(2)

QUARTERLY R&D STATUS REPORT
July 1991

A Program of Continuing Research on Representing, Manipulating, and Reasoning About Physical Objects

Contractor:	Cornell University
Contract Number:	ONR N00014-89-J-1946
Modification Number:	P00001
R&T Project:	4333001-06
ACO Code:	N62927 ONR, Code: 1133
Cage Code:	4B578
Contract Amount:	$320,000
Effective Date of Contract:	May 1, 1989
Principal Investigator:	John E. Hopcroft
Telephone Number:	607/255-7316
Email Address:	jeh@cs.cornell.edu
Reporting Period:	April–June, 1991

Description of Progress

Dexterous Manipulation

One of the great deficiencies of today's robots is their lack of flexibility. Most industrial robots are capable only of simple and repetitive tasks, such as spot welding or spray painting. There are two main reasons for this deficiency. First, typical end-effectors use a very simple two stick structure. Second, dexterous manipulation (manipulation by grippers with independently moving fingers) is not well understood. Effective techniques for dexterous manipulation would have application within a wide range of areas, including industrial assembly, decontamination of nuclear plants, and exploration of remote environments (*e.g.*, ocean bottoms or space).

We are developing a new strategy, *finger tracking* [Rus90,Rus91], for the autonomous manipulation of objects by multifinger robot hands. Most of the earlier efforts devoted to understanding dexterous manipulation have been devoted to grasping or to manipulation for task-specific problems. The finger tracking paradigm reorients an object with a series of rotations effected by fine finger motions, in which the hand maintains contact with the object at all times. It is common for humans to reorient an object using "extra" fingers — those not needed for grasping. Finger tracking captures and formalizes these ideas.

Our notion of manipulation refers to the reorientation of an object by a mechanical hand. The reorientation is accomplished by fine finger motions, with the object held in the hand through the entire process.

Definition 1 Manipulation *is the reorientation of a part inside the grip, while maintaining the grip.*

This definition implies that the reorientation is accomplished with respect to a system of coordinates which is fixed on the robot hand. In the process of reorientation, the grasped object undergoes a Euclidean motion, composed of a translation and a rotation. We are interested in measuring rotations, and therefore we abstract out the translational component of the motion.

Definition 2 *Two congruent objects have* equivalent orientation *if there is a pure translation that takes one into the other.*

We see two basic types of manipulation based on the relationship between the object and the hand. In the first kind, the object is passive inside the grip. The object is fixed with respect to the fingers involved in the grip in the reorientation step. Its motion follows the finger motions. We call the algorithms which result from such an interaction *finger walking algorithms.* In the second type of relationship, the object is active inside the grip. The internal forces of the grip are used to move the object relative to the grasping fingers. The manipulator produces motion by applying forces which take into consideration the geometry of the object. We call the manipulation algorithms which result from an active interaction between the fingers and the object *finger tracking algorithms.* Both finger walking strategies and finger tracking strategies are common in human manipulation. We have experimented with both strategies using Newton, our simulator for rigid-body dynamics. Our most recent efforts have been focused on finger tracking.

In order to make the idea of finger tracking precise, let O be the object to be manipulated and let H be the dexterous robot hand. We assume that H has enough fingers for a good grasp of the object. The hand also has some additional fingers, which we call *free fingers.* Some of the fingers are used to constrain the object by restricting its degrees of freedom, while the other fingers are used to generate its motion. In a typical manipulation problem, we are given the number of fingers n of the hand. Fingers from a subset of size $m \leq n - 1$ are used to grasp the object, usually by assigning each finger to a separate face. The size of m depends on the contact type [MNP90,MSS87,Ngu86]. Once the object is grasped, the m fingers stay fixed in space and the object is constrained to maintain continuous contact with them. In addition to the m grasping fingers that constrain the degrees of freedom, the object is also in contact with a free finger, which tracks along some curve on a different face. This process causes the object to move relative to the grasping fingers.

The free finger tracks a continuous trajectory, while at each instant, the $m + 1$ fingers hold the object in a grasp with some desired property, for instance equilibrium. Using this technique, we can generate the reorientation of a grasped object by commanding a simple, sliding motion for the tracking finger.

The most fundamental question within this framework is exactly how to generate some desired motion. The answer should allow us to program a robot to take an object from a given initial orientation to a goal orientation, or to determine when such a program does not exist.

To answer this question, we have broken the problem into two components. The first is related to the fact that this form of manipulation is defined as a constraint problem. Thus, an important aspect is finding an algorithm to determine the configuration space for the motion of the object to be manipulated. The second component has to do with finding the manner in which the robot must use its fingers to generate some desired trajectory for the object to be manipulated. This involves finding efficient tracking algorithms. We have established a framework in which to adress these algorithmic questions, by using Lie algebra properties.

Some of our results are summarized below:

- *The configuration space for a polyhedral object.* For the case of a polyhedral object held by a robot hand with four frictionless point contacts, we have obtained an algorithm to describe the configuration space as a manifold given by a closed form equation. We have analyzed the properties of the configuration space, and have shown that it is diffeomorphic to the rotation group $SO(3)$. Furthermore, we have shown that for this configuration space, the vertices of the polyhedron can move in a space-filling way. A consequence of this result is that the structure of the configuration space is quite complex, which makes finding finger tracking algorithms non-trivial.

- *Finger tracking for a polyhedral object.* Under the same assumptions as above, we have shown that the differential motion of the tracking finger is given by a 4×4 linear system. This surprising result is very feasible computationally, especially in the context of simulation.

- *Polygons in the plane.* Our newly developed framework for dexterous manipulation has been used to express earlier results [Rus90] for polygons; these results were originally obtained in a more *ad hoc* manner. Virtually the same algorithm used to determine the configuration space for polyhedra can be used to determine the configuration space for polygons.

- *Robustness for polygons.* The planar case has a number of geometric properties that we have been able to exploit in order to generate robust rotation algorithms. The uncertainty inherent in the real world makes robustness an important feature for any realistic manipulation algorithm. Our rotation algorithms are robust in the sense that some of the *a priori* knowledge requirements of the geometry of the object to be rotated and the necessity for precise calculations based on this geometry can be replaced by sensing. The result of our efforts is a condition on the geometry of the polygon to be rotated that guarantees robust rotations by an arbitrary rotation angle. We have shown that for convex polygons, the condition can be checked in $O(n)$ time, with $O(n \log n)$ preprocessing.

We are currently investigating the possibility of extending the robustness results from the planar, polygon case to the 3-dimensional, polyhedron case. We are also developing configuration space algorithms for 3-dimensional objects with curved faces. Another area in which we have made progress is the experimental verification of our results. We are using Newton, the simulator for rigid-body dynamics developed by our group, to verify our finger tracking algorithm for rotating polyhedra.

References

[MNP90] X. Markenscoff, L. Ni, C. Papadimitriou, *The geometry of form closure*, IJRR, Feb 1990.

[MSS87] B. Mishra, J. T. Schwartz and M. Sharir, *On the existence and synthesis of multifinger positive grips*, Algorithmica, 2, 1987.

[Ngu86] V. Nguyen, *The synthesis of stable force-closure grasps*, Technical Report MIT AI-TR-905, MIT AI Lab, 1986.

[Rus90] D. Rus, *Dexterous rotations of polygons*, Proceedings, Second Canadian Computational Geometry Conference, 1990.

[Rus91] D. Rus, *A framework for dexterous manipulation using Lie algebras*, to appear, LNCS – Proceedings, Second Algebraic Methodologies And Software Technology Conference, 1991.

Presentations, Publications, and Reports

Presentations

1. *Integrating Symbolic and Numeric Computing*, New Directions in Computing Symposium, NASA Ames/University of Illinois at Chicago, April 8, 1991. Richard Zippel.

2. *Functional Decomposition*, University of Toronto, Dept. of Computer Science, April 25, 1991. Richard Zippel.

3. *A Framework for Dexterous Manipulation using Lie Algebras*, 2nd Algebraic Methodologies and Software Technology Conference, Iowa City, May 1991. Daniela Rus.

4. *The Role of Simulation in Science and Engineering*, Opening of the Theory Center, Cornell University, June 4, 1991. John Hopcroft.

5. *Shape Control in Implicit Modeling*, Graphics Interface '91, Calgary, Canada, June 3-7, 1991. Baining Guo.

6. *Entering the Information Age*, Seattle University, June 21, 1991. John Hopcroft.

7. *Robust Geometric Algorithms*, Seattle Pacific University, June 22, 1991. John Hopcroft.

8. *Least Constraint: A Framework for the Control of Complex Mechanical Systems*, American Control Conference, Boston, June 27, 1991. Dinesh Pai.

Publications

1. Least Constraint: A Framework for the Control of Complex Mechanical Systems, *Proceedings of the American Control Conference*, American Automatic Control Council, 1991, 1615–1621. Dinesh K. Pai.

2. Shape Control in Implicit Modeling, *Proceedings of Graphics Interface '91*, 230–235. Baining Guo.

3. Automatic Surface Generation using Implicit Cubics, *Scientific Visualization of Physical Phenomena*, edited by N. M. Patrikalakis, Springer-Verlag. Baining Guo.

Reports

1. *Beyond Keyframing: An Algorithmic Approach to Animation*, Department of Computer Science Tech Report 91–1207, Cornell University, May 1991. A. James Stewart and James F. Cremer.

2. *Robust Point Location in Approximate Polygons*, Department of Computer Science Tech Report 91–1208, Cornell University, May 1991. A. James Stewart.

3. *Rational Function Decomposition*, Department of Computer Science Tech Report 91–1209, Cornell University, May 1991. Richard Zippel.

4. *Symbolic/Numeric Techniques in Modeling and Simulation*, Department of Computer Science Tech Report 91–1214, Cornell University, June 1991. Richard Zippel.

in Proc. of 1991 American Control Conferenc
Boston

LEAST CONSTRAINT: A Framework for the Control of Complex Mechanical Systems

Dinesh K. Pai*
Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

We describe an approach to control in which control actions are specified as weakly as desired. We use large time-varying sets of non-zero measure as desired goals instead of specific trajectories, maintaining that we do not care where in such a region we actually are. Inequality constraints and their conjunctions are used to describe such regions. The constraints are satisfied at run time to produce the control. The approach has been successfully used to produce human-like walking in simulation.

We also describe an implemented programming environment for this approach. We discuss the representation of control computations using computational graphs and automatic differentiation for efficiency. Constraint satisfaction is performed using a fast relaxation method.

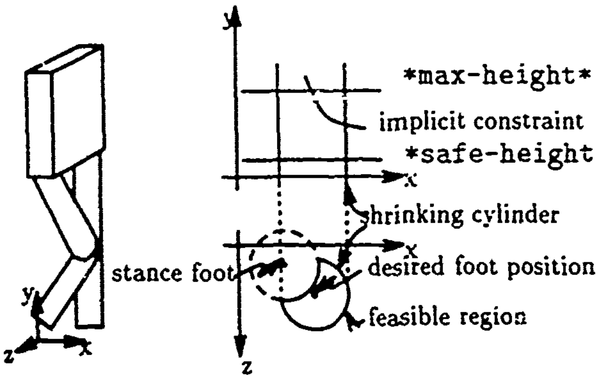


Figure 1: Biped walking machine

1 Introduction

We are concerned with controlling high degree of freedom mechanical systems which have to accomplish several simultaneous tasks. Such systems include robot arms, multi-fingered hands, walking machines, mobile robots, and simulated mechanical systems used in computer animation. The system typically has redundant degrees of freedom for each task, but may have to accomplish a large number of tasks simultaneously. The system may also be autonomous and reactive, which means that a large amount of the "programming" will be done at execution time.

The complexity of the system implies the following:

1. The ease with which complex tasks are expressed and composed is critical.
2. The efficiency of the control computations is very important.
3. Simulation of the mechanical system is necessary to gain insight into the control programs and to aid their development.

These demands are frequently contradictory. We propose a framework called "Least Constraint" (abbreviated as LC) which we believe facilitates the expression of control programs for complex mechanical systems and is efficiently implementable as well. An early version of the framework was presented in [29].

Section 2 describes the LC approach and how motions are expressed in it. Section 3 discusses the data structures for representing control computations in LC and the algorithms used to perform these computations efficiently. Section 4 describes the solution of the inequality constraints that arise.

2 The Least Constraint Approach

2.1 Motivation

Consider the problem of controlling the human-like walking machine of Figure 1 to walk dynamically in three dimensions.

One approach to programming such a task is to pick some periodic trajectory for the joints, and attempt to track it. However, it is not clear that this is the natural characterization of the task. Indeed in problems of this type, a major goal of the process of developing a control program is to discover the task requirements.

We would instead like to program such machines incrementally, by specifying assertions about its behavior. We can specify several requirements for walking, for in-

*Supported in part by ONR Grant N00014-88K-0591, ONR Grant N00014-89J-1946, and NSF Grant DMC-86-17355. The author would like to thank L.-M. Reissell for innumerable discussions and J Cremer for comments at a short notice.

stance, (i) the foot should clear the ground during the swing phase of the leg, (ii) the swing foot should be moved to a location suitable for dynamic balance by foot placement, and so on.

When the machine is controlled to satisfy these requirements, it may turn out that the requirements were inadequate — for example, one may find that there is nothing to prevent knee flexion from becoming so large that walking is impossible. In this case one would like to modify the existing program by *merely adding new assertions*: for example, by adding the assertion that the pelvis should be above a certain height. This is not possible in current robot programming languages. The LC framework was designed to address these problems.

2.2 Least Constraint Framework

In the LC framework, motions are expressed by means of time- and state-dependent assertions. These assertions are defined using inequality constraints which describe the set of allowed states as a function of time. The constraints are solved at run time to produce a motion satisfying them.

Since complex mechanical systems have large state spaces, it is not convenient or natural to express all of the constraints in a single space. For instance, the walking machine in Figure 1 has a 28-dimensional state space. For convenience of expression, users define derived variables in terms of the basic (e.g., state) variables — an example of this is the definition of task and end-effector coordinates for robot manipulators. LC generalizes such constructions to allow the creation of arbitrary, user definable quantities which are natural to the tasks and the constraints being expressed. One can isolate small groups of variables into domains on which to focus. For example, the foot collision constraints in the above walking example are best expressed in a separate foot position domain.

In LC, users define a *domain system*, $\{\mathcal{D}_i \; . \; i \in I\}$, related by *linking functions*

$$l_{ij} : \mathcal{D}_i \to \mathcal{D}_j, (i, j) \in L \subset I \times I, \qquad (1)$$

which satisfy the basic consistency condition that all diagrams of the following form commute.

$$\begin{tikzcd}
\mathcal{D}_k \arrow[r, "l_{kl}"] & \cdots\cdots \arrow[r, "l_{mj}"] & \mathcal{D}_j \\
\cdots \arrow[u, "l_{ok}"] & & \cdots \arrow[u, "l_{qj}"'] \\
& \mathcal{D}_i \arrow[ul, "l_{in}"] \arrow[ur, "l_{ip}"'] &
\end{tikzcd}$$

All domains $\mathcal{D}_i$ are connected to a *basic domain* $\mathcal{D}_0$ by compositions of linking functions :

$$\mathcal{D}_0 \xrightarrow{l_{0j}} \cdots \xrightarrow{l_{ki}} \mathcal{D}_i.$$

Briefly, the motivation for using domain systems is that they allow a constraint on a subdomain $\mathcal{D}_i$ to be lifted to an equivalent constraint on the basic domain $\mathcal{D}_0$ using compositions of linking functions.

The topological properties of the domains $\mathcal{D}_i$ are left somewhat open — in theory, a domain system could be expressed in terms of arbitrary manifolds; in practice, however, the domains will be copies of $\Re^n$, for varying n. Non-Euclidean domains, such as SO(3), are currently treated using their coordinate charts.

A *motion specification* in LC now consists of a system of time-varying inequality constraints $P_\alpha, \alpha \in A$, on the domains $\mathcal{D}_i$, here each constraint P_α is expressed by

$$P_\alpha := f_\alpha(x(t), t) \leq 0, \qquad (2)$$

where

$$f_\alpha : \mathcal{D}_i \times \Re \to \Re, \qquad \alpha \in A$$

is a smooth map, and $x(t)$ denotes a time-dependent trajectory in $\mathcal{D}_i$.

The interpretation of the constraint function is that the system is controlled to make the specified expressions $P_\alpha := f_\alpha(x(t), t) \leq 0$ true at all times t.

Such P_α and their conjunctions

$$P := \bigwedge_\alpha P_\alpha \qquad (3)$$

are executable LC motion programs.

As an example, Figure 1 depicts some constraints on the position of the foot for the walking machine.

The solution of the constraints is achieved by producing a trajectory $x(t) \in \mathcal{D}_0$ such that the derived constraints

$$P_\alpha := f_\alpha((l_{ki} \circ \ldots \circ l_{0j})(x(t)), t) \leq 0$$

obtained by lifting the original constraints using the linking maps are satisfied at all times t. In LC, this is done at run time at discrete time steps t_k, at every time step t_k, a feasible point $x(t_k)$ is produced, and is used to compute the control u.

We have implemented a programming environment using Common Lisp and the X window system to develop and execute LC programs. In this environment, users define variables, domains and linking functions, and express constraints in these domains. Exact partial derivatives can be efficiently computed using automatic differentiation. The environment interacts with a simulator of rigid body mechanics, "Newton" [8,14]. LC accepts models of multi-body mechanical systems described in Newton and generates the computational graph (see Section 3) for the simultaneous computation of forward kinematics of user specified features on each body, differential kinematics, and inverse dynamics. The control programs can be tested by sending the control output produced by LC to the simulator. Tools are provided for debugging control programs and for visualizing constraints. Finally, the control programs can be translated into straight line programs in other languages (currently Lisp and C) for efficiency.

2.3 Examples

We have deliberately left open issues such as how the system is used, the nature of the constraints, and the choice

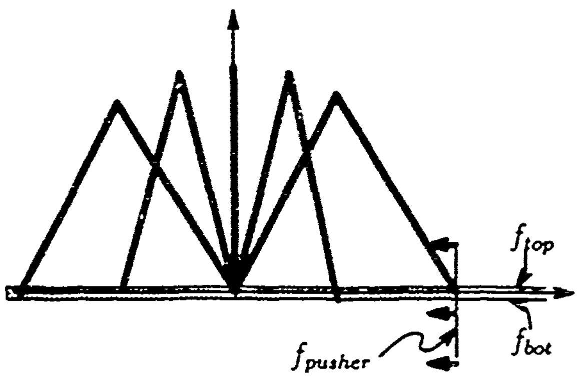


Figure 2: Toleranced move near singularity.

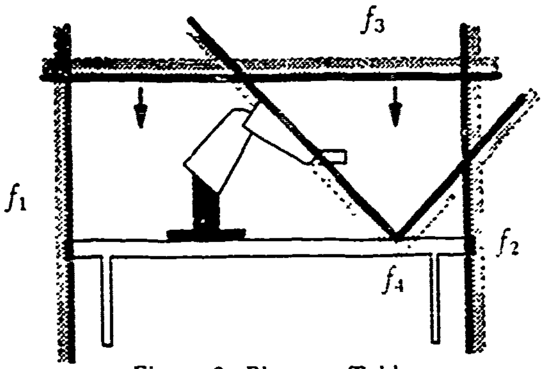


Figure 3: Place on Table

of basic variables. LC is intended to be a general programming framework, rather than a solution to a specific problem. In this section, we present examples of the application of LC to robot programming. The examples are simple for the sake of presentation, but illustrate the use of LC.

Example 1. Figure 2 depicts a 2-link planar robot manipulator which is to be controlled to move along a straight line passing through the singular configuration (corresponding to the end-effector location (0,0)), from the point (1.0, 0.0) to (−1.0, 0.0) with an average speed of at least v. We assume that this is a pure kinematic problem — i.e., the basic variables are the joint angles t_1 and t_2, which are tracked by a separate low level controller. We are allowed a tolerance of ϵ around the nominal straight line.

The assertion that the end-effector position (x_2, y_2) is within ϵ of the nominal straight line is expressed by the constraint functions

$$f_{top}(y_2) = y_2 - \epsilon \quad (4)$$
$$f_{bot}(y_2) = -y_2 - \epsilon \quad (5)$$

and finally the end-effector is "pushed" along the tube by specifying the constraint function

$$f_{pusher}(x_2, t) = (x_2 - 1.0) - vt \quad (6)$$

Our constraint satisfaction procedure is robust near singularities (Section 4) and the robot executes the pose-changing motion shown.

Example 2. Figure 3 depicts the task of placing an object in the robot's hand on the table. In a robot programming language such as VAL [34], this will have to be expressed by arbitrarily commanding a particular motion to an arbitrary point on the table. In LC, the task is programmed by placing constraints in the hand position domain. The hand is constrained to stay within the boundaries of the table by the constraints f_1 and f_2. The constraint f_3 moves down at constant speed v from height h, so that after a time of h/v the end-effector is on the table top. Note that the exact location on the table top is unspecified.

If we now wish to place the object at a particular location on the table, this is achieved by adding the cone-shaped constraint marked f_4 in Figure 3. If it is required that the center of mass of the arm be above a small region of the base during the motion, a new variable x_{cm} is defined in terms of the joint angles and base position, and the constraints $x_l \leq x_{cm} \leq x_r$.

Example 3. We have used the LC framework to successfully program the human-like machine in Figure 1 to walk dynamically in three dimensions [30]. In each qualitative state of the machine, such as "standing on the left leg while stepping with the right," constraints are imposed to achieve a large number of simultaneous tasks such as foot placement for dynamic balance, torso orientation, maintenance of pelvis height, collision avoidance with the ground during swing, inter-leg collision avoidance, joint limit avoidance, etc.

2.4 Discussion

One of the main advantages of the LC framework is that it enables and perhaps encourages subtasks to be expressed weakly. This reduces the number of arbitrary decisions which have to be made, such as arbitrarily picking a trajectory in Example 2. The program reflects the user's intention better and is easier to maintain. It is also easier to do "redundancy maintenance," i.e., to retain the excess degrees of freedom available to perform a task.

LC introduces certain object-oriented features to robot programming. Motion programs are easy to combine, inherit, and specialize for new tasks. This is particularly important for complex systems for which programs are developed incrementally (see Examples 2 and 3.)

Typical users will use libraries of higher-level compound programs for common LC idioms. The following is a brief list of such idioms.

- The *obstacle*: These constraints encode the free space in the environment (e.g., [21,4,10,7]). Care must be taken to grow the constraints or to impose velocity constraints in order to account for the braking characteristics of the manipulator near the obstacle.

- The *interval*: This is a special case of the obstacle idiom, and restricts the range of a variable, as in the case of joint angle limits [17], speed limits, etc.
- The *pusher*: This a time-dependent constraint which moves the system in a given direction, without restricting motion orthogonal to this direction. Equation 6 is an example.
- The *funnel* [23]: Here the constraints define a set which contracts over time. Thus the system can be brought to a desired configuration without overly restricting its trajectory. A canonical example of a funnel is a contracting ball.
- The *toleranced move*: A moving ball with fixed radius is an example. Example 1 provides another instance of a toleranced motion.

