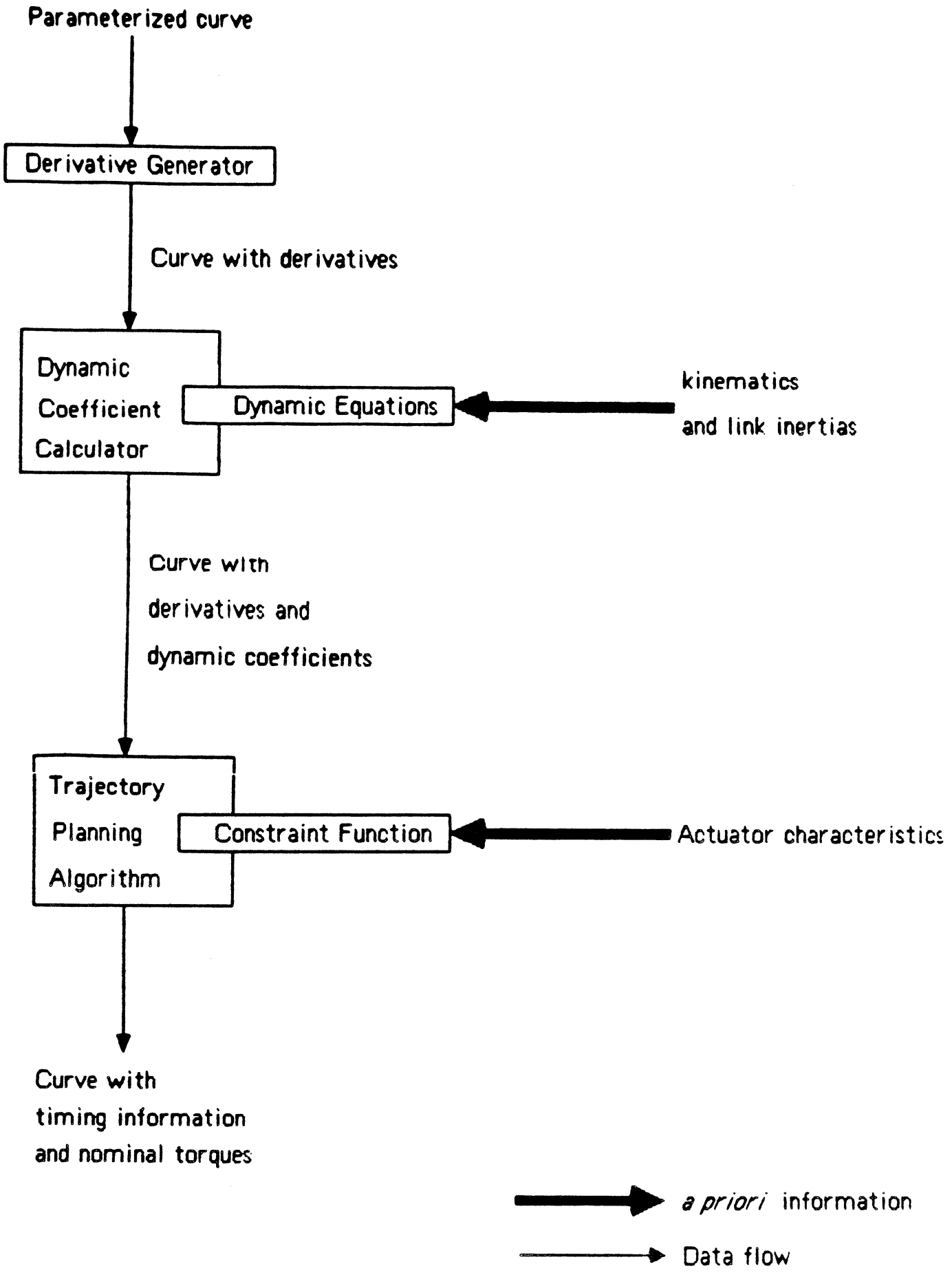


Figure 4. Trajectory planner structure