2.5 Relationship to other approaches

Current approaches to programming complex mechanical systems may be broadly classified as follows:

- Explicit approaches: These consist of approaches which allow users to explicitly specify the motion of the robot, e.g., by prescribing trajectories. These include robot-level programming languages (e.g., [34,33]). While these approaches allow fine control of the motion, they are hard to program and force users to make arbitrary decisions in order to execute a motion.
- Implicit approaches: In these approaches, the users only specify the high-level, global goals of the motion and the system plans a motion that achieves the goals. These include the approaches of motion planning (e.g., [21,15,22,11,9,20]) and optimal trajectory planning [3,32,6]. These methods are powerful when they are well matched to the problem. However this is frequently not the case; for example, it may not be important that the trajectory minimize a specific functional such as energy along the trajectory. More generally, these approaches do not facilitate modification of the planned motions by the users. Finally, they are typically computationally expensive and need to be executed off-line.

LC is an intermediate approach between the explicit and implicit, sharing some features of both.

LC is higher level than the explicit approaches — one need not specify motions explicitly but rather more weakly as a set of constraints. On the other hand, by moving the constraints and restricting the feasible set, one has a degree of explicit user control on the motion.

LC is lower level than the implicit approaches. It has no built-in application specific knowledge. The constraints are satisfied locally, and LC cannot guarantee that the motion generated at one time will not cause a failure (e.g., there may be no feasible point in a small neighborhood of the current state) at some time in the future. An additional planning layer may be necessary to avoid such situations. On the other hand, for problems such as controlling reactive, autonomous systems (e.g., [5]), this lack of guarantees is not a critical issue.

There is a close connection between satisfying constraints and avoiding obstacles — obstacles are "hard" inequality constraints. Conversely, one can think of constraints as being virtual obstacles in abstract spaces, which can change and move over time under the programmer's control. In particular, our approach shares several features with the use of artificial potential fields, proposed by Khatib [17] (see also, for example, [12,27,18]). However, there are important differences.

First, LC generalizes the notion of obstacles to constraints in arbitrary, user-defined domains. This is supported by a programming framework to describe these constraints conveniently, and a constraint satisfaction system which solves the constraints. Thus LC should be applicable to a broad range of motion control tasks.

Another difference is the specification of motions outside the natural constraints imposed by obstacles and joint limits. In potential function approaches the motion is specified by constructing a scalar function ϕ such that the system behaves like a gradient dynamical system with ϕ as potential. The specification of a motion using a potential function is concise — a single scalar function encodes global dynamic behavior of the system. However, this has the drawback that it is difficult for users to specify potential functions for complicated behaviors. Thus the potential functions encountered in the literature are extremely simple or are generated by special purpose planning programs (see [19]). In LC motions are specified by time varying constraints. Each constraint has an intuitive meaning as an assertion. The construction of complicated potential functions by adding simpler potentials together does not necessarily result in easily predictable behavior; but, joining two constraints will always produce motion which satisfies both constraints. The state is manipulated by time-varying constraints in a manner reminiscent of pushing operations [24] and may be viewed as a generalization of pushing to user-defined spaces.

We believe that these features make constraints easier to specify and visualize than potential functions.

3 Representing Computations

3.1 Computational Graph

The domain system of Section 2.2 is implemented in LC as a computational graph (or Kantorovich graph) [31,16]. In a typical constraint satisfaction computation, one needs to compute the value of each constraint, and the gradients of the violated constraint functions. These computations are efficiently performed using the technique of automatic differentiation. Computations of derived variables, especially for kinematics, differential kinematics, and dynamics, contain many common subexpressions, the elimination of which is also a major source of efficiency. The computational graph is a useful data structure for achieving these goals.

A computational graph can be described as follows (see Figure 4 for example). A function $f : \Re^m \to \Re^n$ is said to be *factorable* if every component of f is a function computable from the basic and derived variables by means of

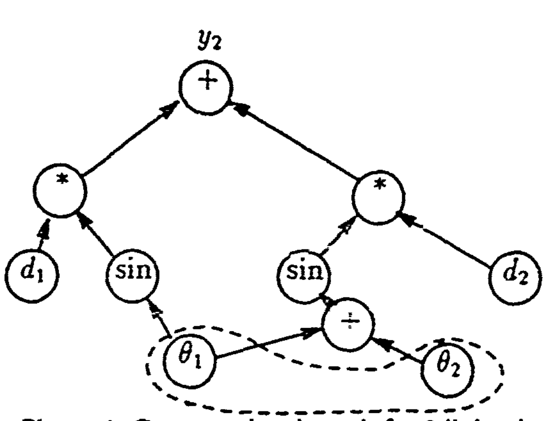
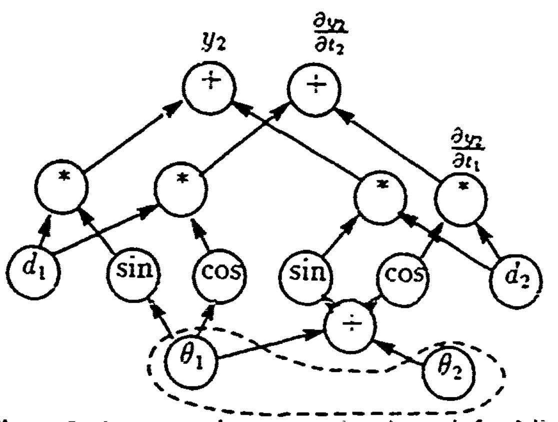


Figure 4: Computational graph for 2-link robot



y_2
$\frac{\partial y_2}{\partial t_2}$
$\frac{\partial y_2}{\partial t_1}$
+
÷
*
*
*
*
d_1
sin
cos
sin
cos
d_2
÷
θ_1
θ_2

Figure 5: Augmented computational graph for 2-link robot

a finite sequence of elementary operations.

We represent a factorable function as a graph as follows: let V_I be the set of m vertices corresponding to the m coordinates of the domain of f, V_O be vertices corresponding to the n coordinates of the range, and V_U be the intermediate quantities in the computation of f. Let $G(V, E)$ be a directed graph with vertex set $V = \{V_I \cup V_U \cup V_O\}$ and edge set E. Let ψ be the set of basic operations. With each vertex we associate an operation by the function $\omega : V \rightarrow \psi$ with the understanding that the operation $\omega(v)$ is applied to the ordered list of immediate predecessors of v in G.

Figure 4 depicts a computational graph for the y-coordinate of the end-effector position of a 2-link robot manipulator.

Vertices of the graph are grouped together to form domains, and the subgraph connecting two domains defines their linking function. The basic domain $\mathcal{D}_0$ consists of all the independent variables in the system's computational graph, i.e. the set of all vertices with in-degree 0, which are not marked as constants. Frequently, the basic domain is identical to the state space of the system.

3.2 Automatic Differentiation

Automatic differentiation is a technique for efficiently computing exact derivatives of factorable functions (see [31,16,13] for recent surveys). It differs from standard symbolic differentiation in that by using the underlying computational graph data structure, the growth of common subexpressions due to differentiation is automatically controlled; it differs from numerical differentiation in that the method is exact and results in no loss of significant digits.

We have implemented both forward and reverse modes of automatic differentiation. The computation of partial derivatives is implemented as augmentation of the original computational graph to produce a new graph G' that computes both the function and its derivatives. This leads to additional savings since the expressions for the partials are frequently already present in the computational graph and are found using simple expression matching algorithms. Figure 5 depicts the augmented computational graph of the 2-link robot kinematics example in Figure 4. In the example $\partial y_2/\partial t_1$ is the same as x_2, the x-coordinate of the end-effector position, and need not be recomputed.

In the reverse mode, the additional cost of computing all the partials of a function is very small. An upper bound was derived by Baur and Strassen [2,25]:
Let f be a rational function of m variables $x_1, \ldots, x_m$ for which the computational graph has $\tau(G)$ vertices, of which $\mu(G)$ are multiplications/divisions. Then

$$\begin{aligned} \tau(G') &\leq 5\tau(G) \\ \mu(G') &\leq 3\mu(G) \end{aligned} \qquad (7)$$

Thus the cost of computing all the partials of a scalar function is at most a small constant multiple of the cost of merely evaluating the function. The extension to transcendental functions and exponentiation is straightforward [25]. Our experience is that the above bounds are conservative. Also, it is frequently the case that the function has been computed before the gradient is required, in which case the cost of computing the gradient is reduced by $\tau(G)$ (or $\mu(G)$).

4 Constraint Satisfaction

LC separates the specification of constraints and the technique used for satisfying them. Thus different constraint satisfaction algorithms can be used. We describe here a non-linear extension of the relaxation method [1], which can efficiently handle constraints in multiple domains

The method is local and iterative. The local nature of the method makes it fast enough to be implementable in real time. It exploits the knowledge of a good starting point for the iteration — the solution to the constraint satisfaction problem at the previous time step.

4.1 Constraints in a single domain

First consider the case in which all constraints are expressed in one domain. The method we use for constraint satisfaction is similar to the relaxation method for linear inequalities [1], but is extended to nonlinear inequalities using Newton's method.

Let $\mathcal{D} = \Re^m$ be the domain, $x \in \mathcal{D}$, and let the system of inequality constraints be

$$f_i(x) \leq 0, i = 1, \ldots, n. \tag{8}$$

Let x^k be the kth iterate at time-step t, and

$$n_i = \frac{\nabla f_i(x^k)}{|\nabla f_i(x^k)|} \tag{9}$$

$$d_i = \frac{f_i(x^k)}{|\nabla f_i(x^k)|} \tag{10}$$

Then each iteration is:

1. For each $i = 1, \ldots, n$ compute $f_i(x^k)$. If all $f_i(x^k) \leq 0$, terminate.
2. Else, let j be such that $d_j = \max_i d_i$.

$$x^{k+1} = x^k - \rho(d_j + o)n_j \tag{11}$$

$\rho \in (0, 2]$ is the relaxation parameter, and o is a small offset.

The behavior of the above algorithm has been analyzed in the case of linear f_i by [1,26]. The algorithm is then globally convergent, with positive offsets o having the effect of inducing convergence in a finite number of steps. The behavior in the non-linear case is similar in a sufficiently small neighborhood of the feasible set. The global convergence property is lost, but this is not a serious problem since we are typically solving the inequalities starting near the feasible set. Finally, the convergence can be linear with a large convergence ratio when the inequalities are ill conditioned. Choosing relaxation parameter $\rho > 1$ can significantly improve convergence. We are currently investigating the tradeoffs with other methods that have super-linear convergence but are more expensive per iteration.

The algorithm works well in practice. Step 1 of each iteration can be computed in parallel to check violated constraints. The gradient required in equations 9 and 10 is efficiently computed using automatic differentiation, and utilizes the effort already expended in computing f_i.

4.2 Constraints in multiple domains

Since LC users specify constraints in different domains, possibly related by linking functions, the single domain case above has to be extended to handle multiple domains. Consider the case of two domains $\mathcal{D}_1$ and $\mathcal{D}_2$, and further assume that $\mathcal{D}_1$ is defined in terms of $\mathcal{D}_2$ by a smooth linking map $l : \mathcal{D}_2 \rightarrow \mathcal{D}_1$. Let f^1 be a constraint in $\mathcal{D}_1$ and f^2 be a constraint in $\mathcal{D}_2$.

It is computationally advantageous to perform the descent in $\mathcal{D}_2$. We lift f^1 to an equivalent constraint $\bar{f}^1 = f^1 \circ l$ in $\mathcal{D}_2$. Thus $\nabla \bar{f}^1 = Dl^T \nabla f^1$, making the method robust near singularities.

Also, it is not necessary to explicitly compute the matrix Dl^T — we directly compute $\nabla \bar{f}^1$ by differentiating $\bar{f}^1$ using automatic differentiation saving the cost of the $\dim \mathcal{D}_1 \times \dim \mathcal{D}_2$ multiplications for the matrix-vector product $Dl^T \nabla f^1$.

This method is preferable to methods which choose the steepest descent and the estimate of the distance in the specification domain $\mathcal{D}_1$. Although the latter choice makes it easier for the user to anticipate the behavior of the constraint satisfaction algorithm, it is also more expensive, in effect requiring the computation of l^{-1} or Dl^{-1}. This also leads to numerical problems when the Jacobian matrix Dl is singular or ill conditioned. This is important in robotics applications, where motion in the vicinity of kinematic singularities is sometimes desirable or inevitable [28].

The general situation involves constraints expressed in several related, and possibly overlapping, domains and is handled similarly by lifting each constraint function f_i in domain $\mathcal{D}_i$ to the basic domain $\mathcal{D}_0$ by using a composition of the linking functions provided.

4.3 Cost per iteration

Let c_i be the cost, in multiplications, of computing each of the n constraint functions f_i. The cost of computing all the f_i is $V \leq \sum_{i=1}^{n} c_i$, where the "<" case arises if the f_i share computations. The test of constraint violation in Step 1 of the iteration costs V and this is fixed by specification of the constraints by the user. Let $\mathcal{A} = \{i | f_i > 0\}$ be the active or violated constraints. Then the update in Step 2 costs

$$U \leq \sum_{i \in \mathcal{A}} 2c_i. \tag{12}$$

Here we assume that $c_i \gg \dim \mathcal{D}_0$ and terms of the order of $\dim \mathcal{D}_0$ are ignored. The cost of computing the gradient is actually bounded by $3c_i$, but of this c_i has already been accounted for in V. The total cost of each iteration is $V + U$.

Note that U is very small when only a few constraints are violated, and is at most twice the cost of merely checking if any constraints are violated.

5 Conclusions

The basic idea of the approach to control described in this paper is "Least Constraint" — by which we mean that we specify control actions as weakly as desired. This permits motions specifications to better reflect the user's intentions, and makes the programs easier to maintain. Another contribution of this work is our use of computational graphs for the efficient computation of a constraint function and it's derivatives. This leads to an inexpensive constraint satisfaction algorithm that has the auxiliary benefit that it is robust near singularities.

We have used the LC framework on a small range of problems, including the task of programming a simulated human-like biped machine to walk dynamically. Our experience indicates that LC is a useful programming approach for complex systems with several constraints. However, the method can perform poorly when the constraints are ill-conditioned and the feasible set is small and disconnected. For such cases, we are considering methods with super-linear convergence properties and for selecting viable connected components of the feasible set using simulation.

Finally, we would like to suggest that LC may be a suitable language in which high level planners express motions. The plans produced can then be easily augmented by sensors or tweaked by users if necessary.

References

[1] S. Agmon. The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, 6(3):382 - 392, 1954.

[2] W. Baur and V. Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22:317–330, 1983.

[3] J. Bobrow, S. Dubowsky, and J. Gibson. Time-optimal control of robot manipulators. *International Journal of Robotics Research*, 4(3), 1985.

[4] R. A. Brooks. Solving the find-path problem by good representation of free space. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(3):190–196, 1983.

[5] R. A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. In *IEEE International Conference on Robotics and Automation*, pages 692–694, 1989.

[6] J. Canny, B. R. Donald, J. Reif, and P. Xavier. On the complexity of kinodynamic planning. In *Symposium on the Foundations of Computer Science*, 1988.

[7] J. F. Canny. *The Complexity of Robot Motion Planning.* PhD thesis, M. I. T., 1987.

[8] J. F. Cremer. *An Architecture for General Purpose Physical System Simulation - Integrating Geometry, Dynamics, and Control.* PhD thesis, Cornell Univesity, 1989.

[9] B. R. Donald. *Error Detection and Recovery for Robot Motion Planning with Uncertainty.* PhD thesis, Massachusetts Institute of Technology, July 1987.

[10] B. R. Donald. A search algorithm for motion planning with six degrees of freedom. *Artificial Intelligence*, 31(3), 1987.

[11] M. A. Erdmann. On motion planning with uncertainty. Technical Report 810, MIT AI Laboratory, 1984.

[12] B. Faverjon and P. Tournassoud. A local based approach for path planning of manipulators with a high number of degrees of freedom. In *IEEE International Conference on Robotics and Automation*, pages 1152–1159, 1987.

[13] A. Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

[14] C. M. Hoffmann and J. E. Hopcroft. Simulation of physical systems from geometric models. *IEEE Journal of Robotics and Automation*, RA-3(3):194–206, June 1987.

[15] J. E. Hopcroft, J. T. Schwartz, and M. Sharir. On the complexity of motion planning for multiple independent objects: PSPACE hardness of the "Warehouseman's Problem". *International Journal of Robotics Research*, 3(4):76–88, 1984.

[16] M. Iri and K. Kubota. Methods of fast automatic differentiation and applications. Research Memorandum RMI 87-02, Dept. of Mathematical Engineering and Instrumentation Physics, University of Tokyo, Japan, 1987.

[17] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1):90 - 98, 1986.

[18] D. E. Koditschek. Robot planning and control via potential functions. In O. Khatib, J. J. Craig, and T. Lozano-Pérez, editors, *The Robotics Review*. MIT Press, 1989.

[19] D. E. Koditschek and E. Rimon. Robot navigation functions on manifolds with boundary. Center for Systems Science, Yale University. (To appear in *Advances in Applied Mathematics*), July 1988.

[20] J. C. Latombe. Motion planning with uncertainty: The preimage backchaining approach. Technical Report STAN-CS-88-1196, Stanford University, Department of Computer Science, 1988.

[21] T. Lozano-Pérez. *Robot Motion: Planning and Control*, chapter 6. The MIT Press, 1982.

[22] T. Lozano-Pérez, M. Mason, and R. Taylor. Automatic synthesis of fine-motion strategies for robots. *International Journal of Robotics Research*, 3(1), 1984.

[23] M. T. Mason. The mechanics of manipulation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 544–548, 1985.

[24] M. T. Mason and J. K. Salisbury, Jr. *Robot Hands and the Mechanics of Manipulation.* M.I.T. Press, 1985.

[25] J. Morgenstern. How to compute fast a function and all its derivatives. *SIGACT News*, 14(4):60–62, 1985.

[26] T. Motzkin and I. J. Schoenberg. The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, 6(3):393 - 404, 1954.

[27] W. S. Newman and N. Hogan. High speed robot control and obstacle avoidance using dynamic potential functions. In *IEEE International Conference on Robotics and Automation*, pages 14–24, 1987.

[28] D. K. Pai. *Singularity, Uncertainty and Compliance of Robot Manipulators.* PhD thesis, Cornell University, Ithaca, NY, May 1988.

[29] D. K. Pai. Programming parallel distributed control of complex systems. In *Proceedings of the IEEE International Symposium on Intelligent Control*, pages 426–432, September 1989.

[30] D. K. Pai. Programming anthropoid walking: Control and simulation. Cornell Computer Science Tech Report TR 90-1178, 1990.

[31] L. B. Rall. *Automatic Differentiation.* Lecture Notes in Computer Science. Springer-Verlag, 1981.

[32] G. Sahar and J. Hollerbach. Planning minimum-time trajectories for robot arms. In *IEEE International Conference on Robotics and Automation*, 1985.

[33] R. H. Taylor, P. D. Summers, and J. M. Meyer. AML. A manufacturing language. *International Journal of Robotics Research*, 1(3):19–41, 1983.

[34] Unimation Inc. User's guide to val, 1983.

Shape Control in Implicit Modeling

Baining Guo*
Department of Computer Science
Cornell University
Ithaca, New York 14853, USA

Abstract

Recently, implicit patches have emerged as alternative modeling primitives for three dimensional objects In designing three dimensional models, one often encounters various shape requests. This paper develops techniques for satisfying such requests through shape control. In particular, we show how to achieve the convexity of quadric patches or cubic patches.

1 Introduction

The end goal of geometric modeling is to design and to manipulate three dimensional models represented by free-form surfaces. Traditionally, free-form surfaces are built from parametric patches. Parametric patches are successful as far as design and rendering are considered, but manipulating three dimensional models with parametric pathes poses fundamental difficulties. For example, parametric patches are not closed under sweeping and convolution. The intersection of two parametric patches are extremely difficult to represent and evaluate [HK86].

One way to avoid these problems is to build free-form surfaces from low-degree implicit patches. Implicit patches are closed under all common operations in geometric modeling [Baj88], and the intersections of low-degree implicit patches can be computed efficiently[OSS]. Recent research shows that quadric and cubic implicit patches are flexible enough for building arbitrary three dimensional models [Guo90, Guo91].

A major reason that parametric patches have become so popular in computer graphics is their good shape control properties. In this paper, we tackle the shape control issues of implicit patches. Using Bernstein-Bezier representation of polynomials, we can control the shapes of implicit patches through manipulating their control points.

*This work is supported by DARPA under ONR contract N00014-86K-0591, NSF Grant DMC-86-17355, and ONR Grant N00014-89J- 1946.

In designing free-form surfaces, one often encounters various shape requirements, such as a nice pattern of reflection lines and restrictions on the minimum radius of curvature. Among all the shape requirements, convexity is the most basic and the most frequently requested one. In this paper, we show how to manipulate the control points of a quadric patch or a cubic patch so that the patch become convex.

1.1 Previous work

Low-degree implicit surfaces are extensively used in the existing solid modeling and graphics systems as modeling primitives [RV83], and geometric operations on low-degree implicit surfaces are well understood [OSS]. Implicit surfaces are also very useful in surface fitting [PK89] and blending [RO87, MS85, HH87, Bli82].

Many authors have addressed the shape control of implicit patches [Sed85, WMW86, BW90]. In particular, Bloomenthal and Wyvill [BW90] discussed shape control using skeletons, and Sederberg pointed out that the Bernstein-Bezier representation are suitable for controlling the shapes of implicit patches [Sed85]

1.2 Overview

This paper is organized as follows. After giving some background information in Section 2, we describe the basic shape control techniques in Section 3. Section 4 shows how to achieve the convexity of quadric patches and cubic patches

2 Bernstein-Bezier representation

Given a tetrahedron V with vertices x_1, x_2, x_3, and x_4, one can express any point p in space as

$$p = \sum_{i=1}^{4} \tau_i x_i,$$

where

$$\sum_{i=1}^{4} \tau_i = 1.$$

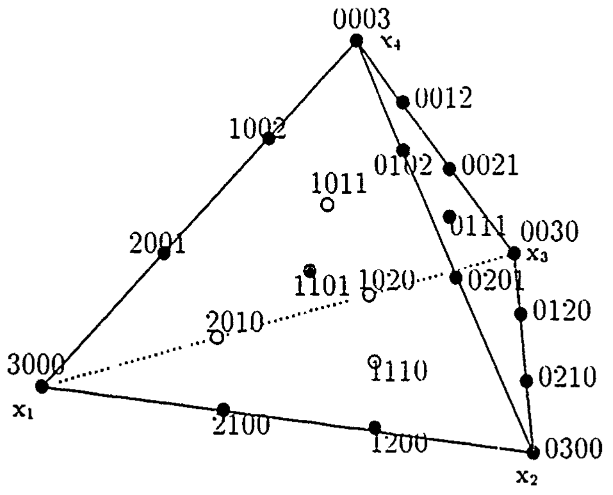


Figure 1: Cubic control points

The tuple $(\tau_1, \tau_2, \tau_3, \tau_4)$ is called the barycentric coordinate of $\mathbf{p}$. The barycentric coordinates are linearly related to Cartesian coordinates, so any implicit polynomial surface may be expressed in barycentric coordinates via a linear change of variables.

For a non-negative integer tuple $\lambda = (\lambda_1, \lambda_2, \lambda_3, \lambda_4)$ with $|\lambda| = \sum_{i=1}^{4} \lambda_i = n$, the Bernstein polynomial for λ is

$$B_\lambda^n(\tau) = \frac{n!}{\lambda_1!\lambda_2!\lambda_3!\lambda_4!}\tau_1^{\lambda_1}\tau_2^{\lambda_2}\tau_3^{\lambda_3}\tau_4^{\lambda_4}$$

Using Bernstein polynomials, one can uniquely represent any polynomial f of degree $\leq n$ as follows.

$$f(\tau) = \sum_{|\lambda|=n} b_\lambda B_\lambda^n(\tau)$$

The b_λ's are referred to as the *control points* of the polynomial f and its surface $S(f) = \{\mathbf{x} | f(\mathbf{x}) = 0\}$. The control points of a cubic polynomial are shown in Figure 1.

The following lemma is very useful.

Lemma 1 *If*

$$f(x) = \sum_{|\lambda|=k} c_\lambda B_\lambda^k(\tau)$$

and $c_{ke^i} = 0$, *then*

$$c_{(k-1)e^i+e^j} = (\nabla f(\mathbf{x}_i), \mathbf{x}_j - \mathbf{x}_i),$$

for $j = 1, 2, 3, 4$.

Proof: From [Dah86] ,

$$(\mathbf{x}_i - \mathbf{x}_j, \nabla f(\mathbf{x}))$$

$$= k \sum_{|\lambda|=k-1} (c_{\lambda+e^i} - c_{\lambda+e^j}) B_\lambda^{k-1}(\tau)$$

Letting $\mathbf{x} = \mathbf{x}_i$, we prove the lemma. ♣

3 The basic techniques for shape control

An implicit patch is defined as the zero contour of a polynomial f inside a tetrahedron $[\mathbf{x}_1\mathbf{x}_2\mathbf{x}_3\mathbf{x}_4]$[1].

$$f(\tau) = \sum_{|\lambda|=k} b_\lambda B_\lambda^k(\tau).$$

where τ is the barycentric coordinate defined by the tetrahedron $[\mathbf{x}_1\mathbf{x}_2\mathbf{x}_3\mathbf{x}_4]$. The basic idea of shape control is to express the geometric properties the implicit patch in terms of the control points so that one can achieve shape objectives by requiring the control points to satisfy certain constraints.

To get a feel for the effects of the control points, we study a univariate cubic polynomial f.

$$f(u) = b_{30}B_{30}^3 + b_{21}B_{21}^3 + b_{12}B_{12}^3 + b_{03}B_{03}^3.$$

The value of the function f over $[0,1]$ and the convex hull of the points $(0, b_{30})$, $(1/3, b_{21})$, $(2/3, b_{12})$, and $(1, b_{03})$ are shown in Figure 2. The functions B_{30}^3, B_{21}^3, B_{12}^3, and B_{03}^3 are shown in Figure 3.

From these figures, we can see the following.

1. At the end points of the interval $[0,1]$, the control points b_{30} and b_{03} equal to the function values of f.
2. The gradient of f at the end points of the interval $[0,1]$ are determined by b_{30}, b_{21}, b_{12}, and b_{03}.
3. The control point b_{30} has a effect on the value of $f(u)$ for all u except $u = 0$, and the effect is the strongest near $u = 1$. Similar statement can be made about other control points.

All these relations between the control points and the properties of the polynomial f generalize to trivariate polynomials

[1] We denote by $[\mathbf{x}_1, \ldots, \mathbf{x}_k]$ the convex hull of $\{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$

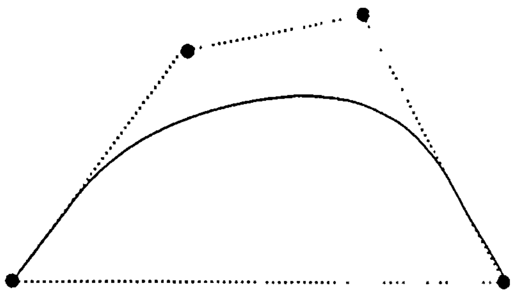


Figure 2: Function values of a univariate cubic function

Having understood the effect of the control points, we use the control points to control the shape of the surfaces. Consider the problem of interpolating points and lines in space by a surface $S(f)$. Since the values of f at a vertex of the tetrahedron $[\mathbf{x}_1\mathbf{x}_2\mathbf{x}_3\mathbf{x}_4]$ is equal to the value the control point at the vertex, setting the control point to zero forces the $S(f)$ to pass through the vertex. This method of interpolating a points can be generalized to a method of interpolating the edges of the tetrahedron $[\mathbf{x}_1\mathbf{x}_2\mathbf{x}_3\mathbf{x}_4]$.

Moving on to the problem of controlling the tangent plane of $S(f)$, we consider the tangent plane of $S(f)$ at the vertex $\mathbf{x}_1$. The tangent plane at $\mathbf{x}_1$ is defined by its the gradient $\nabla f(\mathbf{x}_1)$. From Section 2, we know that

$$b_{(k-1)e^1+e^j} = \frac{1}{k}(\nabla f(\mathbf{x}_1), \mathbf{x}_j - \mathbf{x}_1), \quad j = 2,3,4.$$

Since the vectors $\mathbf{x}_j - \mathbf{x}_1$ $(j = 2,3,4)$ are three linearly independent vectors, the above relation implies that the control points next to $\mathbf{x}_1$ completely determine the gradient $\nabla f(\mathbf{x}_1)$.

More sophisticated examples of shape control are easy to come by. The restriction of f to an edge of the tetrahedron $[\mathbf{x}_1\mathbf{x}_2\mathbf{x}_3\mathbf{x}_4]$ is a univariate polynomial. If the control points on the edge are all positive or all negative, then the surface $S(f)$ does not intersect the edge. Otherwise, the surface $S(f)$ intersects the edge exact once if there is exactly one sign change in the list of control points along the edge. Similar statements can be made for the faces of $[\mathbf{x}_1\mathbf{x}_2\mathbf{x}_3\mathbf{x}_4]$.

4 Achieving convexity

As an application of the techniques for shape control, we derive the convexity condition of an implicit patch in terms of its control points. Throughout this section, we concentrate on the implicit patch defined as the portion of a surface $S(f)$ inside a tetrahedron $V = [\mathbf{x}_1\mathbf{x}_2\mathbf{x}_3\mathbf{x}_4]$.

The reader is familiar with convex objects as a set of points in three dimensional space such that the line segment connecting two points in the set is contained in the set. Convex surfaces are often defined as surfaces whose Gaussian curvatures are positive over the entire surface. Convex objects and convex surfaces are related in that if a convex surface is closed and it bounds a point set with finite volume, then the point set is a convex object.

Defining a convex surface in terms of Gaussian curvature is not convenient when dealing with implicit surfaces. So we use the definition of convex surfaces in terms of the tangent planes. Let an implicit surface $S(f)$ have a tangent plane $S(P_{\mathbf{x}})$ at point $\mathbf{x} \in S(f)$. The surface $S(f)$ is convex at the point $\mathbf{x}$ if the surface $S(f)$ is in the half space bounded by $S(P_{\mathbf{x}})$ and pointed to by $-\nabla f(\mathbf{x})$. An implicit patch is convex if its primary surface is convex at every point on the implicit patch.

Notice the relationship between the convexity of the surface $S(f)$ and the convexity of the polynomial f. A polynomial f is convex over the tetrahedron V if for any two points $\mathbf{x}$ and $\mathbf{y}$ in the tetrahedron,

$$f(\frac{\mathbf{x}+\mathbf{y}}{2}) \leq \frac{1}{2}(f(\mathbf{x}) + f(\mathbf{y})).$$

It is easy to show that if the polynomial f is convex over the tetrahedron V, then the implicit patch defined as the portion of $S(f)$ inside V is convex. However, the converse is not true.

Motivated by the design of parametric convex surfaces, researchers in CAGD have obtained many results on the convexity conditions of polynomials over triangles [CP84]. It is possible to generalize these results to polynomials over tetrahedra [DM88], thus obtaining sufficient conditions for implicit patches to be convex. However, the convexity conditions obtained this way are often overly restrictive. So in the following, we derive the convexity conditions of an implicit patch directly from the definition of a convex implicit patch.

Let $\mathbf{p}' = (\tau_1', \tau_2', \tau_3', \tau_4')$ be a point close to a point $\mathbf{p} = (\tau_1, \tau_2, \tau_3, \tau_4)$ on the surface $S(f)$. The Taylor expansion of f, with higher order terms omitted, is

$$f(\mathbf{p}') = f(\mathbf{p}) + \sum_{i=1}^{4} \frac{\partial f}{\partial \tau_i}(\tau_i' - \tau_i) +$$

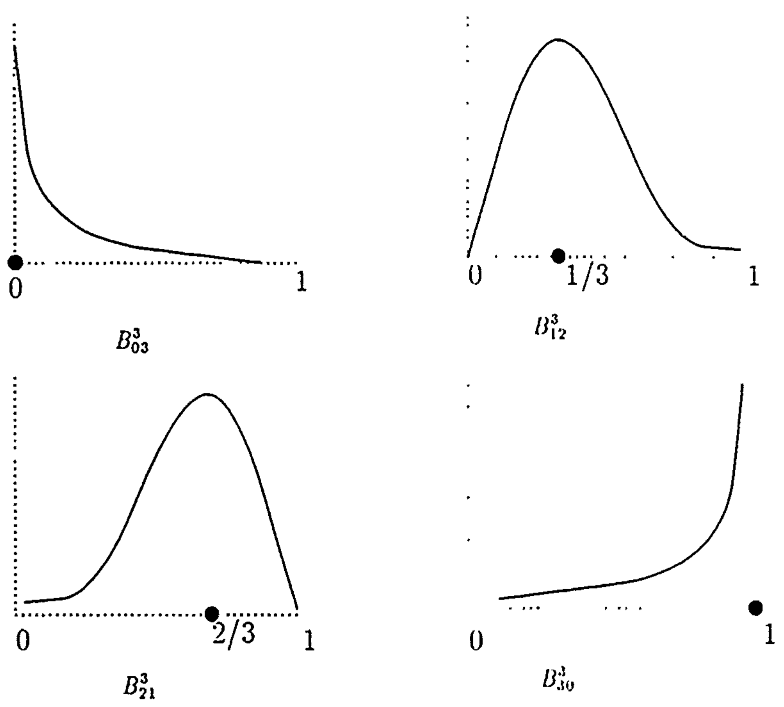


Figure 3: Weight functions for a univariate cubic function

$$\frac{1}{2}\sum_{i,j=1}^{4}\frac{\partial^2 f}{\partial\tau_i\partial\tau_j}(\tau_i'-\tau_i)(\tau_j'-\tau_j) \qquad (1)$$

The first term on the right-hand side of the above equation, $f(\mathbf{p})$, vanishes because $\mathbf{p}$ is on $S(f)$. Moving on to the second term, we notice that this term is the same as the left-hand side of the tangent plane equation at $\mathbf{p}$,

$$\sum_{i=1}^{4}\frac{\partial f}{\partial\tau_i}(\tau_i'-\tau_i)=0. \qquad (2)$$

So the definition of a convex implicit surface implies that $S(f)$ is convex at $\mathbf{p}$ if

$$\sum_{i,j=1}^{4}\frac{\partial^2 f}{\partial\tau_i\tau_j}(\tau_i'-\tau_i)(\tau_j'-\tau_j)\geq 0 \qquad (3)$$

for all $\mathbf{p}'$. Introducing new variable $\sigma_i=\tau_i'-\tau_i$, we can write (3) as

$$\sum_{i,j=1}^{4}\frac{\partial^2 f}{\partial\tau_i\partial\tau_j}\sigma_i\sigma_j\geq 0. \qquad (4)$$

The σ's satisfy the constraint

$$\sum_{i=1}^{4}\sigma_i=0 \qquad (5)$$

since the barycentric coordinates (τ_i) and (τ_i') satisfy the constraints

$$\sum_{i=1}^{4}\tau_i=1 \text{ and } \sum_{i=1}^{4}\tau_i'=1.$$

To eliminate the constraint (5), we substitute $-\sum_{i=1}^{3}\sigma_i$ for σ_4 in (4). The result is

$$\sum_{i,j=1}^{3}a_{ij}(\mathbf{p})\sigma_i\sigma_j\geq 0 \qquad (6)$$

for arbitrary $(\sigma_1,\sigma_2,\sigma_3)$ with

$$a_{ij}(\mathbf{p})=\frac{\partial^2 f}{\partial\tau_i\partial\tau_j}+\frac{\partial^2 f}{\partial\tau_4\partial\tau_4}-\frac{\partial^2 f}{\partial\tau_i\partial\tau_4}-\frac{\partial^2 f}{\partial\tau_j\partial\tau_4}. \qquad (7)$$

The condition (6) is the condition for the surface $S(f)$ to be convex at the point $\mathbf{p}$.

Applying the condition (6) to every point on an implicit patch, we have the following theorem.

Theorem 1 *An implicit patch is convex if the 3×3 matrix $A=(a_{ij})$ is positive definite for all points* $\mathbf{p}$ *on the implicit patch.*

Proof: Obvious from the above arguments. ♣

Generalizing the convexity conditions for bivariate polynomials would give a sufficient condition requiring the matrix A to be positive definite over the entire tetrahedron as opposed to the implicit patch. The condition in Theorem 1 is much less restrictive.

Although Theorem 1 gives a condition for the convexity of an implicit patch, the condition is hard to use because checking the condition for the infinitely many points on the implicit patch is impossible. So in the rest of the section, we use Theorem 1 to derive the convexity condition of an implicit patch in terms of its control points.

If f is a degree k polynomial given by

$$f = \sum_{|\lambda|=k} b_\lambda B^k_\lambda(\tau),$$

then

$$\frac{\partial^2 f}{\partial \tau_i \partial \tau_j} = k(k-1) \sum_{|\mu|=k-2} b_{\mu+e^i+e^j} B^{k-2}_\mu(\tau),$$

and $A = (a_{ij}(\mathrm{p}))$ is a 3×3 symmetric matrix whose entries are homogeneous degree $k-2$ polynomials in (τ_i). From linear algebra, A is positive definite if and only if

$$a_{11} \geq 0, \quad \begin{vmatrix} a_{11} a_{12} \\ a_{12} a_{22} \end{vmatrix} \geq 0, \text{ and } |A| \geq 0. \tag{8}$$

In order to decide whether A is positive definite for all points p on an implicit patch, we have to determine the signs of the minimum values of the quantities listed in (8) under the following the constraints,

$$f(\mathrm{p}) = 0, \tag{9}$$

$$\tau_1 + \tau_2 + \tau_3 + \tau_4 = 1, \text{ and } \tau_i \geq 0 \ (i = 1,2,3,4). \tag{10}$$

Here the constraints characterize the points on the implicit patch. The inequality constraints and nonlinear constraints in (9) and (10) make the problem of deciding the convexity of a general implicit patch very hard.

Fortunately, practical criterions for the convexity of quadric patches and cubic patches can be derived. For quadric patches, notice that

$$\frac{\partial^2 f}{\partial \tau_i \partial \tau_j} = b_{e^i+e^j}$$

is independent of p, so the convexity of a quadric patch can be decided by evaluating (8) with the constant $a_{ij} = b_{e^i+e^j} + b_{0002} - b_{e^i+e^4} - b_{e^j+e^4}$.

Deciding the convexity of a cubic patch is a little bit harder. Using the formula for Bernstein-Bezier polynomials, it is easy to verify that

$$\frac{\partial^2 f}{\partial \tau_i \partial \tau_j} = 6 \sum_{m=1}^{4} b_{e^m+e^i+e^j} \tau_m$$

Using this relation, we can re· rite (6) as

$$\sum_{m=1}^{4} \tau_m Q_m(\sigma) \geq 0. \tag{11}$$

where

$$Q_m(\sigma) = \sum_{i,j=1}^{4} (b_{e^i+e^j+e^m} +$$
$$b_{2e^4+e^m} - b_{e^4+e^i+e^m} - b_{e^4+e^j+e^m})\sigma_i\sigma_j.$$

Since the left-hand side of inequality (11) is a convex combination of $Q_m(\sigma)$, the inequality (11) is valid over the entire tetrahedron enclosing the cubic patch if and only if the inequality is valid at the vertices of the tetrahedron, i.e.

$$Q_m(\sigma) \geq 0, \text{ for } m = 1,2,3,4.$$

So the cubic patch is convex if the inequalities in (8) holds for $m = 1,2,3,4$ with constant

$$a_{ij} = b_{e^i+e^j+e^m} + b_{2e^4+e^m} - b_{e^4+e^i+e^m} - b_{e^4+e^j+e^m}.$$

An important observation is that for each m, the above condition is exactly the same as the convexity condition for a quadric patch. Using the terminology of CAGD, we can say that a cubic patch inside a tetrahedron is convex if the subpolynomials at the vertices of the tetrahedron are convex.

Acknowledgement: I would like to thank Prof. J. Hopcroft for providing the support, guidance, and encouragement for this work.

References

[Baj88] C. Bajaj. Geometric modeling with algebraic surfaces. Technical Report CSD-TR-825, Com. Sci., Purdue University, 1988.

[Bli82] J. Blinn. A generalization of algebraic surface drawing. *ACM Transaction on Graphics*, 1:235-256, 1982.

[Bli84] J. Blinn. The algebraic properties of homogeneous second order surfaces. In *Siggraph84*, 84.

[BW90] J. Bloomenthal and B. Wyvill. Interactive techniques for implicit modeling. *Computer Graphics*, pages 109-116, 1990.

[CP84] G. Chang and P. Davis. The convexity of Bernstein polynomials over triangles. *Journal of Approximation Theory*, 40:11-28, 1984.

[DM88] W. Dahmen and C. Micchelli. Convexity and Bernstein polynomials over k-simploids. Technical Report RC13969, T. J. Watson Research Center, IBM, 1988.

[Dah86] W. Dahmen. Bernstein-Bezier representation of polynomial surfaces. *Extension of B-spline curve algorithms to surfaces*, SIGGRAPH'86 Course Notes, 1986.

[Guo90] B. Guo. Free-form surface modeling using implicit patches. *Proceedings of the Second Canadian Conference on Computational Geometry*, Ottawa, ON, 1990.

[Guo91] B. Guo. Surface generation using implicit cubics. to appear in *Proceedings of Computer Graphics International '91*, Cambridge, Massachusetts, June, 1991, Springer-Verlag.

[HH87] C. Hoffmann and J. Hopcroft. The potential method for blending surfaces and corners. In *Geometric Modeling: Algorithms and New Trends*, pages 347–366, 1987.

[HK86] J Hopcroft and D. Krafft. The challenge of robotics for computer science. In C. Yap and J. Schwartz, editors, *Advances in Robotics, Vol. 1: Algorithmic and Geometric Aspects of Robotics*, 1986.

[MS85] A. Middleditch and K. Sears. Blend surfaces for set volume modeling systems. *SIGGRAPH Computer Graphics*, 19:161–170, 1985.

[OSS] S. Ocken, J. Schwartz, and M. Sharir. Piecewise implementation of CAD primitives using rational parameterizations of standard surfaces. In *Planning, Geometry, and Complexity of Robot Motion*, pages 245–266. Norwood, NJ, 1983.

[PK89] N. Patrikalakis and G. Kriezis. Representation of piecewise continuous algebraic surfaces in terms of B-splines. *The Visual Computer*, 5:360–374, 1989.

[RO87] A. Rockwood and J. Owen. Blending surfaces in solid geometric modeling. In G. Farin, editor, *Geometric Modeling: Algorithms and New Trends*, 1987.

[RV83] A. Requicha and H. Voelcker. Solid modeling: current status and research directions. *IEEE Journal on Computer Graphics and Its Applications*, 3:25-37,1983.

[Sed85] T.W. Sederberg. Piecewise algebraic surface patches. *CAGD*, 2:53–59, 1985.

[WMW86] G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for soft objects. *The Visual Computer*, 2, 1986.

Symbolic/Numeric Techniques in Modeling and Simulation*

Richard Zippel

TR 91-1214
June 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This paper will appear in the Proceedings of a DARPA workshop on manufacturing technology.

SYMBOLIC/NUMERIC TECHNIQUES IN MODELING AND SIMULATION

RICHARD ZIPPEL
Department of Computer Science, Cornell University
Ithaca, NY 14853, USA

ABSTRACT

Modeling and simulating collections of physical objects which are subject to a wide variety of physical forces and interactions is exceedingly difficult. The construction of a single simulator capable of dealing with all possible physical processes is completely impractical and, it seems to us, wrong-headed. Instead, we propose to build custom simulators for single, particular collections of physical objects and where pre-specified physical phenomena are involved. For such an approach to be practical, an environment needs to be provided that facilitates the quick construction of these simulators. In this paper we describe the essential features of such an environment and describe in some detail how a general implementation of the weighted residual method, one of the more general classes of of numerical integration techniques, can be used.

Keywords: Simulation, physical modeling, computational fluid dynamics, symbolic computation, weighted residual method, software development tools.

1 Introduction

We are intereste¹ in building software systems that simulate reality—especially when several different physical phenomena are involved in the simulation. Depending on the nature of the objects in a scene, their behavior may be governed by rigid body dynamics, fluid flow, quantum mechanics or other families of laws. The forces that act on these objects are gravitation and electromagnetism for macroscopic systems, and weak and strong interactions for systems at atomic scales. In addition, many observable properties of physical systems, including superconductivity, semiconductors and chemistry, are manifestations of statistical averages of detailed lower level behavior. These macroscopic phenomena are usually simulated through their own models—it being prohibitively expensive to simulate from first principles.

Besides the computational costs, the complexity of dealing with all physical phenomena and mechanisms would make such a simulator ferociously difficult to build. Rather than build such a general purpose simulator we propose a new program: Build special purpose simulators tuned for a particular configuration of physical objects and where a particular set of physical phenomena are involved. Such a simulator should be less complex than a general purpose simulator, which must be prepared for any eventuality. The specialized simulator will only have to consider a known system of equations with known parameters. It will consist of more straight-line code and will have fewer parameters and thus should be easier to tune

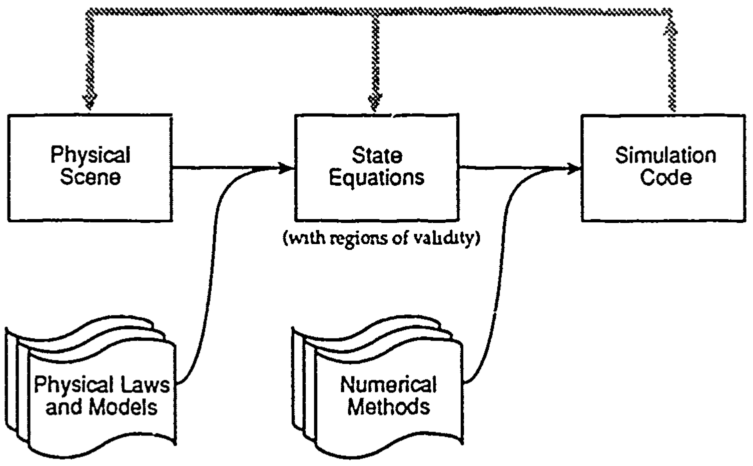


Figure 1: Simulation Architecture

for high performance/parallel computer architectures. However, each new problem configuration would require the creation of a new simulator.