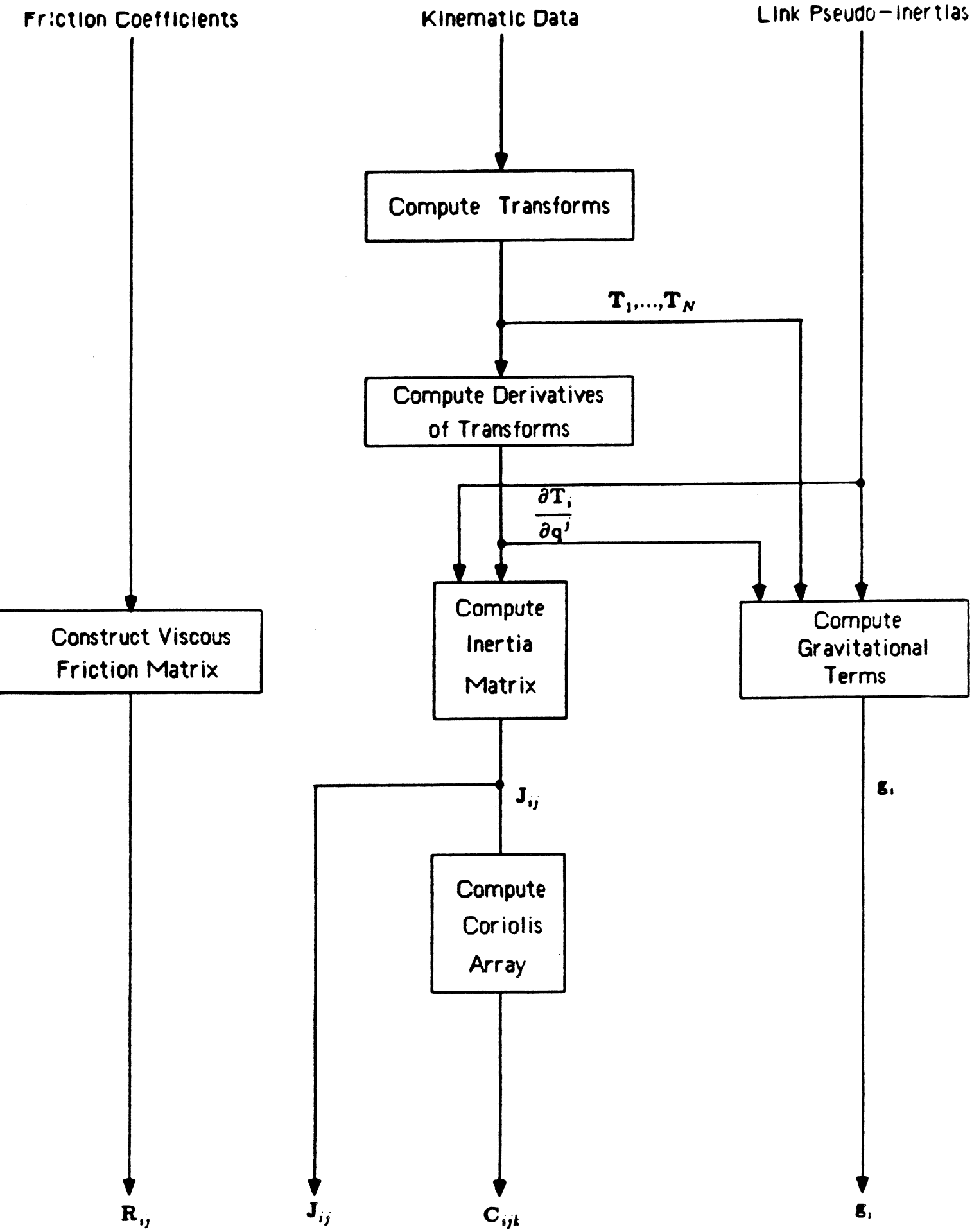


Figure 5. Schematic of Dynamic Equations