To make this endeavor practical, we are combining a wide array of techniques from artificial intelligence, computer algebra, compiler technology and code transforms to provide an environment that vastly simplifies the process of building special purpose simulators. In effect, one builds a "simulator generator" that crafts a custom simulator for a particular configuration of physical objects. Such a simulator generator will generate the particular set of differential equations that model the scene and then will generate a piece of code for the explicit equations that apply to the problem. This approach has a number of advantages:

- More sophisticated mathematical techniques can be used to generate the systems of equations to be solved.
  - Conformal mapping techniques can be applied to the non-linear differential equations to simplify and regularize boundary conditions.
  - Averaging and perturbation techniques can be applied to reduce the order of the equations.
- Numerical techniques specialized to the equations being solved can be used.
- Software can be retargeted to different computer architectures relatively easily.

This new approach to simulation and modeling is replete with new problems that need to be studied and new technologies that need to be developed and applied. In this paper we sketch a general framework for simulator generation and consider some of the components in detail. It should be noted that we are sketching a simulation and analysis framework that is to be used not only for Newtonian mechanics but also for problems that are driven by electrodynamics, relativistic mechanics and/or quantum mechanics as well as aggregate models like solid state theory, galactic dynamics, chemical kinetics and fluid dynamics. Thus one should exercise caution when extrapolating from experience in just one simulation domain.

The process of performing a simulation is shown in Figure 1. We begin with the *observable scene* to be simulated. An observable scene is those properties of the system that ba be observed and are independent of the physics used to model the behavior of the scene. By applying the laws of physics to the scene *state equations* are generated whose solution describes the evolution of the scene with time, within certain regions of validity. The state equations are then converted into code that numerically computes the scene's state changes. As time advances, the state equations may cease to be valid and must be changed. Similarly, the geometric or topological characteristics of the scene itself may change. These effects are indicated by the shaded "feedback" arrows in Figure 1.

The state of a physical system is determined by the values of a set of *state variables*, which may include a subset of the observable parameters of the objects in a scene. The result of applying the physical laws to a scene are a set of *state equations* that constrain the state variables over time. For instance, for clocked boolean logic circuits, there is a finite set of state variables, each of which ranges over {`true, false`}, time is modeled by a sequence of discrete events occurring on clock edges and the state equations are boolean equations. For rigid body dynamics, there is a discrete set of state variables that have continuous values, time is modeled as a sequence of irregularly spaced events and the state equations are ordinary differential equations. In fluid dynamics, there is a continuous number of of state variables, one for each point in the (continuous) fluid, and their values range over a continuous vector space. The state equations are partial differential equations.

Once the state equations of a scene have been generated (the middle box of Figure 1), general mathematical techniques can be used to convert them into a form where numerical information about their state variables can be determined. Examples include conversions of ordinary differential equations into finite difference formulas by Runge-Kutta methods, or the conversion of partial differential equations into systems of linear equations by finite element methods. We call the process of converting a system of equations into an effective computational form a *discretization.*

These computation structures can then be converted to actual programs (or codes) that numerically simulate the scene. If something is known about the architecture of the computer that will run the program then especially fast codes can be generated by symbolic elimination of variables, unrolling of loops or duplication of code. Each of these options may be appropriate because of cache sizes, vector processing structures or interprocessor communication costs. Other techniques of compiler theory are also appropriate and should be carefully considered at this point. More radical transformations like changing the order of the discretization or the discretization method itself may also be appropriate. This entire process of converting state equations into computational structures and then into executable code is indicated in the right half of Figure 1.

This paper discusses each of these steps in the simulation process. In Section 2 we discuss one approach to representing scenes, their components and the underlying physics. Once the state equations have been generated, they can be directly solved numerically, yielding the trajectory of the scene from a given set of initial conditions. The approach we are pursuing is discussed in some detail in Section 4.

However, occasionally some property of the trajectory is of interest—not the trajectory itself. We argue that by using symbolic techniques, we can often transform the differential equations that describe the system into other equations whose solutions more precisely answer the questions being asked. Solving these transformed equations is often substantially easier than solving the original system. However, substantial (non-numeric computation) is required to produce the transformed equations. In Section 3 we illustrate how averaging techniques can be applied to reduce the dimension of the problem being solved and more directly answer the questions of interest. This technique is classical, but we fell is representative of the type of reduction that will be valuable in the future and is possible by the general framework being proposed.

In the domain in which we are working (fluid dynamics), the state equations are partial differential equations. A wide variety of different methods are available for their numerical solution. Many of these methods can be subsumed within the general mathematical framework of *weighted residual methods*. Because we have access to the state equations in symbolic form, we can directly apply the weighted residual methodology to the differential equations of the problem to produce a computational structure based on a wide variety of different techniques including finite element, spectral and collocation methods. This approach is discussed in Section 4. In Section 4.1 we describe the general principles behind the weighted residual method. In Section 4.2 we use the weighted residual method to produce a spectral method computational structure for a problem in fluid dynamics. This particular example illustrates the complexity of the codes generated in the study of turbulent fluid dynamics.

In Section 4.3 we give another illustration of the weighted residual method in fluid dynamics, but this time the result of the discretization process is not a system of linear equations, but rather a system of ordinary differential equations. This is another example of where symbolic techniques can be used to convert a numerical problem into one that more directly provides the desired answers.

2 Scenes and Laws of Physics

This section makes more precise what we mean by scenes and physical laws. Section 2.1 discusses scenes while Section 2.2 discusses the components of a physics and some their functions.

2.1 Scenes

When describing a physical system that is to be simulated, we distinguish the observable properties and characteristics of the system from those properties and characteristics that are required by a particular physical model. The former are components of the *observable scene*, while the later belong to the physical laws and models that are to be applied to the scene. For instance, the charge, mass and position of an electron are components of the scene, but the electric field is a characteristic of a physical model that might be used to determine the effect of the electron on other charged particles. Fields in physics are not themselves directly observable. It is only through their effect on other objects that their existence can be ascertained. The effect of one object on another is the purpose of a physics and thus fields are artifices used to facilitate the physics itself. (Recall that general relativity replaces a gravitational field by bending of the fabric of space itself. For small masses these disparate mechanisms give the same predications.)

Similarly, the "ether" of nineteenth century physics belongs to a set of physical laws, and is not intrinsic to the scene. Ether is posited by nineteenth century physics and is not, itself, observable. A more modern example is the wave function of quantum mechanics. It cannot be observed in the scene but is essential for a particular set of physical laws. In all of these cases the physics used to analyze the system imposes additional parameters (*e.g.*, wave functions) or objects (*e.g.*, fields or ether) as an aid in specifying the physics itself.

A scene consists of a number of *objects* (rods, resistors, fluids, etc.) and *connections* (hinges, electrical nodes, etc.) between them. The constraints constrain the behavior of two or more objects in some fashion. For instance, a hinge between two rods requires that the rods remain connected, while an electrical node connecting the pins of two resistors ensures that the two pins always have the same potential.

In addition, the objects possess a number of "observable" properties, *e.g.*, the position and momentum of a particle. These properties are those aspects of the state of the object that may be observed in the scene, and thus are independent of the physics used to model the behavior of the scene. The observable properties may be redundant and related by some equations. For instance, the observable properties of a particle include the particle's mass (m), position ($\mathbf{r}$), velocity ($\mathbf{v}$), momentum ($\mathbf{p}$) and kinetic energy (T), where

$$\mathbf{v} = \frac{d\mathbf{r}}{dt},$$

$$\mathbf{p} = m\mathbf{v},$$

$$T = \frac{m\mathbf{v}\cdot\mathbf{v}}{2} = \frac{\mathbf{p}\cdot\mathbf{p}}{2m}.$$

For some models of physics, like Newtonian mechanics, the observable parameters actually correspond to state of the physics. That is, the observable position and momentum are actually the position and momentum of the object in the physics. In other models, *e.g.* quantum mechanics, the observables are derived from the

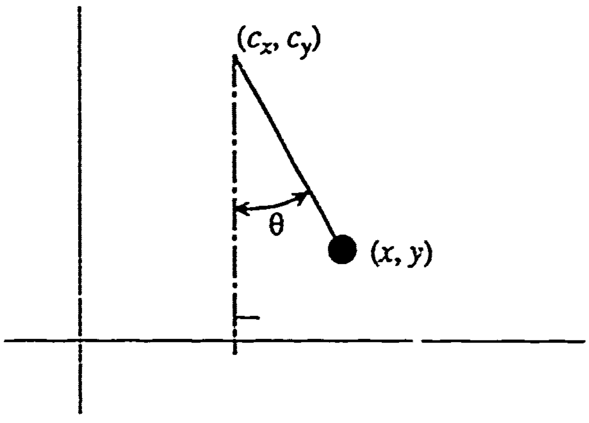


Figure 2: Simple Pendulum

their correspondents. That is the quantum mechanical position and momentum of a particle are not interchangeable with the observable position and momentum of the particle.

2.2 Physics

The properties of an observable scene are not necessarily appropriate for simulation. Instead, the physical laws translate the scene into one where the new scene's objects are described using state variables. For instance, a two dimensional scene that consists of a heavy bob at the end of a rigid, massless rod whose other end is hinged (*i.e.*, a two dimensional pendulum) might have constitutive parameters of the length of the rod (ℓ) and the mass of the bob (m)—see Figure 2. The observable parameters in the scene might be the position of the bob ($(x, y) \in \Re^2$). However, when formulating a simulation, one would probably use the deflection of the rod from vertical ($\theta \in [-\pi, \pi)$) as the state variable of the system. The position of the bob can be derived from θ by

$$(x, y) = (c_x + \ell \sin\theta, c_y + \ell \cos\theta).$$

Each set of physical laws acts similarly. It must construct from an observable scene an *interpretation scene* that consists of objects, state variables that are appropriate to the physical model and the manifold structure in which the state variables lie. A correspondence also needs to be provided between state variables in the interpretation scene and quantities in the observable scene. The combination of the state variables, their manifold, and the correspondence we call an *interpretation scene* or just an *interpretation*. Examples of interpretations are the generalized coordinates of Hamiltonian mechanics (which were used in the pendulum example) and the wave functions of quantum mechanics.

A description of a physics consists of (1) the domains of validity of the physics, (2) a method to generate an interpretation scene from an observable scene and (3) how fields and energies are to be derived from the resulting interpretation scene. In addition we must have a *formulation of mechanics* that allows us to combine the fields and energies produced by the different sets of physics to generate the state equations. Examples of formulations are Lagrangian and Hamiltonian mechanics, both of which can be applied outside the domain of Newtonian mechanics.

The physical laws that apply to a scene are kept separate from the scene and should be expressed independently of their application to a particular scene. There should be one (or more) descriptions of rigid body dynamics and one (or more) descriptions of electrodynamics. These descriptions include the "laws of physics" (*e.g.*, $\mathbf{F} = m\mathbf{a}$ for rigid body dynamics, or Maxwell's equations), specifications of when the particular laws are applicable and procedural specifications of how to apply the laws to a particular scene.

We call a set of physical laws a *physics*. Each physics has a limited range of applicability (until the Grand Unified Theory is discovered). Among the components of a physics is a specification of how forces and energies of objects in a scene can be computed. There are multiple physics's, some of which are compatible with each other over certain ranges of state variables and some of which are incompatible. For instance, Newtonian mechanics and classical electrodynamics are compatible for small masses and slowly moving macroscopic particles. Electrodynamics merely introduces a new force, which is characterized by Coulomb's law. Quantum mechanics and general relativity seem incompatible.

This approach ensures that different physical considerations are dealt with separately. For instance, one should be able to simulate an electric motor by applying both rigid body dynamics and electromagnetics to a scene that consists of the rotor and stator of the motor, with the appropriate constitutive properties. We believe that greater modularity will result from this approach, although it places a premium on the symbolic techniques.

3 Harmonic Balance

When setting up a system of differential equations that models some physical situation, it is often easier to generate the equations in terms of state variables that are different from the ones that the user is really interested in. For instance, for a mechanical system it may be easiest to generate equations in terms of cartesian coordinates while the interesting behavior might best be expressed in terms of radial coordinates, or angular momenta or even averaged angular momenta. Each of these conversions can be performed after the numerical solutions are generated, however, using symbolic techniques to perform this conversion before the integration process makes generating an accurate solution easier.

To illustrate this approach we will use a more sophisticated type of coordinate change that also facilitates an averaging technique. Thus we will ultimately generate differential equ· ons for the average values of the variables of interest.

A large varie‹‚ of simple oscillatory type systems can be modeled by differential equations of the form

$$\begin{aligned}\dot{x} &= y,\\ \dot{y} &= -x + \epsilon h(x, y).\end{aligned} \tag{1}$$

For $h(x,y) = (1-x^2)y$ we have the van der Pol equation [14], for $h(x,y) = (1-y^2)y$ the Rayleigh equation [12], etc. When $\epsilon = 0$ (1) reduces to a simple harmonic oscillator, whose solution is:

$$x(t) = r_0 \cos(t + \phi_0), \qquad y(t) = \dot{x}(x) = -r_0 \sin(t + \phi_0), \tag{2}$$

where r_0 and ϕ_0 are constants set by the initial conditions. In the x-y plane (the phase plane), the solutions are circles centered at the origin. The term $\epsilon h(x,y)$ of (1) acts as a perturbing non-linear damping factor on the solution to the harmonic oscillator. An example of the behavior of this damping factor can be seen from the van der Pol equation where $h(x,y) = (1-x^2)y$:

$$\begin{aligned}\dot{x} &= y,\\ \dot{y} &= -x + \epsilon(1 - x^2)y.\end{aligned} \tag{3}$$

The phase plot of the van der Pol equation, for $\epsilon = 0.6$ and various initial conditions, is shown in Figure 3.

In the phase plane, (3) has a stable *limit cycle* of radius approximately 2. If the initia' conditions of the system are outside the limit cycle, the system will cycle inwards around the limit cycle continuously getting closer. If the starting point is inside the the limit cycle the system will oscillate outwards towards the limit cycle. From a physical point of view we might have two basic questions:

- What is the average amplitude of the limit cycle?
- How quickly does the system converge to the limit cycle?

We can study the behavior of the non-linear oscillator by ass‹ n·ing the solution is of the form (2) but allow the constants to be time varying functions, : .

$$\begin{aligned}x &= r(t)\cos(t + \phi(t)),\\ y &= r(t)\sin(t + \phi(t)).\end{aligned}$$

Substituting these expressions into (1) gives the following system of equations

$$\begin{aligned}&\dot{r}\cos(t+\phi) - \sin(t+\phi)(1+\dot{\phi}) = r\sin(t+\phi),\\ &\dot{r}\sin(t+\phi) + \cos(t+\phi)(1+\dot{\phi}) = \epsilon h(r\cos(t+\phi), r\sin(t+\phi)).\end{aligned}$$

When solved for $\dot{r}$ and $\dot{\phi}$, which must be done symbolically, we have

$$\begin{aligned}\dot{r} &= \epsilon h(r\cos(t+\phi), r\sin(t+\phi))\sin(t+\phi)\\ \dot{\phi} &= -\frac{\epsilon}{r} h(r\cos(t+\phi), r\sin(t+\phi))\cos(t+\phi).\end{aligned}$$

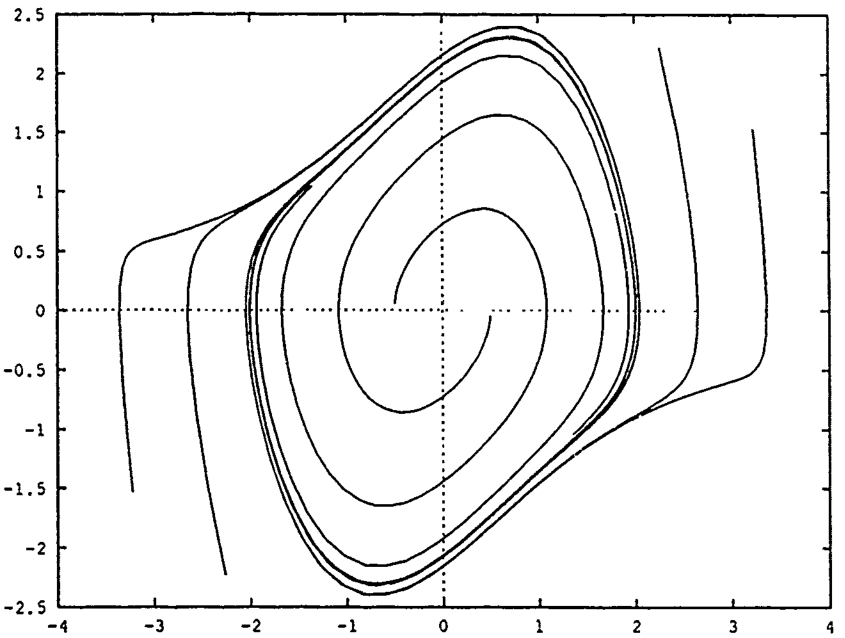


Figure 3: van der Pol Oscillator

The solution to this system of equations gives the amplitude of the system, which is closer to what we are looking for. In a physical system we probably don't care about the phase information. We are more interested in the asymptotic behavior of the system. This can be obtained by *averaging* these equations over one oscillation, *i.e.* t to $t + 2\pi$:

$$\frac{d\langle r\rangle}{dt} = \frac{\epsilon}{2\pi}\int_0^{2\pi} h(r\cos(t+\phi), r\sin(t+\phi))\sin(t+\phi)\,dt,$$

$$\frac{d\langle\phi\rangle}{dt} = -\frac{\epsilon}{2\pi r}\int_0^{2\pi} h(r\cos(t+\phi), r\sin(t+\phi))\cos(t+\phi)\,dt.$$

In the r-ϕ coordinate system, the van der Pol equation becomes

$$\begin{aligned}\dot{r} &= \epsilon r(1 - r^2\cos^2(t+\phi))\sin^2(t+\phi)\\ \dot{\phi} &= \epsilon(1 - r^2\cos^2(t+\phi))\sin(t+\phi)\cos(t+\phi)\end{aligned} \tag{4}$$

When, averaged, the equation for $\dot{r}$ becomes

$$\frac{d\langle r\rangle}{dt} = \frac{\epsilon}{2\pi}\int_0^{2\pi}(r^2\cos^2(t+\phi) - 1)r\sin^2(t+\phi)\,dt = -\epsilon\left(\frac{\langle r\rangle^3}{8} - \frac{\langle r\rangle}{2}\right). \tag{5}$$

The solution of this equation is precisely the evolution of the "average" amplitude of the oscillation without any additional information. Notice that we have been able

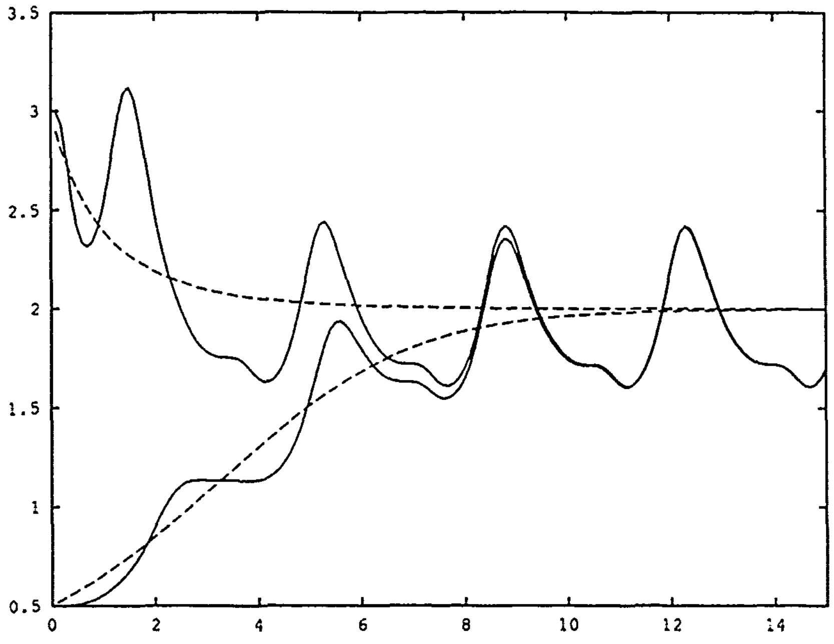


Figure 4: Amplitude of van der Pol Oscillator: raw and averaged

to reduce the order of the equation by one by averaging out the phase information. In Figure 4 we have shown the evolution of the a solution of (4) for two starting points, one inside and one outside the limit cycle, using a solid line. The dotted lines indicate solutions of the averaged equation (5) from the same two starting points.

On the stable limit cycle of the system, $\langle \dot{r} \rangle$ will vanish, so by solving

$$0 = -\epsilon \left(\frac{\langle r \rangle^3}{8} - \frac{\langle r \rangle}{2} \right)$$

we see that the average radius of the limit cycle is 2, which is independent of the initial conditions. This can also be observed from Figure 4.

The rate at which a solution approaches the limit cycle can be determined by solving (5):

$$r(t) = \left(\frac{4c_1 e^{\epsilon t}}{c_1 e^{\epsilon t} - 1} \right)^{1/2} .$$

This type of perturbation analysis has been used by in celestial mechanics since the time of Laplace and Lagrange. The particular problem we consider here, the behavior of solutions of equations of the form (1) was discussed in some detail by Poincaré [11]. More recently Krylov and Bogoliubov [8] have demonstrated the use of averaging techniques in a wide variety of problems. For a more modern treatment, one might look at [13].

4 Weighted Residual Methods

For a large number of physical simulation problems the state equations are partial differential equations. Though the number of techniques for solving these systems can be bewildering in their number, the most important techniques can be divided into two major classes: finite difference algorithms and weighted residual methods. We have decided to focus on weighted residual methods because most of the techniques of interest in our application area are of the weighted residual type.

There are a vast number of implementations of numerical algorithms based on particular weighted residual methods, most often for particular partial differential equations, but to our knowledge there have been no previous attempts to build a system that generates a numerical solver for a wide class of weighted residual methods.

We describe the basic principles behind the weighted residual method in Section 4.1. In Section 4.2 we give a brief illustration of how the weighted residual method is used to generate particular numerical codes in fluid mechanics. Finally, in Section 4.3 we use the weighted residual method, along with a number of other ideas, to reduce some questions about fluid flow to questions about a system of ordinary differential equations.

4.1 General Approach

Let

$$Lu = f \tag{6}$$

be a partial differential equation, where L is a partial differential operator and u is a function of $\{x_1, \ldots, x_m\}$. The weighted residual method assumes there exists a (possibly infinite) set of trial functions $\{\phi_i\}$ such that, for some choice a_i,

$$\hat{u} = \sum_{0 \leq i < N} a_i \phi_i \tag{7}$$

is an approximation to u, the solution of (6). The ϕ_i are function of some subset of $\{x_1, \ldots, x_m\}$ while the a_j are functions of the remaining variables. Substituting (7) into (6) we have residual error

$$R_E(\hat{u}) = L\left(\sum_{0 \leq i < N} a_i \phi_i\right) - f.$$

The goal of a weighted residual method is to choose the a_i in a fashion that minimizes $R_E(\hat{u})$ in some global sense.

A set of equations for the a_i can be deduced by choosing a set of weighting functions, w_j and requiring the inner product of $R_E(\hat{u})$ with the weights to vanish. *i.e.*

$$\int w_j R_E(\hat{u})\, dV = 0 \tag{8}$$

If the ϕ_i are functions of all of the variables $\{x_1, \ldots, x_m\}$, then the resulting equations are algebraic in the a_j. When L is a linear differential operator, the resulting equations are linear. Applying L to the components of the expansion and rewriting (8) we have

$$\sum_{0 \le i < N} a_i \int w_j\, L\phi_i\, dV = \int w_j f\, dV$$

For certain operators L and known ϕ_i and w_j the integrals above can be tabulated. Thus the bulk of the symbolic computation inherent in the reduction of (8) to systems of equations in the a_i can be performed *a priori*. However, if the ϕ_i and w_j are supplied by the user and, especially, if L is non-linear then symbolic computation is unavoidable in the application of the weighted residual method.

If the ϕ_i involve a subset of $\{x_1, \ldots, x_m\}$ then (8) will be a system of ordinary differential equations. Often the ϕ_i are functions only of the spatial variables and the a_j are functions of time. This is the situation in the two examples considered here. In Section 4.2 the partial differential equation is first discretized in time and then the weighted residual method is applied, producing a system of linear equations that need to be solved. In Section 4.3, the weighted residual method is applied directly to the spatial variables resulting in a system of ordinary differential equations for the a_i.

A wide variety of different integration schemes fall into this general framework. If the w_j are chosen to be the same as the ϕ_i we get a Galerkin projection. This is especially convenient if the ϕ_i are orthogonal and eigenfunctions of L. The system of linear equations are then diagonal. The resulting technique is called a spectral method. The most common spectral method chooses $\phi_k = e^{i\mathbf{k}\cdot\mathbf{x}}$.

The finite element method discretizes the computational domain Ω into a number of elements, $\Omega_1, \ldots, \Omega_N$. It then chooses the ϕ_i to be continuous functions that are zero everywhere except within Ω_i. A Galerkin projection then gives the equations for the a_i.

In general, determining the linear equations or ordinary differential equations that need to be solved from (8) is a rather painful process that must be performed by hand. By taking advantage of methods from symbolic computation we can largely automate this process.

4.2 Numerical Example

In this section we illustrate how the weighted residual method is used to produce a numerical code for a problem that arises in the study of turbulent channel flow. This example illustrates the complexities that arise in practical applications of the weighted residual method. A simplification of one of the equations that arose in Kim, Moin and Moser's study of turbulence in a channel flows [6] is

$$\frac{\partial g(x,y,t)}{\partial t} = h(g) + \frac{1}{\mathrm{Re}}\nabla^2 g, \tag{9}$$

where x and y are the spatial dimensions of the the problem (only two are needed for this illustration). h is a known non-linear function of g and other functions that occur in the problem. In practice it can be a fairly complex expression and more than one partial differential equation be involved.

In solving this problem, discretization must occur in three different dimensions—time and the two spatial dimensions. Three different schemes will be used: An implicit finite difference scheme for time (t), a spectral method for the x dimension and a Galerkin type method using Chebyshev polynomials for the y dimension.

These three schemes are used in three successive steps. First, time is discretized and the value of $g(x,y,t)$ at the n^{th} time step is denoted by $g^n(x,y) = g(x,y,n\,\Delta t)$. Second, $g^n(x,y)$ is discretized in the x dimension using Fourier expansion with coefficients $\tilde{g}_k^n(y)$. Finally, $\tilde{g}_k^n(y)$ is discretized using Chebyshev polynomials in the y dimension with coefficient $\tilde{g}_{k,j}^n$. That is,

$$\begin{aligned} g^n(x,y) &= \sum_{0\le k<N_x} \tilde{g}_k^n(y) e^{2\pi i k x/L_x}, \\ &= \sum_{0\le k<N_x} \sum_{0\le j<N_y} \tilde{g}_{k,j}^n T_j(y) e^{2\pi i k x/L_x}. \end{aligned}$$

At this point the coefficients are numbers, and if done properly they are solutions of linear equations. Once these linear equations have been solved we can reconstruct $g(x,y,t)$ by summing the series.

Each of these transformations can be automated using symbolic techniques. In practice, their application is not completely straightforward. The following paragraphs illustrate this with some comments on the implementation of these techniques using symbolic computation.

The first step is to perform the time-wise discretization. We denote by $g^n = g^n(x,y)$ the function that corresponds to g at the n^{th} time step. The most straightforward discretization would be the explicit formula

$$\frac{g^{n+1} - g^n}{\Delta t} = h(g^n) + \frac{1}{\mathrm{Re}}\nabla^2 g^n$$

But this is known to be relatively unstable.

Figure 5 gives a number of different discretization techniques that can be used for ordinary differential equations. For this particular problem no one of them is completely satisfactory. For instance, the explicit methods (Euler or Adams-Bashforth [2 4]) are not sufficiently stable when applied to the entire equation. The implicit methods require the solution of a non-linear equation at each time step (because of the nonlinearity of h) and are thus too costly.

$$\frac{x^{n+1}-x^n}{\Delta t} = \begin{cases} f(x^n) & \text{Explicit Euler} \\ f(x^{n+1}) & \text{Implicit Euler} \\ \dfrac{f(x^{n+1})+f(x^n)}{2} & \text{Crank-Nicolson} \\ \dfrac{1}{2}\left[3f(x^n)-f(x^{n-1})\right] & 2^{nd}\text{ Adams-Bashforth} \\ \dfrac{1}{12}\left[23f(x^n)-16f(x^{n-1})+5f(x^{n-2})\right] & 3^{rd}\text{ Adams-Bashforth} \end{cases}$$

Figure 5: Discretization techniques for $\dot{x}(t) = f(x)$

The solution is to use an explicit scheme on the nonlinear terms and an implicit scheme on the linear terms. Using the second order Adams-Bashforth formula for the linear terms and the Crank-Nicolson formula [3] for the linear terms yields

$$\frac{g^{n+1}-g^n}{\Delta t} = \frac{1}{2}\left(3h(g^n)-h(g^{n-1})\right) + \frac{1}{2\mathrm{Re}}\left(\nabla^2 g^{n+1} + \nabla^2 g^n\right).$$

In a symbolic manipulation system this process is quite simple. The differential equation is first converted to a sum of terms form. Each term is then examined to see if it is linear in g. If so, an implicit formula is applied to each term, otherwise an explicit one. The results of these replacements are then added together and simplified.

The terms that involve g^{n+1} can be isolated on the left hand side to give

$$g^{n+1} - \frac{\Delta t}{2\mathrm{Re}}\nabla^2 g^{n+1} = \frac{\Delta t}{2}\left(3h(g^n)-h(g^{n-1})\right) + g^n + \frac{\Delta t}{2\mathrm{Re}}\nabla^2 g^n. \tag{10}$$

Again the symbolic processing involved is straightforward, each term is examined and placed on one side of the equation or the other based on its dependence on g^{n+1}.

At a given time step, each of the terms on the right hand side of (10) is known and can be computed directly. The next step is to compute the Fourier transform of this equation, eliminating the functional dependence on x.

$$g^n(x,y) = \sum_{0\leq k<N_x} \bar{g}_k^n(y)e^{2\pi ikx/L_x}.$$

Thus the k^{th} mode the Fourier transform of the left hand side of (10) is

$$\mathcal{F}_k\left\{\left(1-\frac{\Delta t}{2\mathrm{Re}}\nabla^2\right)g^{n+1}\right\} = \bar{g}_k^{n+1} + \frac{\Delta t}{2\mathrm{Re}}\left(4\pi^2\bar{g}_k^{n+1}\left(\frac{k}{L_x}\right)^2 - \frac{\partial^2\bar{g}_k^{n+1}}{\partial y^2}\right)$$

So for each k we need to solve the equation:

$$\tilde{g}_k^{n+1} + \frac{\Delta t}{2\mathrm{Re}}\left(4\pi^2\left(\frac{k}{L_x}\right)^2 - \frac{\partial^2}{\partial y^2}\right)\tilde{g}_k^{n+1}$$
$$= \mathcal{F}_k\left\{\frac{\Delta t}{2}\left(3h(g^n) - h(g^{n-1})\right) + g^n + \frac{\Delta t}{2\mathrm{Re}}\nabla^2 g^n\right\}. \tag{11}$$

This equation is finally discretized in y by expanding $\tilde{g}_k^n(y)$ in terms of Chebyshev polynomials:

$$\tilde{g}_k^n(y) = \sum_{0\le j<N_y} \tilde{g}_{k,j}^n T_j(y),$$

where $\tilde{g}_{k,j}^n$ are numbers. The Chebyshev expansion of the left hand side of (11) is

$$\tilde{g}_k^{n+1} + \frac{\Delta t}{2\mathrm{Re}}\left(4\pi^2\left(\frac{k}{L_x}\right)^2 - \frac{\partial^2}{\partial y^2}\right)\tilde{g}_k^{n+1}$$
$$= \sum_{0\le j<N_y} \tilde{g}_{k,j}^{n+1}T_j(y) + \frac{\Delta t}{2\mathrm{Re}}\left(4\pi^2\left(\frac{k}{L_x}\right)^2 T_j(y)\tilde{g}_{k,j}^{n+1} + \frac{d^2T_j(y)}{dy^2}\tilde{g}_{k,j}^{n+1}\right)$$
$$= \sum_{0\le j<N_y}\left[\left(1 + \frac{2\pi^2\Delta t}{\mathrm{Re}}\left(\frac{k}{L_x}\right)^2\right)\tilde{g}_{k,j}^{n+1}\right]T_j(y) - \frac{d^2T_j(y)}{dy^2}\tilde{g}_{k,j}^{n+1}$$

The last term in this sum causes some problems because it is not expressed as a sum of Chebyshev polynomials, but as a sum of their derivatives. However, derivatives of Chebyshev polynomials can be expressed as a sum of Chebyshev polynomials by repeated application of the formula

$$\frac{T''_{n+2}(x)}{(n+1)(n+2)} - \frac{T''_n(x)}{(n^2-1)} + \frac{T''_{n-2}(x)}{(n-1)(n-2)} = 4T_n(x),$$

or by solving the tridiagonal system it implies. At this point, we have converted the problem of advancing time in (9) to solving systems of linear equations and computing Fourier and Chebyshev transforms.

For other basis and weight functions, and for other differential equations, very similar approaches are used. Simple symbolic methods (arithmetic operations and some simplification) are used to reduce the projection process to a sequence of integral. In the case discussed here, all of the integrals could be performed by table lookup. In the next section the integrals will have to be performed numerically.

4.3 Proper Orthogonal Decomposition

By discretizing the spatial dimensions but not the time dimension, we can reduce the Navier-Stokes equations to a system of ordinary differential equations. If the proper basis functions are chosen and sufficient terms are used the dynamical behavior of the ODE's should closely approximate that of the Navier-Stokes equations.

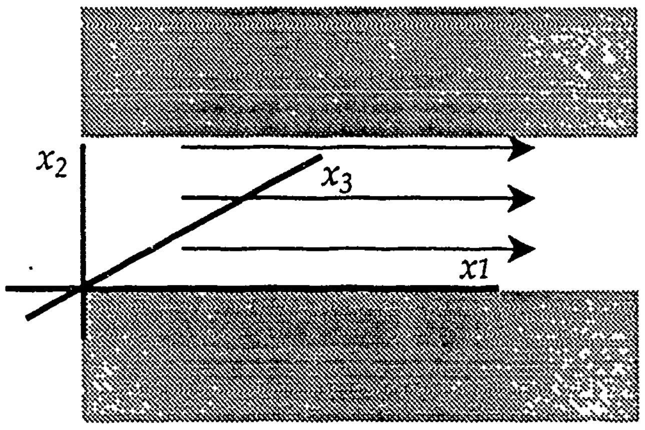


Figure 6: Coordinate System for a Channel

Lumely [9] has suggested using this approach to study the behavior of the turbulent boundary layer of a fluid moving over a flat plate. Within the boundary layer bursts can be observed that are spatially and temporally somewhat periodic [7]. It would be interesting to know if these periodic phenomena manifests themselves in the ordinary differential equations where more powerful mathematical techniques can be used to analyze their behavior.

This reduction and detailed study of the resulting dynamical systems was originally performed by John Lumley, Philip Holmes and their students [1]. As we shall see the ordinary differential equations that result are extremely complex and are best generated by symbolic techniques.

Fluid flow is governed by the Navier-Stokes equations. In the absence of external forces the dimensionless form of these equations is

$$\frac{\partial \mathrm{v}}{\partial t} + (\mathrm{v} \cdot \nabla)\mathrm{v} = -\nabla\pi + \frac{1}{\mathrm{Re}}\nabla^2\mathrm{v}, \tag{12}$$

$$\nabla \cdot \mathrm{v} = 0 \tag{13}$$

where v denotes the velocity field of the fluid and Re is the Reynolds number of the fluid. Flows with small Reynolds numbers tend to be rather steady, while flows with Reynolds numbers greater than 2300 are generally turbulent. In order to write the equations in a dimensionless form, characteristic lengths need to be defined in each of the three dimensions. We denote these different characteristic lengths by L_1, L_2 and L_3.

If $f(x_1, x_2, x_3)$ is function of position in the channel, we will denote by $\langle f \rangle$ its spatial average in a plane parallel to the walls of the channel:

$$\langle f(x_1, x_2, x_3) \rangle = \frac{1}{L_1 L_3} \int f(x_1, x_2, x_3)\, dx_1 dx_3,$$

where L_1 and L_3 are characteristic dimensions in the x_1 and x_3 directions respectively. Within this plane the turbulent flow is relatively homogeneous. Variations occur in the orthogonal direction. Thus $\langle f(x_1, x_2, x_3)\rangle$ is a function of only the distance from the wall, x_2.

The streak structure that we are interested in is not a function of the mean velocity of the fluid, only its fluctuations. Denoting

$$\langle \mathrm{v}\rangle = (U(x_2), 0, 0) = \mathrm{U},$$

we can determine $U(x_2)$ exactly:

$$U(x_2) = \mathrm{Re}\int_0^{x_2} \langle u_1 u_2\rangle \, dx_2' + \mathrm{Re} u_T^2 \left(x_2 - \frac{x_2^2}{H}\right), \tag{14}$$

where u_T is the dimensionless wall shear velocity and H is the half height of the channel. Applying this to (12) we get the *Reynolds averaged* Navier-Stokes equations:

$$\begin{aligned} \frac{du_i}{dt} &+ \mathrm{Re}\frac{\partial u_i}{\partial x_1}\left[\int_0^{x_0} \langle u_1 u_2\rangle \, dx_2 + u_T^2 \left(x_2 - \frac{x_2^2}{H}\right)\right] \\ &+ \mathrm{Re}\, u_2 \delta_{i1}\left[\langle u_1 u_2\rangle + u_T^2\left(1 - \frac{2x_2}{H}\right)\right] + \sum_{1\le j\le 3} u_j\left(\frac{\partial u_i}{\partial x_j} - \left\langle u_j \frac{\partial u_i}{\partial x_j}\right\rangle\right) \\ = -p_{,i} &+ \frac{1}{\mathrm{Re}}\nabla^2 u_i. \end{aligned} \tag{15}$$

These are the equations to which we will apply the weighted residual method.

Notice that while the Navier-Stokes equations are quadratic these equations for the velocity fluctuation are cubic due to the quadratic behavior of the mean velocity in (14).

4.3.1 Eigenfunction Projections

Because the flow is homogeneous in the plane parallel to the wall, we can use a Fourier expansions in the in x_1 and x_3 directions (parallel to the wall). We assume we are given a set of basis functions in the inhomogeneous direction. These basis will not be known numerically, but rather will be provided numerically. These basis functions are called the "empirical eigenfunctions." Expanding the velocity fluctuation u just along x_1 and x_3 dimensions we have

$$\mathrm{u}(x_1, x_2, x_3, t) = \frac{1}{\sqrt{L_1 L_3}} \sum_{\substack{k_1=-\infty \\ k_3=-\infty}}^{\infty} \hat{\mathrm{u}}(x_2, t; k_1, k_3) e^{2\pi i\left(\frac{k_1}{L_1}x_1 + \frac{k_3}{L_3}x_3\right)}.$$

Each of the $\hat{\mathrm{u}}(x_2, t; k_1, k_3)$ can be expanded in series based on the empirical eigenfunctions:

$$\hat{\mathrm{u}}(x_2, t; k_1, k_3) = \sum_{n=1}^{\infty} a_{k_1 k_3}^{(n)}(t)\phi_{k_1 k_3}^{(n)}(x_2).$$

Combining these two expansions gives the following representation of the velocity fluctuation field:

$$u(x_1, x_2, x_3, t) = \frac{1}{\sqrt{L_1 L_3}} \sum_{n=1}^{\infty} \sum_{\substack{k_1=-\infty \\ k_3=-\infty}}^{\infty} a_{k_1 k_3}^{(n)}(t)\, e^{2\pi i(\frac{k_1}{L_1}x_1 + \frac{k_3}{L_3}x_3)} \phi_{k_1 k_3}^{(n)}(x_2). \qquad (16)$$

Notice that (16) is actually three equations, one for each component of u. We denote the components of $\phi_{k_1 k_3}^{(n)}$ by

$$\phi_{k_1 k_3}^{(n)} = \left\langle \phi_{1_{k_1 k_3}}^{(n)}, \phi_{2_{k_1 k_3}}^{(n)}, \phi_{3_{k_1 k_3}}^{(n)} \right\rangle.$$

In addition we use the following identity, which can be computed by almost any symbolic system.

$$\int_0^{L_3} \int_0^{L_1} e^{2\pi i(\frac{p_1}{L_1}x_1 + \frac{p_3}{L_3}x_3)} dx_1 dx_3 = \begin{cases} L_1 L_3 & \text{if } p_1 = p_3 = 0. \\ 0 & \text{otherwise.} \end{cases} \qquad (17)$$

Rather than compute the projection of the entire differential equation, we will illustrate the technique using the following term from (15),

$$\mathrm{Re}\, u_T^2 \frac{\partial u_i}{\partial x_1} \left(x_2 - \frac{x_2^2}{H} \right).$$

We can ignore the $\mathrm{Re}\, u_T^2$ term since it is a constant.

Our goal is to compute $f_{k_1 k_2}^{(n)}$ such that

$$\frac{\partial u_i}{\partial x_1} \left(x_2 - \frac{x_2^2}{H} \right) = \frac{1}{\sqrt{L_1 L_3}} \sum_{n=1}^{\infty} \sum_{\substack{k_1=-\infty \\ k_3=-\infty}}^{\infty} f_{k_1 k_3}^{(n)} e^{2\pi i(\frac{k_1}{L_1}x_1 + \frac{k_3}{L_3}x_3)} \phi_{k_1 k_3}^{(n)}(x_2). \qquad (18)$$

The $f_{k_1 k_3}^{(n)}$ are functions of the $a_{k_1 k_3}^{(n)}$. They are obtained by taking inner products (integrals) of (18) with the orthonormal basis functions. The first two inner products are Fourier transforms, that is

$$\bar{f}_{k_1 k_3} = \frac{1}{\sqrt{L_3}} \int_0^{L_3} \left(\frac{1}{\sqrt{L_1}} \int_0^{L_1} \frac{\partial u_i}{\partial x_1} \left(x_2 - \frac{x_2^2}{H} \right) e^{-2\pi k_1 x_1 / L_1} dx_1 \right) e^{-2\pi k_3 x_3 / L_3} dx_3$$

The final inner product takes the form

$$f_{k_1 k_3}^{(n)} = \int_0^{L_2} \bar{f}_{k_1 k_3} \phi_{k_1 k_3}^{(n)*} dx_2.$$

The final term of U is a polynomial in x_2 and is easily dealt with. The Fourier transform gives

$$\mathcal{F}_{k_1 k_2} \left\{ u_{i,1} u_T^2 \left(x_2 - \frac{x_2^2}{H} \right) \right\} = \left(\sum_{l=1}^{\infty} a_{k_1 k_3}^{(l)} \frac{2\pi i k_1}{L_1} \phi_{i_{k_1 k_3}}^{(l)} \right) u_T^2 \left(x_2 - \frac{x_2^2}{H} \right)$$

```
(* (var l k1 k3)
   (integral (* (make-sampled-function
                  (lambda (x) (- x (/ (* x x) H)))
                 (dot-product (eigen l k1 k3)
                              (conjugate (eigen m k1 k3))))
             :lower-bound 0
             :upper-bound X2))
```

Figure 7: Weyl code for integral

The Galerkin projection is just a simple integral, so

$$\mathcal{G}_{\phi}\left\{u_{i,1}u_T^2\left(x_2-\frac{x_2^2}{H}\right)\right\}=u_T^2\frac{2\pi ik_1}{L_1}\sum_{l=1}^{\infty}a_{k_1k_3}^{(l)}\int_0^{L_2}\left(x_2-\frac{x_2^2}{H}\right)\phi_{i_{k_1k_3}}^{(l)}\phi_{i_{k_1k_3}}^{(n)*}\,dx_2$$

The result many symbolic computations like this is the system of ordinary differential equations shown in Appendix A.

4.3.2 Numerical Computation

Having produced a symbolic system of ordinary differential equations like that shown in Appendix A we must still compute each of the coefficients. This can itself be a rather complex undertaking without the proper abstractions. Consider for instance, the a piece of the sample term computed in the last section:

$$a_{k_1k_3}^{(l)}\int_0^{L_2}\left(x_2-\frac{x_2^2}{H}\right)\phi_{i_{k_1k_3}}^{(l)}\phi_{i_{k_1k_3}}^{(n)*}\,dx_2.$$

The product of the two eigenvectors $\phi_{i_{k_1k_3}}^{(l)}\phi_{i_{k_1k_3}}^{(n)*}$ really means the dot product:

$$\phi_{i_{k_1k_3}}^{(l)}\phi_{i_{k_1k_3}}^{(n)*}=\phi_{1_{k_1k_3}}^{(l)}\phi_{1_{k_1k_3}}^{(n)*}+\phi_{2_{k_1k_3}}^{(l)}\phi_{2_{k_1k_3}}^{(n)*}+\phi_{3_{k_1k_3}}^{(l)}\phi_{3_{k_1k_3}}^{(n)*}.$$

Furthermore, each of the components of the of the eigenvectors are complex valued functions that are only known by their numerical values at selected points.

To deal with this problem we have extended Weyl [15] for this problem to include a new type of a object, a "sampled function." A sampled function is a function from $\mathbb{R}\longrightarrow\mathbb{C}$ (or $\mathbb{R}$) that is represented by its value at certain points. When evaluated at other points, its value is automatically interpolated (extrapolated) form the values at which it is known. Like all objects in Weyl, arithmetic with sampled functions can be performed using the usual Lisp operators, including conjugation.

The eigenvectors $\phi_{k_1k_3}^{(n)}$ are just (Weyl) 3-vectors of sampled functions. Being 3-vectors we can use the dot-product operator to multiply them. The whole expression can thus be computed as shown in Figure 7. Notice that the Weyl code is a direct translation of the more mathematical form given above.