Generator

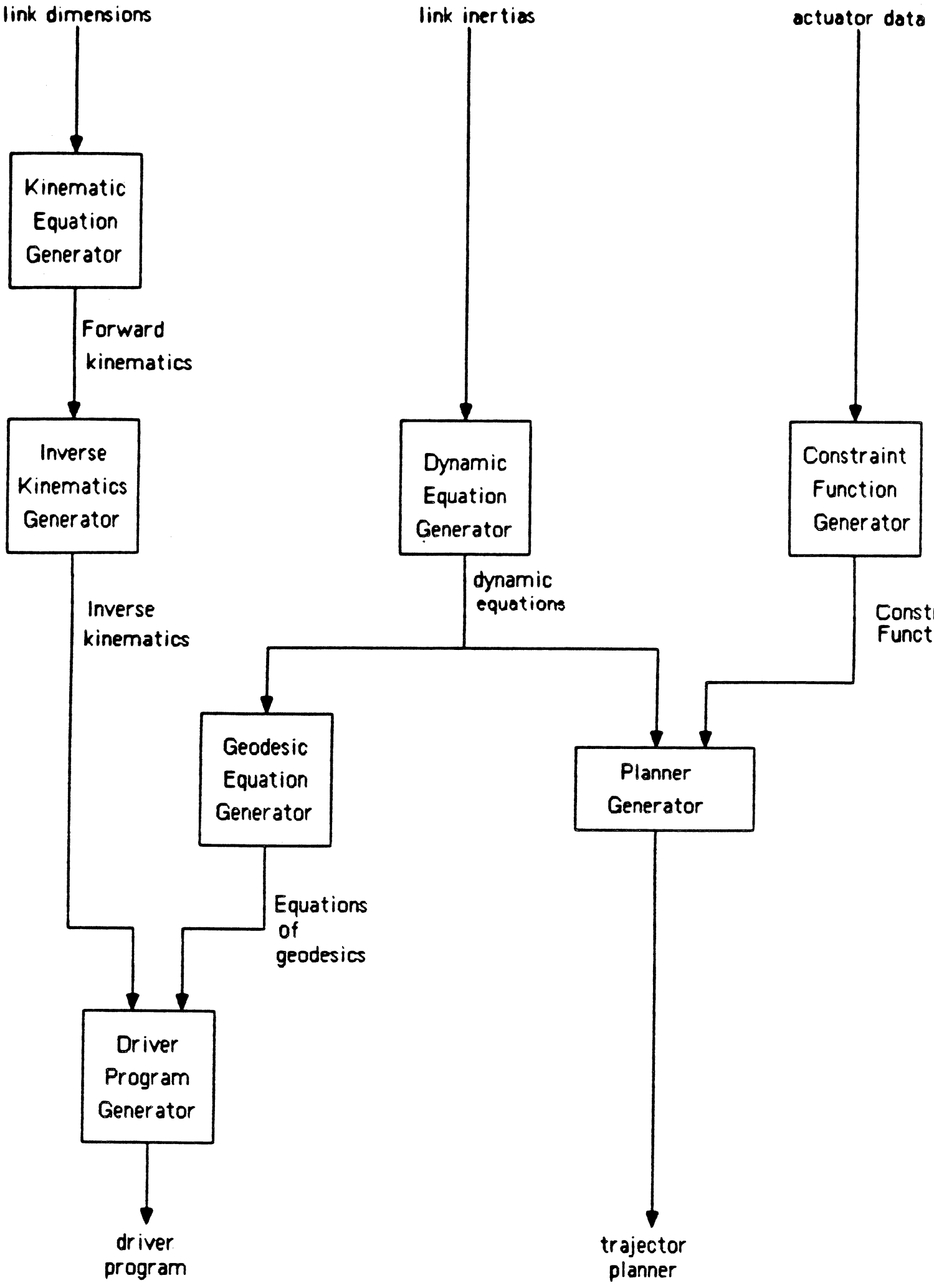


Figure 6. ATPG structure