The resulting system of differential equations is somewhat complex, as indicated in Appendix A. One of these equations, when using only a single eigenfunction, has the form

$$\begin{aligned} \dot{a}_1 = 6.1a_1 + 2.1a_1a_2^* + 1.1a_2a_3^* + 0.4a_3a_4^* + 0.3a_4a_5^* \\ - (3.0a_1a_1^* + 3.7a_2a_2^* + 2.4a_3a_3^* + 1.3a_4a_4^* + 0.6a_5a_5^*)a_1. \end{aligned} \tag{19}$$

where we have only given the coefficients to one decimal place for conciseness. The a_i are complex valued functions, so to numerically integrate the equations, each a_i must be converted to a pair of real valued functions. For (19) this gives the following pair of equations.

$$\begin{aligned} \dot{x}_1 &= 6.1x_1 + 2.1(x_2x_1 + y_2y_1) + 1.1(x_3x_2 + y_3y_2) + 0.4(x_4x_3 + y_4y_3) \\ &\quad + 0.3(x_5x_4 + y_5y_4) - 3.0(x_1^2 + y_1^2)x_1 + 3.7(x_2^2 + y_2^2)x_1 \\ &\quad + 2.4(x_3^2 + y_3^2)x_1 + 1.3(x_4^2 + y_4^2)x_1 + 0.6(x_5^2 + y_5^2)x_1, \\ \dot{y}_1 &= 6.1y_1 - 2.1(x_2y_1 - y_2x_1) - 1.1(x_3y_2 - y_3x_2) - 0.4(x_4y_3 - y_4x_3) \\ &\quad - 0.3(x_5y_4 - y_5x_4) - 3.0(x_1^2 + y_1^2)y_1 + 3.7(x_2^2 + y_2^2)y_1 \\ &\quad + 2.4(x_3^2 + y_3^2)y_1 + 1.3(x_4^2 + y_4^2)y_1 + 0.6(x_5^2 + y_5^2)y_1. \end{aligned}$$

Currently, we are integrating these equations with the LSODE package [5, 10]. A typical integration is shown in Figure 8. There are periodic bursts of behavior, where the equations become very stiff. We are currently generating the Jacobians of these equations symbolically to speed the calculation during the stiff regimes. Since the right hand sides of these equations are polynomials, symbolic differentiation does not cause the expressions to grow.

The bursts of activity in Figure 8, when converted into a velocity fluctuation, correspond to the periodic formation of the counter-rotating vortices. Thus the reduced system of ordinary differential equations has the same qualitative behavior has the far more complex Navier-Stokes equations. We are currently studying how to make this correlation more quantitative and how the correlation with physical behavior depends upon the number of empirical eigenfunctions used.

One should note that for a modest number of empirical eigenfunctions, the size of the system of ordinary differential equations becomes very large and their formation and manipulation without symbolic techniques would be impractical.

5 Conclusions

In this paper we have advocated the construction of special purpose simulators for particular scenes, rather than building a general purpose simulator. Towards this end, we have discussed one possible approach to the construction of an environment to enable the construction of such simulators. We have particularly focused on the use of symbolic techniques to transform differential equations into executable code.

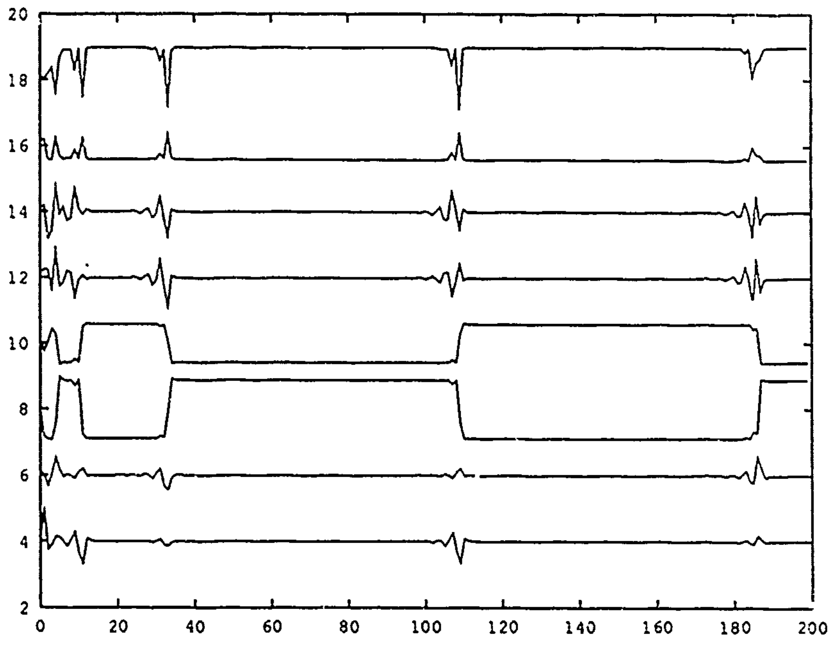


Figure 8: Typical Amplitudes

We have outlined two major areas in which symbolic computation can be effectively used in numerical computations: (1) transforming differential equations into equations that more accurately address the questions being asked of the system under study, and (2) the formation of the numerical integration code itself from libraries of technique fragments. Both of these techniques suggest different organizations of symbolic computation systems than we currently have available.

Acknowledgements

This work has greatly benefited from discussions with other members of the Modeling and Simulation group at Cornell, in particular John Hopcroft, Paul Chew, Jim Cremer, Anne Nierynck and Rick Palmer. The technique of proper orthogonal decomposition and the application of it the determination of coherent structures was brought to my attention by Gal Berkooz, who is a student of John Lumley's. Steve Rapkin has been developing the code that actually generates differential equation code for the boundary layer problem. I am most grateful for their assistance.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-88-K-0591, the National Science Foundation through grant IRI-9006137, the Office of Naval Research through contract N00014-89-J-1946 and in part by the U.S. Army

Research Office through the Mathematical Science Institute of Cornell University.

References

[1] AUBRY, N., HOLMES, P., LUMLEY, J. L., AND STONE, E. The dynamics of coherent structures in the wall region of a turbulent boundary layer. *Journal of Fluid Mechanics 192* (1988), 115–173.

[2] BASHFORTH, F., AND ADAMS, J. C. *Theories of Capillary Action.* Cambridge U. Press, New York, NY, 1883.

[3] CRANK, J., AND NICOLSON, P. *Proceedings of the Cambridge Philosophical Society 32*, 50 (1947).

[4] GEAR, G. W. *Numerical Initial Value Problems in Ordinary Differential Equations.* Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1971.

[5] HINDMARSH, A. C. Odepack, a systematized collection of ODE solvers. In *Scientific Computing*, R. S. Stepleman and et al., Eds. North-Holland, Publ.. Amsterdam, 1983, pp. 55–64.

[6] KIM, J., MOIN, P., AND MOSER, R. T. Turbulence statistics in fully developed channel flow at low Reynolds number. *Journal of Fluid Dynamics 177* (1987), 133–166.

[7] KLINE, S. J., REYNOLDS, W. C., SCHRAUB, F. A., AND RUNDSTADLER, P. W. The structure of turbulent boundary layers. *Journal of Fluid Mechanics 30* (1967).

[8] KRYLOV, N., AND BOGOLIUBOV, N. *Introduction to Non-Linear Mechanics*, vol. 11 of *Annals of Mathematics Studies.* Princeton University Press, Princeton, NJ, 1947.

[9] LUMLEY, J. L. The structure of inhomogeneous turbulent flows. In *Atmospheric Turbulence and Radio Wave Propogation*, A. M. Yaglom and V. I. Tatarski, Eds. Nauka, 1967, pp. 166–178.

[10] PETZOLD, L. R. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *SIAM Journal of Scientific and Statistical Computing 4* (1983), 135–148.

[11] POINCARÉ, H. *Les Méthodes Nouvelles de la Mécanique Céleste.* Gauthier-Villars, Paris, 1892.

[12] RAYLEIGH. On maintained vibrations. *Philosphical Magazine 15* (1883). Series 5.

[13] SANDERS, J. A., AND VERHULST, F. *Averaging Methods in Nonlinear Dynamical Systems*, vol. 59 of *Applied Mathematical Sciences.* Springer-Verlag, New York,, 1985.

[14] VAN DER POL, B. On relaxation-oscillations. *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science 2* (1926).

[15] ZIPPEL, R. E. The Weyl computer algebra substrate. Tech. Rep. 90-1077, Department of Computer Science, Cornell University, Ithaca, NY, 1990.

A Final System of ODE's

$$\dot{a}^{(n)}_{k_1k_3} = \sum_{l=1}^{\infty} a^{(l)}_{k_1k_3} \left\{ \begin{array}{r} -u_\tau^2 \mathrm{Re}\dfrac{2\pi i k_1}{L_1} \displaystyle\int_0^{L_2} (x_2 - \frac{x_2^2}{H})\phi^{(l)}_{i_{k_1k_3}} \phi^{(n)*}_{i_{k_1k_3}} dx_2 \\ -u_\tau^2 \mathrm{Re} \displaystyle\int_0^{L_2} \phi^{(l)}_{2_{k_1k_3}} \phi^{(n)*}_{1_{k_1k_3}} (1 - \frac{2x_2}{H}) dx_2 \\ +\dfrac{1+\alpha_1\nu_T}{\mathrm{Re}} \left[\left(\left(\dfrac{2\pi i k_1}{L_1}\right)^2 + \left(\dfrac{2\pi i k_3}{L_1}\right)^2 \right) \delta_{ln} \right. \\ \left. + \displaystyle\int_0^{L_2} \frac{d^2}{dx_2^2} \phi^{(l)}_{i_{k_1k_3}} \phi^{(n)*}_{i_{k_1k_3}} dx_2 \right] \end{array} \right\}$$

$$+ \frac{1}{\sqrt{L_1 L_3}} \left(1 - \delta_{\substack{k_1=0\\k_3=0}}\right) \sum_{\substack{k_1'=-\infty\\k_3'=-\infty\\m=1,q=1}}^{\infty} a^{(m)}_{k_1'k_3'} a^{(q)}_{k_1-k_1'\,k_3-k_3'} \times$$

$$\left[\int_0^{L_2} \phi^{(m)}_{l_{k_1'k_3'}} \Omega_l' \phi^{(q)}_{i_{k_1-k_1'\,k_3-k_3'}} \phi^{(n)*}_{i_{k_1k_3}} dx_2 \right.$$

$$\left. - \frac{2}{3} l_>^2 \alpha_2 \left(\begin{array}{r} \Omega_l \phi^{(m)}_{k_{k_1'k_3'}}(X_2) \Omega_l \phi^{(q)}_{k_{k_1-k_1'\,k_3-k_3'}}(X_2) \\ + \Omega_l \phi^{(m)}_{k_{k_1'k_3'}}(X_2) \Omega_k \phi^{(q)}_{l_{k_1-k_1'\,k_3-k_3'}}(X_2) \end{array} \right) \phi^{(n)*}_{2_{k_1k_3}}(X_2) \right]$$

$$- \frac{\mathrm{Re}}{L_1 L_3} \sum_{\substack{k_1'=0,k_3'=0\\l=1,m=1,q=1}}^{\infty} a^{(l)}_{k_1k_3} \times$$

$$\int_0^{L_2} \left[\begin{array}{c} \dfrac{2\pi i k_1}{L_1} \phi^{(l)}_{i_{k_1k_3}} \phi^{(n)*}_{i_{k_1k_3}} \Lambda_{k_1'k_3'} \left(a^{(q)}_{k_1'k_3'} a^{(m)*}_{k_1'k_3'} \displaystyle\int_0^{x_2} \phi^{(q)}_{1_{k_1'k_3'}} \phi^{(m)*}_{2_{k_1'k_3'}} dx_2' \right) \\ + \phi^{(l)}_{2_{k_1k_3}} \phi^{(n)*}_{1_{k_1k_3}} \Lambda_{k_1'k_3'} \left(a^{(q)}_{k_1'k_3'} a^{(m)*}_{k_1'k_3'} \phi^{(q)}_{1_{k_1'k_3'}} \phi^{(m)*}_{2_{k_1'k_3'}} \right) \end{array} \right] dx_2$$

$$\Omega_l = \begin{cases} \frac{2\pi i k_1}{L_1} & \text{if } l = 1 \\ \dfrac{d}{dx_2} & \text{if } l = 2 \\ \frac{2\pi i k_3}{L_1} & \text{if } l = 3 \end{cases} \qquad \Omega_l' = \begin{cases} \frac{2\pi i (k_1 - k_1')}{L_1} & \text{if } l = 1 \\ \dfrac{d}{dx_2} & \text{if } l = 2 \\ \frac{2\pi i (k_3 - k_3')}{L_3} & \text{if } l = 3 \end{cases}$$

$$\Lambda_{k_1k_3}(f) = \begin{cases} f & \text{if } k_1 = k_3 = 0 \\ 2f & \text{if } k_1 = 0, k_3 > 0 \\ 2\Re(f) & \text{if } k_1 > 0, k_3 = 0 \\ 4\Re(f) & \text{if } k_1 > 0, k_3 > 0 \end{cases}$$

Robust Point Location in Approximate Polygons

A. James Stewart

TR 91-1208
May 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

Robust Point Location in Approximate Polygons

A. James Stewart
Computer Science Department
Cornell University
jstewart@cs.cornell.edu

Abstract

This paper presents a framework for reasoning about robust geoemtric algorithms. *Robustness* is formally defined and a data structure called an *approximate polygon* is introduced and used to reason about polygons constructed of edges whose positions are uncertain.

A robust algorithm for point location in an approximate polygon is presented. The algorithm uses only the *signature* of the point (not its location) to determine whether the point is inside or outside the polygon.

An approximate polygon could, by shifting its edges back and forth within their error bounds, induce a large number of different line arrangements. The cell C_α with signature α in one such arrangment will be different than the cell C'_α with signature α in another arrangement. This paper proves that, regardless of their positions and shapes, the cells C_α and C'_α are always to the same side of the polygons which induce their respective arrangements.

Introduction

Most geometric algorithms assume that perfect "real" arithmetic is available. When these algorithms are implemented they often fail because this assumption is not borne out; that is, these algorithms are not *robust*. This failure occurs because either the input or the intermediate calculations are imprecise, leading to inconsistent decisions by the algorithm.

This paper presents a framework for reasoning about robust geometric algorithms which operate on polygons. *Robustness* is formally defined and a data structure called an *approximate polygon* is introduced and used to reason about polygons constructed of edges whose positions are uncertain.

A robust algorithm for point location in an approximate polygon is described. The interesting aspect of this algorithm is that in addition to the polygon's position being uncertain, the point's position in the plane does not have to be known; only the point's *signature* is important (that is, its left/right relations to the edges of the polygon). The point location algorithm has immediate practical application to solid modeling, particularly in the robust intersection of polyhedra.

An approximate polygon could, by shifting its edges back and forth within their error bounds, induce a large number of different line arrangements. In each of these arrangements some points with a given signature α may or may not appear, and if they appear, they may be to the interior or to the exterior of the polygon which induces the arrangement. An interesting *uniqueness theorem* is presented which states that in all such line arrangements, the points with signature α in each arrangement are always to the same side of the polygon which induces that arrangement.

Practical Applications

The point location algorithm has immediate practical application to solid modeling. In particular, a solid modeler performing an intersection operation needs to determine whether an edge of one polyhedron intersects a face of another. This is achieved by calculating the intersection of the edge with the plane in which the face lies, and then asking whether this point of intersection is on the interior of the polygon representing the face. If the boundary of the face and the location of the point of intersection are known precisely then this is a trivial problem.

However, polyhedra usually have overconstrained faces and vertices, and the *exact* locations of the vertices, edges, and faces of the polyhedra can require a very large number of bits to represent. Since the input is rounded off to a small number of bits the locations of these features are imprecise. In addition, the location of the point of intersection can be completely unknown, particularly in ill-conditioned cases in which the intersecting edge lies very close to the plane of the face. Thus there is an important practical application for a point location algorithm which handles uncertainty in the face boundary and in the point location.

An approximate polygon can represent a face whose boundaries are not known exactly, and the point location algorithm presented in this paper can determine whether a point whose location is also uncertain lies on the interior of such a face. Since both the location of the boundary and the location of the point are uncertain, the algorithm must make use of some other information. This other information consists of the *signature* of the point; that is, its position (left or right) with respect to each edge of the boundary. It is a reasonable assumption that such information exists since the signature is often derivable from logical information available in the solid modeler (for example, see Karasick's modeler [5]).

Background

The theory of approximate polygons is based upon the "representation and model" approach of Hoffmann, Hopcroft, and Karasick [4]. In this approach the algorithm operates on a computer representation, but presents output as though it were operating on some mathematical model corresponding to the representation.

An approximate polygon is a computer *representation* of some real, mathematical polygon, the *model*. The model is rarely explicitly constructed by the algorithm. An approximate polygon P_{rep} can be thought of as a set of constraints on the topology and position of the implicit model polygon. Any real polygon P satisfying these constraints is considered a model for P_{rep}.

Under the representation and model approach, the definition of robustness is very close to that of Fortune [2]. Consider a geometric problem $\mathcal{P}$ as a function from an input space consisting of *models* to an output space, $\mathcal{P} : \mathcal{I} \rightarrow \mathcal{O}$, and consider an algorithm $\mathcal{A}$ as function from a different input space consisting of *representations* to the same output space, $\mathcal{A} : \mathcal{R} \rightarrow \mathcal{O}$. Given a representation x_{rep}, the set of its models is denoted $\text{MODELS}(x_{rep})$. This leads to a definition of robustness:

An algorithm $\mathcal{A}$ for a problem $\mathcal{P}$ is *robust* if

$$\forall\, x_{rep} \in \mathcal{R},\ \exists\, x \in \text{MODELS}(x_{rep})$$

such that $\mathcal{A}(x_{rep}) = \mathcal{P}(x)$.

Note that we can pick an arbitrary $x \in \text{MODELS}(x_{rep})$. It could be that there are two models x^1 and x^2 such that $\mathcal{P}(x^1) \neq \mathcal{P}(x^2)$. In this case the algorithm could choose to output either $\mathcal{P}(x^1)$ or $\mathcal{P}(x^2)$ and would still be considered to be robust. This leads to a definition of consistency:

A problem $\mathcal{P}$ and a representation $\mathcal{R}$ are *consistent* if

$$\forall\, x_{rep} \in \mathcal{R},\ \forall\, x^1, x^2 \in \text{MODELS}(x_{rep}),$$

$$\mathcal{P}(x^1) = \mathcal{P}(x^2).$$

A definition of *correctness* would be similar to that of robustness, except that the model and representation spaces would be identical and $\text{MODELS}(x_{rep}) = \{x_{rep}\}$.

In evaluating geometric algorithms which use the representation and model approach, the criteria of robustness and consistency should be used in place of the usual criterion of correctness.

Note that, unlike in Fortune's work [2], there is no notion of *stability* in the definition of robustness. That is, there is no notion of the distance between the representation x_{rep} and the model x which allows us to say "the implementation is stable if x is near x_{rep}". However, bounds on the models can be achieved by ensuring that $\text{MODELS}(x_{rep})$ is sufficiently small.

Other approaches

The approach with approximate polygons is most similar to that of Milenkovic's hidden variable method [6]. His method constructs arrangements of pseudolines which are constrained to lie within strips of fixed width. The pseudolines can be considered as a model and the strips as a representation. Milenkovic's pseudoline arrangement algorithm can then be said to be provably robust in the sense of the above definition. It is interesting to note that, as with many algorithms of the "representation and model" variety, the model is never explicitly constructed.

There are several other approaches to building robust algorithms (where "robust" is defined differently). Sugihara [10, 11, 12] emphasizes removing redundant decisions from the algorithm in order to maintain topological consistency. Salesin, Stolfi, and Guibas [8] use what they call *epsilon geometry* to reason about the amount of perturbation of the input required for certain *epsilon predicates* to be true. Construction of robust algorithms is based upon these epsilon predicates. Dobkin and Silver [1] keep track of roundoff error and, when the error becomes too large, increase precision and backtrack to some earlier point in the computation. Segal and Sequin [9] alter the symbolic data to make it more amenable to precise computation, and signal the user when tolerances on the input become too large. (Milenkovic also alters the symbolic data in his "data normalization" approach [6].) Greene and Yao [3] discretize the problem domain, allowing the algorithm to perform exact computations.

In the remainder of the paper approximate polygons are defined, some of their properties are enumerated, the point location algorithm is outlined, and the uniqueness theorm for point location in an approximate polygon is presented.

Definitions

A *polygon* $P = (e_1, e_2, \ldots e_n)$ is an ordered list of directed edges, where each edge e_i lies on a line ℓ_i and only intersects the edges e_{i-1} and e_{i+1} at its endpoints. Each edge is *directed* such that the interior of the polygon it to its right.

An approximate polygon closely mirrors the appearance of a real polygon, as shown in Figure 1. The approximate polygon consists of an ordered list of *bands* corresponding to the edges of the model. The position of the bands in the plane constrains the line equations of the model.

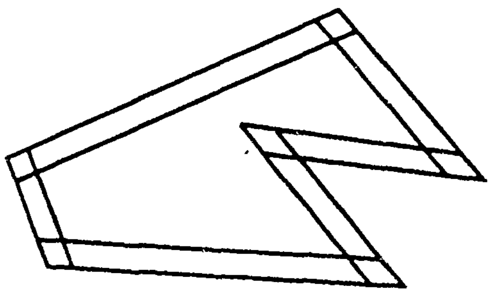


Figure 1: An Approximate Polygon

Just as real polygons are based upon lines, approximate polygons are based on swaths: A *swath* S_i is the region between two lines ℓ_i^{out} and ℓ_i^{in}. These lines have the restriction that $\forall x\ \ell_i^{in}(x) \geq \ell_i^{out}(x)$. The restriction causes the lines to be parallel and conveniently defines the region between them as

$$S_i = \{x \mid \ell_i^{in}(x) \geq 0 \wedge \ell_i^{out}(x) \leq 0\}.$$

Just as an edge is part of a line, a band is part of a swath. Assuming for now that an approximate polygon is represented by an ordered list of swaths, a *band* B_i of an approximate polygon P_{rep} having swaths S_i is the shaded region in Figure 2, defined as

$$B_i = S_i \cap E_{i-1}^i \cap E_{i+1}^i,$$

$$\text{where } E_j^i = \begin{cases} \{x \mid \ell_j^{out}(x) \leq 0\} & \text{if } i/j \text{ is convex} \\ \{x \mid \ell_j^{in}(x) \geq 0\} & \text{if } i/j \text{ is reflex.} \end{cases}$$

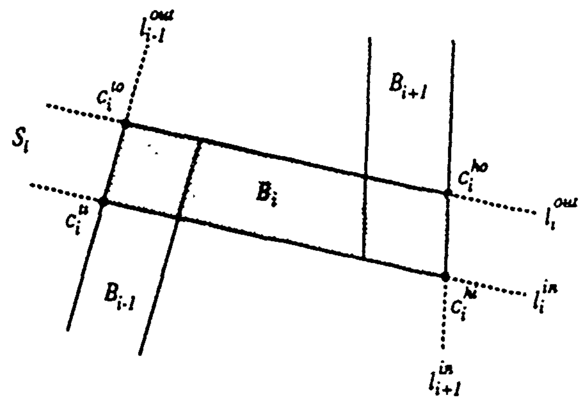


Figure 2: The Pieces of a "Band"

It will be useful to define the corners of a band as c^{ti}, c^{to}, c^{hi}, and c^{ho}, where h, t, i, and o denote head, tail, in and out. The definitions are shown in the following table, and depend on whether the bands adjacent to the corner make a convex or a reflex turn. In Figure 2 the tail of B_i is convex, so the definitions for c^{ti} and c^{to} are chosen from the "convex" column of Table 1. Since the head of B_i is reflex, the definitions of c^{hi} and c^{ho} come from the "reflex" column.

	CONVEX	REFLEX
c^{ti}	$\ell_i^{in} \cap \ell_{i-1}^{out}$	$\ell_i^{in} \cap \ell_{i-1}^{in}$
c^{to}	$\ell_i^{out} \cap \ell_{i-1}^{out}$	$\ell_i^{out} \cap \ell_{i-1}^{in}$
c^{hi}	$\ell_i^{in} \cap \ell_{i+1}^{out}$	$\ell_i^{in} \cap \ell_{i+1}^{in}$
c^{ho}	$\ell_i^{out} \cap \ell_{i+1}^{out}$	$\ell_i^{out} \cap \ell_{i+1}^{in}$

Table 1: Defining the Band Corners

An *approximate polygon* P_{rep} is an ordered list of bands[1] B_i which lie on swaths S_i. $B_i \cap B_j = \emptyset$ *iff* i and j differ by more than one. A real polygon P is a *model* for an approximate polygon P_{rep} if the following constraints are met:

[1] When given as input to an algorithm, the bands are defined exactly with floating point numbers. Subsequent computation on the bands is also done exactly (with extended precision, if necessary).

1. There is a one-to-one correspondence between the lines ℓ_i of P and the swaths S_i of P_{rep}. Assume that ℓ_i corresponds to S_i.

2. Each line $\ell_i(x) = 0$ must lie between the corners of band B_i. That is, it must satisfy the following four constraints (see Figure 2):

$$\ell_i(c^{ti}) \leq 0, \quad \ell_i(c^{to}) \geq 0,$$

$$\ell_i(c^{hi}) \leq 0, \quad \ell_i(c^{ho}) \geq 0.$$

It will be useful later on to talk about the *span* of a band. This is the set of points swept out by all lines which fit within the band. The *left* and *right* of a band are the set of those points to the left and right of the span. By convention, the interior of the approximate polygon is to the right of the band. In Figure 3 the shaded region is $\text{SPAN}(B_i)$ and to its left and right are $\text{LEFT}(B_i)$ and $\text{RIGHT}(B_i)$. For a band B_i, define the set of lines satisfying Condition 2 above as $\text{LINES}(B_i)$.

$$\begin{aligned} \text{SPAN}(B_i) &= \{x \mid \exists\, \ell \in \text{LINES}(B_i),\ \ell(x) = 0\} \\ \text{RIGHT}(B_i) &= \{x \mid \forall\, \ell \in \text{LINES}(B_i),\ \ell(x) < 0\} \\ \text{LEFT}(B_i) &= \{x \mid \forall\, \ell \in \text{LINES}(B_i),\ \ell(x) > 0\} \end{aligned}$$

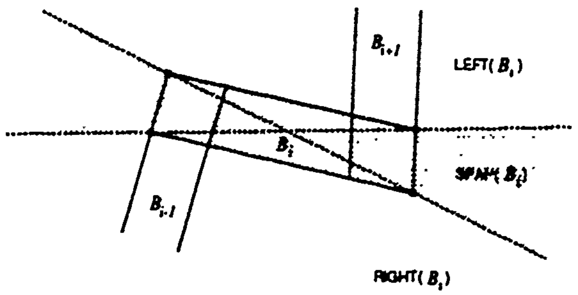


Figure 3: The SPAN of a Band

Some useful properties follow from the previous definitions (these are stated without proof).

1. An approximate polygon is closed and simple.

2. Edge e_i lies completely within B_i.

3. Edges e_i and e_{i+1} intersect within $B_i \cap B_{i+1}$.

4. All models of an approximate polygon are simple.

5. $\text{SPAN}(B_i) \cap \text{SPAN}(B_{i+1}) = B_i \cap B_{i+1}$.

6. If $x \notin \text{SPAN}(B_i)$ then in all models, x lies to the same side of e_i.

7. If $x \in \text{SPAN}(B_{i+1}) - B_i$ then in all models x lies to the same side of e_i.

Robust Point Location in Approximate Polygons

The point location problem would be simple if the exact location of the point were given. However, in most practical applications the point's location is known only to be within some region of uncertainty. In particularly ill-conditioned situations this region of uncertainty can be as large as the polygon itself.

Some practical applications (geometric modelers, for example) can, from other information, logically deduce the LEFT/RIGHT status of the point with respect to each edge of the polygon. Call this L/R sequence the ***signature***. If the polygon's location is known exactly, then in the induced line arrangement a cell decomposition can easily determine whether all points with a given signature lie inside or outside the polygon. It is a different matter, however, when there is uncertainty in the polygon's location. If uncertainty is modeled with an approximate polygon then the following questions must be answered:

> **Question 1 (Robustness)** Given an approximate polygon P_{rep} and a signature $\alpha \in (\text{L}|\text{R})^*$, does P_{rep} have a model P in which the induced line arrangement contains a cell with signature α, and is the cell INSIDE or OUTSIDE the model P?
>
> **Question 2 (Consistency)** Consider that an approximate polygon can have two models, P^1 and P^2, which induce two different line arrangments. These two arrangements each contain a cell with signature α (call them C^1 and C^2). Then is it possible that C^1 is INSIDE P^1 and C^2 is OUTSIDE P^2?

If the answer to Question 2 were affirmative then the signature α and the approximate polygon P_{rep} would not be sufficient information to determine point location, and the problem would not be *consistent.* The Uniqueness Theorem which is presented later proves that this is *not* the case.

Some final definitions

A *signature* $\alpha^P(v)$ is a string in $(\text{L}|\text{R})^*$. The signature denotes the relation of the point v to each edge e_i

of the polygon P. The i_{th} element of $\alpha^P(v)$ is the relation of the point v to edge e_i of the polygon P. $\alpha_i^P(v) = \text{R}$ means that v is to the right of edge e_i in P and $\alpha_i^P(v) = \text{L}$ means that v is to the left of edge e_i in P. The superscripts will be dropped if the polygon in question is evident.

Refer to Figure 3 for the following definitions. A *half-region* is similar to a half-space, except that it has a polygonal boundary. The following half-regions R_i and L_i consist of those points which, in *at least one* model P, are either ON e_i or to the RIGHT or LEFT of e_i, respectively, in that model. Given some $\alpha_i(v)$, the half-region H_i is that region in whose interior v must lie if it is to have $\alpha_i(v)$ as the i^{th} component of its signature. The *interior* of the cell $\tilde{C}_\alpha$ consists of those points which have signature α in *at least one* model.

$$R_i = \text{SPAN}(B_i) \cup \text{RIGHT}(B_i)$$

$$L_i = \text{SPAN}(B_i) \cup \text{LEFT}(B_i)$$

$$H_i = \begin{cases} R_i & \text{if } \alpha_i = \text{R} \\ L_i & \text{if } \alpha_i = \text{L} \end{cases}$$

$$\tilde{C}_\alpha = \bigcap_{i=1}^{n} H_i$$

The next two lemmas will be used to construct the point location algorithm. The first lemma shows that for each point in $\tilde{C}_\alpha$ there exists some model in which the point has signature α; the second lemma shows how to determine whether the point is INSIDE or OUTSIDE that model.

Lemma 1 (Model Existence) *Given an approximate polygon P_{rep} and a signature α, construct $\tilde{C}_\alpha$ as described above. Then for each point v on the interior of $\tilde{C}_\alpha$, there exists* some *model $P \in \text{MODELS}(P_{rep})$ in which v has signature α.*

Proof Since $v \in \tilde{C}_\alpha$, for each i, $v \in H_i$ and there is some edge e_i in the band B_i which has v to the side specified by α_i. These edges join to form a model polygon P in which v has signature α. □

Lemma 2 (Point Location)
Given an approximate polygon P_{rep}, a model polygon $P \in \text{MODELS}(P_{rep})$, and a point v which has a signature α with respect to P, the following are true:

1. *If v is strictly to the interior of P_{rep} (that is, it does not lie on any band B_i) then α, v INSIDE P.*

2. *If v is strictly to the exterior of P_{rep} then v OUTSIDE P.*

3. *If $v \in B_i$, but $v \notin B_{i\pm1}$, then v INSIDE P* iff *$\alpha_i = \text{R}$.*

4. *If $v \in B_i \cap B_{i+1}$ and the $i/i+1$ corner is* convex, *then v INSIDE P* iff *$\alpha_i = \text{R}$ and $\alpha_{i+1} = \text{R}$.*

5. *If $v \in B_i \cap B_{i+1}$ and the $i/i+1$ corner is* reflex, *then v INSIDE P* iff *$\alpha_i = \text{R}$ or $\alpha_{i+1} = \text{R}$.*

Proof In Figure 4 the cases 1 through 5 are demonstrated by the points x_1 through x_5. □

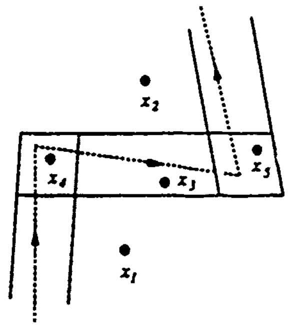


Figure 4: Cases for the Point Location Lemma

Given the Model Existence Lemma and the Point Location Lemma, a point location algorithm can be developed. This algorithm will construct the region $\tilde{C}_\alpha$, pick a point from its interior, and apply the rules of the Point Location Lemma to determine whether the point is INSIDE or OUTSIDE the model in which it has signature α. The following Uniqueness Theorem shows that if one such point is INSIDE its model polygon then *all* such points are INSIDE their respective model polygons (similarly for OUTSIDE).

Theorem 1 (Uniqueness) *Given an approximate polygon P_{rep} and a signature α, if for* some *model polygon in $\text{MODELS}(P_{rep})$ there is a point with signature α which is INSIDE the polygon, then, for* every *model polygon, all points which have signature α with respect to that polygon are INSIDE that polygon (similarly for OUTSIDE).*

Proof in Appendix A.

Point Location Algorithm

The Model Existence Lemma, Point Location Lemma, and Uniqueness Theorem combine to form the point location algorithm shown in Figure 5. Note that the algorithm is quite simple and never actually constructs the model polygon.

Lemma 3 (Robustness) *The point location algorithm is robust.*

1. Compute $\tilde{C}_\alpha$.
2. If $\tilde{C}_\alpha = \emptyset$ then no model of P_{rep} induces a cell with signature α.
3. Pick a point w on the interior of $\tilde{C}_\alpha$.
4. Apply the Point Location Lemma to determine whether w is INSIDE or OUTSIDE of the models in which it has signature α.

Figure 5: Point Location Algorithm

Proof This follows directly from the Model Existence Lemma and the Point Location Lemma. □

Lemma 4 (Consistency) *The approximate point location problem is consistent.*

Proof This follows directly from the Uniqueness Theorem. □

Lemma 5 (Complexity) *The point location algorithm has time complexity $\mathcal{O}(n^2)$.*

Proof Step 1 of the algorithm can be accomplished by computing the arrangement of half-regions in $\mathcal{O}(n^2)$ time. This is done by computing the arrangement of the $3n$ lines which define the n half-regions H_i, then joining adjacent cells which are separated by a line segment which is *not* part of the boundary of some H_i. Those cells separated by a line segment which *is* part of the boundary of some H_i will have signatures which differ in a single position, so the signature of each cell can be found in constant time.

The remaining steps of the algorithm take constant time. Step 3 is easily accomplished given the convex decomposition of $\tilde{C}_\alpha$ which is computed simultaneously with $\tilde{C}_\alpha$ itself. Thus, the overall running time is $\mathcal{O}(n^2)$. □

Summary

Most geometric algorithms are not *robust*; they fail due to inexact input or with inexact intermediate computations. This paper has introduced (a) formal definitions of robustness and consistency, and (b) the notion of an *approximate polygon*, along with several of its properties. With these, one can formally develop robust and consistent algorithms that deal with inexact polygons.

One such algorithm for point location in an approximate polygon has been presented. The algorithm is particularly suited for practical application in a solid modeler because it assumes uncertainty in both the polygon position and the point position. The point location algorithm has been proved robust, and the point location problem has been shown to be consistent.

Acknowledgments

I am pleased to acknowledge the many helpful discussions which I have had with Chanderjit Bajaj and Paul Chew during the course of this work. Improvements to the algorithm were contributed by Victor Milenkovic, and this line of research was originally suggested by John Hopcroft.

My research has been supported by Nasa Training Grant NGT-50327. This work is also supported by Office of Naval Research Contract N00014-88K-0591, Office of Naval Research Grant N00014-89J-1946, and National Science Foundation Grant IRI-9006137.

Appendix A

Theorem 1 (Uniqueness)
Given an approximate polygon P_{rep} and a signature α, if for some model polygon in MODELS(P_{rep}) *there is a point with signature α which is* INSIDE *the polygon, then, for every model polygon, all points which have signature α with respect to that polygon are* INSIDE *that polygon (similarly for* OUTSIDE*).*

Proof (by contradiction): Let $\alpha^k(x)$ be the signature of point x in model P^k. Let e_i^k be edge e_i in model P^k. Then assume the following:

$$\exists P^1, P^2 \in \text{MODELS}(P_{rep}),\ \ \exists u, v \in \Re^2,$$

$$u \text{ INSIDE } P^1 \ \wedge \ v \text{ OUTSIDE } P^2 \ \wedge \ \alpha^1(u) = \alpha^2(v).$$

The theorem is proved by showing that there is some edge e_k which always separates u and v, violating this assumption.

Lemma 6 *One of u or v must lie within the boundary of P_{rep}.*

Proof Assume that neither u nor v is within the boundary. Then by the initial assumption and the Point Location Lemma u and v lie on opposite sides of the boundary. Say u is inside and v is outside. Then the segment $\overline{uv}$ must traverse both of the parallel sides of some band B_i, as shown in Figure 6. From the definition of the corners of B_i and the definition of SPAN(B_i), u and v lie to different sides of SPAN(B_i). Then by Property 6 u lies to one side of all models of P_{rep} and v lies to the other side. Thus, in the models P^1 and P^2, $\alpha_i^1(u) \neq \alpha_i^2(v)$ and the initial assumption is contradicted. □

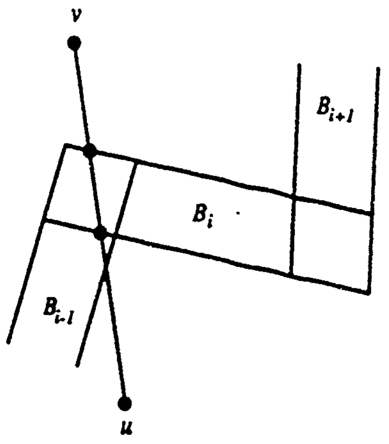


Figure 6: $\overline{uv}$ crosses some B_i

From Lemma 6 assume without loss of generality that u lies in the boundary of P_{rep}.

Lemma 7 *The point u lies in some band B_i such that $\alpha_i^1(u) = $ R and $\alpha_i^2(v) = $ R.*

Proof u is INSIDE P^1 by the initial assumption and is in some band by Lemma 6. If $u \in B_k$ and $u \notin B_{k\pm1}$, then by the Point Location Lemma u is to the RIGHT of e_k^1. Choose $i = k$. If $u \in B_k \cap B_{k+1}$ then by the Point Location Lemma u is to the right of at least one of e_k^1 or e_{k+1}^1. Choose i to be k or $k+1$ to satisfy the lemma. Then by the initial assumption, $\alpha_i^1(u) = $ R means that $\alpha_i^2(v) = $ R also. □

Lemma 8 *On the segment $\overline{uv}$ there is some point $x \neq v$ which is inside P^2. Furthermore, e_i^2 does* not *intersect $\overline{uv}$ between x and v.*

Proof

Case 1: $u \in B_i$, but $u \notin B_{i\pm1}$. If $\alpha_i^2(u) = $ R, then u INSIDE P^2 (by the Point Location Lemma), so choose $x = u$ and we are done. Otherwise consider the case in which $\alpha_i^2(u) \neq$ R.

Refer to Figure 7. The point u lies in B_i and not in B_{i-1}, so by Property 7, in all models P^k, u is to the same side of e_{i-1}^k. In particular, $\alpha_{i-1}^1(u) = \alpha_{i-1}^2(u)$. By the initial assumption, $\alpha_{i-1}^1(u) = \alpha_{i-1}^2(v)$, so $\alpha_{i-1}^2(u) = \alpha_{i-1}^2(v)$. By a similar argument for e_{i+1},

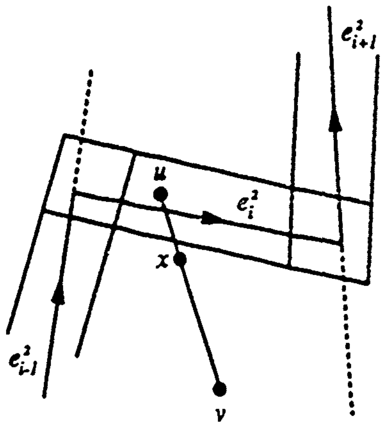


Figure 7: x exists if $u \in B_i$

$\alpha_{i+1}^2(u) = \alpha_{i+1}^2(v)$. Thus u and v lie between two lines touching the endpoints of e_i^2.

Since $\alpha_i^2(u) \neq$ R and since by Lemma 7 $\alpha_i^2(v) = $ R, $\overline{uv}$ must cross the *line* defined by e_i^2. But u and v lie between the two lines touching the endpoints of e_i^2, so $\overline{uv}$ must cross the *edge* e_i^2. Since by Property 4 all models P are simple, there is a small neighborhood to the right of e_i^2 which contains only points interior to P. The segment $\overline{uv}$ passes through this neighborhood, so there is some point $x \neq v$ on $\overline{uv}$ which is INSIDE P^2, and $\overline{xv} \cap e_i^2 = \emptyset$.

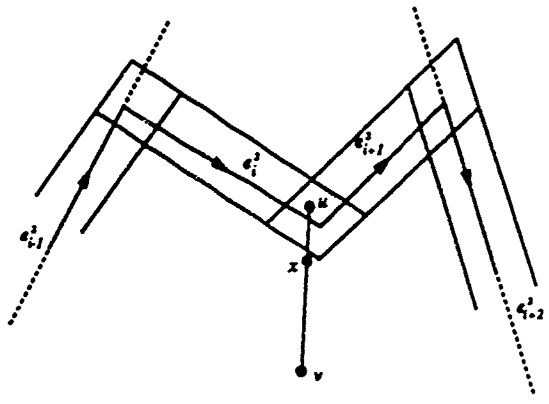


Figure 8: x exists if $u \in B_i \cap B_{i+1}$

Case 2: $u \in B_i \cap B_{i+1}$. Refer to Figure 8. By an argument similar to Case 1, u and v must lie to the same side of e_{i-1}^k in all models P^k, and must lie to the same side of e_{i+2}^k in all models P^k. If u INSIDE P^2 then choose $x = u$ and we are done. Otherwise, if u

OUTSIDE P^2 then the edges e_i^2 and e_{i+1}^2 must separate u and v. Thus the segment $\overline{uv}$ must cross either e_i^2 or e_{i+1}^2, and in doing so must pass through a neighborhood of interior points to the right of the edge that it crosses. Therefore there is some point $x \neq v$ on $\overline{uv}$ which is INSIDE P^2, and $\overline{xv} \cap e_i^2 = \emptyset$. □

Lemma 9 *In P^2 there is some edge e_j^2 which crosses $\overline{uv}$ such that $\alpha_j^2(u) = \text{R}$ and $\alpha_j^2(v) = \text{L}$. Furthermore, e_j^2 can be chosen such that no other e_k^2 crosses $\overline{uv}$ between e_j^2 and v.*

Proof From lemma 8, $\overline{uv}$ contains a point x which is INSIDE P^2, and by the initial assumption, v OUTSIDE P^2. From the Jordan curve theorem we know that $\overline{xv}$ must cross the boundary of P^2 on some edge e_j^2 with x to the inside (right) of e_j^2 and v to the outside (left) of e_j^2. From the ordering of points along $\overline{uv}$ $(u \leq x < v)$, if $\alpha_j^2(x) = \text{R}$ then $\alpha_j^2(u) = \text{R}$ also. If there are many candidates for e_j^2, choose that which is closest to v to satisfy the second part of the lemma. □

Lemma 10 $B_j \cap B_i = \emptyset$

Proof If $|i-j| > 1$ then by the definition of an approximate polygon $B_i \cap B_j = \emptyset$. So we only have to consider $|i-j| \leq 1$. But by Lemma 9 e_j^2 intersects $\overline{uv}$ between x and v, and by Lemma 8 e_i^2 does *not* intersect $\overline{uv}$ between x and v. So $e_i^2 \neq e_j^2$ and hence $i \neq j$.

Assume that $j = i-1$. Then, by the initial assumption, $\alpha_j^2(v) = \text{L}$ means that $\alpha_j^1(u) = \text{L}$. By Lemma 9, $\alpha_j^2(u) = \text{R}$. Since u is to different sides of e_j in P^1 and P^2, u must lie in SPAN(B_j). Since u also lies in B_i, by Property 5 u lies in the corner $B_i \cap B_j$.

Suppose corner i/j is convex. Since u INSIDE P^1, by the Point Location Lemma $\alpha_i^1(u) = \text{R}$ and $\alpha_j^1(u) = \text{R}$. But, by the initial assumption, $\alpha_j^1(u) = \text{R}$ means that $\alpha_j^2(v) = \text{R}$, contradicting lemma 9. So corner i/j is not convex.

Suppose corner i/j is reflex. Refer to Figure 9. By Lemma 9, $\alpha_j^2(u) = \text{R}$ and $\alpha_j^2(v) = \text{L}$, and by Lemma 7, $\alpha_i^2(v) = \text{R}$. But if $\overline{uv}$ is to intersect e_j^2 then it must also intersect e_i^2 closer to v, as shown in the Figure. This contradicts Lemma 9, which states the e_j^2 is the closest intersection to v. So corner i/j is not reflex.

Thus the assumption is false that $j \neq i-1$. By a similar argument $j \neq i+1$. Therefore $B_j \cap B_i = \emptyset$. □

Lemma 11 $u \in \text{SPAN}(B_j) - B_j$

Proof: By lemma 9, $\alpha_j^2(u) = \text{R}$ and $\alpha_j^2(v) = \text{L}$. By the initial assumption, $\alpha_j^2(v) = \text{L}$ means that $\alpha_j^1(u) =$

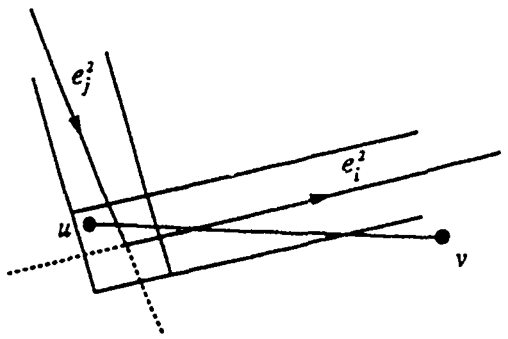


Figure 9: $\overline{uv}$ must intersect e_i^2 closer to v

L. Since $\alpha_j(u)$ is different in models P^1 and P^2, u must lie in SPAN(B_j). From lemma 10, u cannot lie in B_j since it already lies in B_i. □

Lemma 12 *Define k such that $B_j \cap B_k$ is closest to u. Then u and v are on opposite sides of e_k^2.*

Proof Refer to Figure 10. Note that $k = j \pm 1$, otherwise B_j and B_k wouldn't intersect at all. In any model P the point u and the edge e_j are on opposite sides of the line defined by e_k ($u \in \text{SPAN}(B_j) - B_j$ by lemma 11, and since B_k is closest to u, it separates u from the rest of B_j, which contains e_j). Thus $\overline{uv}$ must intersect e_j^2 at some point z to the side of e_k^2 which is opposite to u. From the ordering of points along $\overline{uv}$ $((u < z < v)$, v must also be opposite to u. □

Lemma 13 $\alpha_k^1(u) = \alpha_k^2(u)$.

Proof By Lemma 12, $u \in \text{SPAN}(B_j) - B_j$. By Property 5, SPAN$(B_j) \cap$ SPAN$(B_k) = B_k \cap B_k$, so with a bit of algebra we can conclude that $u \notin \text{SPAN}(B_k)$. Then by Property 6, $\alpha_k^1(u) = \alpha_k^2(u)$. □

By lemma 12 there is some edge e_k^2 in model P^2 such that $\alpha_k^2(u) \neq \alpha_k^2(v)$, and by lemma 13, $\alpha_k^1(u) = \alpha_k^2(u)$. So $\alpha_k^1(u) \neq \alpha_k^2(v)$. But this contradicts the initial assumption, so the theorem is proved by contradiction.

□

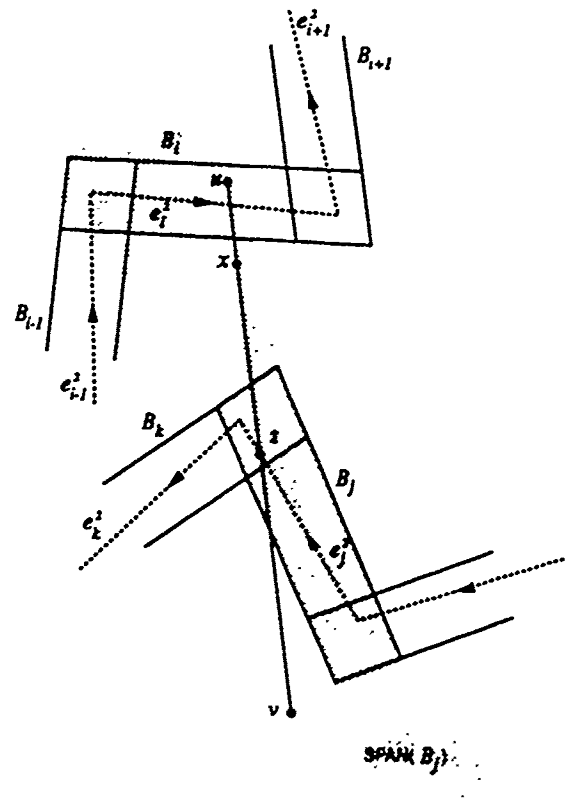


Figure 10: u and v are on opposite sides of e^2_k

References

[1] D. Dobkin and D. Silver. Recipes for geometry & numerical analysis - Part I: An empirical study. In *Annual Symposium on Computational Geometry*, pages 93–105, 1988.

[2] S. Fortune. Stable maintenance of point set triangulations in two dimensions. In *IEEE Annual Symposium on Foundations of Computer Science*, pages 494–499, 1989.

[3] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. In *IEEE Annual Symposium on Foundations of Computer Science*, pages 143–152, October 1986.

[4] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick. Towards implementing robust geometric computations. In *Annual Symposium on Computational Geometry*, pages 106–117, June 1988.

[5] M. Karasick. *On the Representation and Manipulations of Rigid Solids*. PhD thesis, McGill University, 1989.

[6] V. J. Milenkovic. *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic*. PhD thesis, Carnegie Mellon University, 1988.

[7] V. J. Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. In *IEEE Annual Symposium on Foundations of Computer Science*, pages 500–505, 1989.

[8] D. Salesin, J. Stolfi, and L. Guibas. Epsilon geometry: Building robust algorithms from imprecise calculations. In *Annual Symposium on Computational Geometry*, pages 208–217, 1989.

[9] M. Segal and C. Sequin. Partitioning polyhedral objects into nonintersecting parts. In *IEEE Computer Graphics and Applications*, pages 53–67, January 1988.

[10] K. Sugihara and M. Iri. Construction of the Voronoi diagram for over 10^5 generators in single-precision arithmetic. In *Abstracts of the First Canadian Conference on Computational Geometry*, page 42, August 1989.

[11] K. Sugihara and M. Iri. Two design principles of geometric algorithms in finite-precision arithmetic. *Appl. Math. Lett.*, 2(2):203–206, 1989.

[12] K. Sugihara, Y. Ooishi, and T. Imai. Topology-oriented approach to robustness and its applications to several Voronoi-diagram algorithms. In *Canadian Conference on Computational Geometry*, pages 36–39, August 1990.

Rational Function Decomposition*

Richard Zippel**

TR 91-1209
May 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This paper will be presented at the International Symposium on Symbolic and Algebraic Computing, July 14, 1991, in Bonn, Germany.
**This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-88-K-0591, the National Science Foundation through grant IRI-9006137, the Office of Naval Research through contract N00014-89-J-1946 and in part by the U.S. Army Research Office through the Mathematical Science Institute of Cornell University.

Rational Function Decomposition

Richard Zippel*
Cornell University
Ithaca, NY 14853
rz@cs.cornell.edu

May 21, 1991

Abstract

This paper presents a polynomial time algorithm for determining whether a given univariate rational function over an arbitrary field is the composition of two rational functions over that field, and finds them if so.

1 Introduction

The problem of determining if a function can be written as the composition of two "smaller" functions $f(x) = g(h(x))$ has been of interest for a long time. Until now, work has focused on the univariate, polynomial version of this problem: When can the polynomial $f(x)$ be written as $g(h(x))$, where both $g(x)$ and $h(x)$ are polynomials? The original work in the symbolic computation community was presented in 1976 [2], but the algorithms, which in the worst case required exponential time, were not published until 1985 [3]. This was soon followed by the work of Kozen and Landau [11] who provided a polynomial time algorithm for decomposition of polynomials over fields of characteristic zero, which did not require factorization of polynomials. Some additional improvements and analysis of the positive characteristic case where then presented by von zur Gathen [23, 21, 22]. A number of other papers have since been published on different extensions and variations of this problem [1, 7, 5, 4].

All of these results deal with polynomial decomposition. The generalization to rational functions, which has significantly wider applicability, appears to be a far harder problem. Notice that in the polynomial case, the degree of g and h must divide the degree of f. This limits the number of different polynomials that must be considered and even allows one to solve the problem by looking for solutions of non-linear algebraic equations (admittedly in exponential time). When f, g and h are rational functions, there is no immediately obvious bound on the degrees of the numerators of g and h, since the numerator and denominator of $g(h(x))$ could have a common factor. In fact, no such common factor can arise, as we prove below.

Furthermore, we demonstrate that in the rational function case, g and h can be determined from f in polynomial time. This algorithm is valid even if the charactertistic of the field is positive, which for the polynomial case is not a completely resolved problem. One other difference between our approach and other approaches, is that in this paper we obtain a decomposition over the field of definition of $f(x)$. Thus we may fail to find rational function decompositions that exist over

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-88-K-0591, the National Science Foundation through grant IRI-9006137, the Office of Naval Research through contract N00014-89-J-1946 and in part by the U.S. Army Research Office through the Mathematical Science Institute of Cornell University.

algebraic extensions. Such issues do not arise for the corresponding problem of polynomials over a field of characteristic zero, but do for polynomials over fields of finite characteristic.

Section 2 provides some general background material. In Section 3 we present the new algorithms for rational function decomposition. Finally, we comment on previous work in and give some conclusions in Section 4.

2 Preliminaries

Let $f(x)$ be a rational function in x with coefficients in a field k. We extend the notion of degree of a polynomial by defining the *degree* of $f(x)$, denoted by $\deg f$, to be the maximum of the polynomial degrees of the (relatively prime) numerator and denominator of f. The degree of the field $k(x)$ over $k(f(x))$ is the degree of f, if f is a polynomial. This remains true even if f is a rational function, as shown by the following proposition.

Proposition 1 *Let $k(x)$ be an extension of the field $k(f(x))$ where $f(x)$ is a rational function of degree n. Then $[k(x):k(f(x))] = n$.*

Proof: Denote the numerator of $f(x)$ by $p(x)$ and the denominator by $q(x)$. We can instead consider the isomorphic fields $k(y) \cong k(f(x))$ and

$$k(y)[x]/(p(x) - yq(x)) \cong k(x).$$

$P(x, y) = p(x) - yq(x)$ is primitive as a polynomial in y since $p(x)$ and $q(x)$ are relatively prime. Since it is linear in y it is irreducible. Therefore, the degree of x over the field $k(y)$ is

$$\deg_x P(x, y) = \max(\deg p, \deg q) = \deg f.$$

□

Let $f(x) = g(h(x))$ be a rational function decomposition over a field k. The following proposition provides bounds on the degrees of $g(x)$ and $h(x)$ in terms of the degree of $f(x)$. In principle, this result gives an algorithm for rational function decomposition, albeit an exponential time algorithm.

Proposition 2 *Assume $f(x)$, $g(x)$ and $h(x)$ are elements of $k(x)$ such that $f(x) = g(h(x))$. Then*

$$\deg f = (\deg g) \cdot (\deg h)$$

Proof:

Consider the fields shown in Figure 1. The degrees of the extensions are $[k(x):k(h(x))] = \deg h$, $[k(x):k(f(x))] = \deg f$ and $[k(y):k(g(y))] = \deg g$. $k(h(x))$ is an algebraic extension of $k(f(x))$ inside $k(x)$. Thus,

$$\begin{aligned}\deg f &= [k(x):k(f(x))] \\ &= [k(x):k(h(x))] \cdot [k(h(x)):k(f(x))] \\ &= [k(x):k(h(x))] \cdot [k(y):k(g(y))] \\ &= (\deg h) \cdot (\deg g),\end{aligned}$$

using Proposition 1. □

A function that is the ratio of to linear polynomials is called a *fractional linear function, viz.*

$$\lambda(x) = (ax + b)/(cx + d).$$

$$\begin{tikzcd}
k(x) \arrow[d, dash] & \\
k(h(x)) \arrow[d, dash] & k(y) \arrow[l, "\varphi_h"'] \arrow[d, dash] \\
k(f(x)) & k(g(y)) \arrow[l, "\varphi_h"']
\end{tikzcd}$$

Figure 1: Fields involved in decomposition

Fractional linear functions have degree 1. If two fields $k(f_1(x))$ and $k(f_2(x))$ are isomorphic then there exist rational functions such that

$$f_1(x) = R_1(f_2(x))$$
$$f_2(x) = R_2(f_1(x)) = R_2(R_1(f_2(x)))$$

By Proposition 2 $(\deg R_1) \cdot (\deg R_2) = 1$ and R_1 and R_2 must be fractional linear functions.

We say that two rational functions are linearly equivalent if there exists fractional linear functions λ_1 and λ_2 such that

$$f(x) = \lambda_1(g(\lambda_2(x))).$$

Two decompositions (polynomial or rational function)

$$\begin{aligned} f &= g_1 \circ g_2 \circ \cdots \circ g_m \\ &= h_1 \circ h_2 \circ \cdots \circ h_n \end{aligned}$$

are said to be *equivalent* if $m = n$ and g_i is linearly equivalent to h_i.

The link between field structure and function decomposition comes from *Lũroth's theorem*, which was proven by Lũroth [15] for $k = \mathbb{C}$ and by Steinitz in general [18].

Proposition 3 (Lũroth) *If $k \subsetneq K \subset k(x)$ then $K = k(g(x))$ where $g(x)$ is a rational function in x over k.*

An elementary proof of Lũroth's theorem may be found in van der Waerden [20]. An effective proof appears in Weber [24] §124, and in English in Schinzel [17].

The key insight in studying functional decomposition is the following corollary of Lũroth's theorem.

Proposition 4 *Let k be an arbitrary field and $f(x)$ a rational function over k. There is a one to one correspondence between the lattice of subfields between $k(x)$ and $k(f(x))$ and the rational function decompositions of $f(x)$ up to equivalence.*

Proof: If $f(x)$ has a nontrivial decomposition $f(x) = g(h(x))$, then $k(h(x))$ will be an intermediate field between $k(x)$ and $k(f(x))$. Conversely, if K is field intermediate between $k(x)$ and $k(f(x))$ then it must be of the form $k(h(x))$, where $h(x)$ is a rational function in x. $k(h(x))$ is canonically isomorphic to $k(y)$ as shown in Figure 1, where $\varphi_h(y) \mapsto h(x)$. $k(f(x))$ is intermediate between

$k(y) = k(h(x))$ and k, so by Lüroth's theorem, there is a rational function $g(y)$ such that $k(f(x)) = k(g(y))$. Thus $f(x)$ is linearly equivalent to $g(h(x))$. $\square$

The following two propositions follow from Proposition 2 and are quite useful.

Proposition 5 *Let k be an arbitrary field and g_1 and g_2 relatively prime elements of $k[x]$. Then for all polynomials $h(x) \in k[x]$, $g_1(h(x))$ and $g_2(h(x))$ are relatively prime.*

Proof: Without loss of generality assume that $\deg g_1 \geq \deg g$. Define $g(x)$ to be the ratio of $g_1(x)$ and $g_2(x)$. Since g_1 and g_2 are relatively prime and $\deg g_1 \geq \deg g_2$, $\deg g(x) = \deg g_1$. Let

$$f(x) = g(h(x)) = \frac{g_1(h(x))}{g_2(h(x))} = \frac{f_1(x)}{f_2(x)},$$

where f_1 and f_2 are relatively prime. Thus

$$\deg f_i(x) \leq \deg g_i(h(x)) = (\deg g_i) \cdot (\deg h),$$

where equality holds if and only if $g_1(h(x))$ and $g_2(h(x))$ are relatively prime. Furthermore, $\deg f_1 \geq \deg f_2$ so $\deg f = \deg f_1$. By Proposition 2

$$\deg f(x) = (\deg g) \cdot (\deg h) = (\deg g_1) \cdot (\deg h)$$

so $\deg f_1(x) = (\deg g_1) \cdot (\deg h)$ and $g_1(h(x))$ and $g_2(h(x))$ are relatively prime. $\square$

The argument of previous proposition applies equally when $h(x)$ is a rational function. In this case, it is best to view g_1 and g_2 as bivariate homogeneous functions of the same degree, which gives the following result.

Proposition 6 *Let g_1 and g_2 be relatively prime, homogeneous polynomials in two variables. If h_1 and h_2 are also relatively prime polynomials, then $g_1(h_1, h_2)$ and $g_2(h_1, h_2)$ are also relatively prime.*

Notice that the requirement that g_1 and g_2 be homogeneous is necessary as the following example shows:

$$\left.\begin{aligned} g_1(x,y) &= x+1 \\ g_2(x,y) &= y-2 \\ h_1(t) &= t \\ h_2(t) &= t^2+1 \end{aligned}\right\} \longrightarrow \left\{\begin{aligned} g_1(h_1,h_2) &= t+1 \\ g_2(h_1,h_2) &= t^2-1 \end{aligned}\right.$$

As a consequence of Proposition 6, rational function decomposition can be viewed as a coupled polynomial decomposition problem, *viz.*

$$\begin{aligned} f_1(x,y) &= g_1(h_1(x,y), h_2(x,y)), \\ f_2(x,y) &= g_2(h_1(x,y), h_2(x,y)), \end{aligned}$$

where f_i, g_i and h_i are homogeneous polynomials and the pairs $\{f_1, f_2\}$, $\{g_1, g_2\}$ and $\{h_1, h_2\}$ have the same degree.

3 Rational Function Decomposition

The bounds of Proposition 2 provide significant insight into rational function decomposition. In particular, if the degree of $f(x)$ is prime, then it has no non-trivial decomposition. A simple, exponential time algorithm for determining a decomposition can be constructed by using undetermined coefficients. Assume that $\deg f = rs$ and we are looking for a decomposition $f(x) = g(h(x))$, where $\deg g = r$ and $\deg h = s$. We can write g and h in terms undetermined coefficients, *e.g.*

$$g(x) = \frac{g_n(x)}{g_d(x)} = \frac{g_0x^r + g_1x^{r-1} + \cdots + g_r}{g_{r+1}x^r + g_{r+2}x^{r-1} + \cdots + g_{2r+1}}.$$

There are $2r+2$ undetermined coefficients in $g(x)$ and $2s+2$ in $h(x)$. By Proposition 6, we can treat the numerator and denominator of $f(x)$ independently. Equating the coefficients of x^i in the following equations gives a system of $2rs+2$ algebraic equations in the g_i and h_i.

$$\begin{aligned} f_0x^{rs} &+ \cdots + f_{rs} \\ &= g_0h_n(x)^r + \cdots + g_rh_d(x)^r \\ f_{rs+1}x^{rs} &+ \cdots + f_{2rs+1} \\ &= g_{r+1}h_n(x)^r + \cdots + g_{2r+1}h_d(x)^r \end{aligned} \tag{1}$$

Any decomposition of $f(x)$ is a solution to this system of equations. Conversely, any solution to this system for which $\deg g = r$ and $\deg h = s$ gives a decomposition of $f(x)$. However, this approach is not very efficient. Nonetheless, it does demonstrate the existence of an algorithm.

The efficient techniques that have been developed all tend to be divided into two phases, computing $h(x)$ and then given $h(x)$ computing $g(x)$. (The hard part is finding $h(x)$.) We discuss the phases out of order for simplicity. Determining g from f and h is discussed in Section 3.1, while the determination of h is discussed in Section 3.2.

3.1 Determining g from f and h

The most direct way to obtain $g(x)$ such that $f(x) = g(h(x))$, when f and h are known is to explicitly solve the *linear* equations for the coefficients of $g(x)$ that arise from (1). This approach is discussed in detail by Dickerson [5, 4] as "computing the left composition factor." In this section we present a simple analytic technique that relies on reversion of power series and is valid when the coefficient field has characteristic 0.

Let λ_f be a fractional linear function such that $\hat{f} = \lambda_f \circ f$ has a zero at 0. Define $\hat{h}$ and λ_h similarly. If $\hat{f} = \hat{g} \circ \hat{h}$ then

$$f(x) = (\lambda_f^{-1} \circ \hat{g} \circ \lambda_h) \circ h(x),$$

and $g(x) = (\lambda_f^{-1} \circ \hat{g} \circ \lambda_h)(x)$. So without loss of generality we can assume $f(0) = h(0) = 0$.

$h(x)$ has a power series expansion of the form

$$h(x) = h_\ell x^\ell + h_{\ell+1}x^{\ell+1} + \cdots$$

Using standard techniques [10] we can obtain a power series in t for x in $t = h(x)$

$$x = h^{-1}(t) = h'_1 t^{1/\ell} + h'_2 t^{2/\ell} + \cdots.$$

Replacing x by this power series in the power series for $f(x)$ we get a power series in t. If there are any fractional powers then there does not exist a "left composition factor." Compute the first $2r$ terms of the power series expansion of $f(h^{-1}(x))$ at 0. The (r,r) Padé approximate [16] to this power series is the only possible candidate for $g(x)$. This power series technique may be easier to program than Dickerson's technique, and using fast power series techniques [12] it might have better asymptotic complexity.

$$\begin{tikzcd}
k(x) \arrow[r, dash] \arrow[d, dash] & E[\alpha]/(f(\alpha)-t) = F \arrow[d, dash] \\
k(h(x)) \arrow[r, dash] \arrow[d, dash] & E[\beta]/(h(\beta)-t) \arrow[d, dash] \\
k(f(x)) \arrow[r, dash] & k(t) = E
\end{tikzcd}$$

Figure 2: Field Structure

3.2 Determination of h

For rational function decomposition, we determine $h(x)$ by explicitly determining a subfield of $k(x)$ and then use a constructive version of Lũroth's theorem to compute a generator for the subfield. The tower of fields we will be working with is shown in Figure 2. Note that the fields on the same horizontal line in Figure 2 are isomorphic. By Proposition 3 every subfield of F is of the form $k(h(x))$ and there exists a rational function g such that $g(h(x)) = f(x)$, since $k(f(x))$ lies between $k(y) = k(h(x))$ and k. Thus every non-trivial subfield of F yields a non-trivial decomposition of $f(x)$.

To illustrate our procedure consider the following example:

$$\begin{aligned} f(x) &= \left(\frac{x^2+1}{x^2-2}\right) \circ \left(\frac{x^2+1}{x^2+2}\right) \\ &= -\frac{2x^4+6x^2+5}{x^4+6x^2+7} = \frac{f_n(x)}{f_d(x)}, \end{aligned}$$

where f_n and f_d are relatively prime. We want to find an intermediate field between $k(x)$ and $k(f(x))$. Our first step is to convert these fields to a more conventional form. If $E = k(t) = k(f(x))$ and $E[\alpha] = k(x)$ then α satisfies the minimal polynomial

$$\hat{f}(t, Z) = f_n(Z) - t f_d(Z) = (t+2)Z^4 + (6t+6)Z^2 + 7t + 5.$$

This polynomial's factorization over $E[\alpha]$ is

$$\hat{f}(t, Z) = (Z-\alpha)(Z+\alpha)((t+2)Z^2 + (t+2)\alpha^2 + 6(t+1)). \tag{2}$$

Over a proper subfield of $E[\alpha]$, $\hat{f}(t, Z)$ will not factor so much. In particular, over a subfield it cannot have a linear factor. Given (2), the only possible factors of $\hat{f}(t, Z)$ over the subfield $E[\beta]$ are $Z - \alpha^2$ and $((t+2)Z^2 + (t+2)\alpha^2 + 6(t+1))$. Thus $E[\beta]$ must contain the coefficients of these two polynomials. If $E[\beta]$ is the smallest subfield of $E[\alpha]$ for which $\hat{f}(t, Z)$ has such a factorization, then it must be generated by the coefficients of these two polynomials. In this case we can assume that $\beta = \alpha^2$, whose minimal polynomial is

$$\hat{h}(t, Z) = (t+2)Z^2 + (6t+6)Z + 7t + 5. \tag{3}$$

To convert $E[\beta]$ back to the form $k(f(x))$ we observe that the elements of $E[\beta]$ are rational functions in x over k by Lũroth's theorem. When t is replaced by $f(x)$, (3) must have linear factors, *viz.*

$$\hat{h}(f(x), Z) = (Z - x^2)\left(Z - \frac{3x^2+4}{2x^2+3}\right),$$

which leads to the intermediate fields $k(x^2)$ and $k((3x^2 + 4)/(2x^2 + 3))$. These two fields are isomorphic by the fractional linear map $x \mapsto (3x+4)/(2x+3)$. Using the $k(x^2)$ as the intermediate field, we have $h(x) = x^2$, and thus the irreducible decomposition:

$$-\frac{2x^4+6x^2+5}{x^4+6x^2+7} = -\frac{2x^2+6x+5}{x^2+6x+7} \circ x^2.$$

The original decomposition is equivalent to this one since

$$\frac{x^2+1}{x^2+2} = \left(\frac{x+1}{x+2}\right) \circ x^2$$
$$\frac{x^2+1}{x^2-2} = \left(-\frac{2x^2+6x+5}{x^2+6x+7}\right) \circ \left(\frac{-2x+1}{x-1}\right)$$

This basic approach is applicable to the general problem except for deciding which factors of $\hat{f}(t, Z)$ should be recombined to generate a factorization over a subfield of $E[\alpha]$. We could try all possible combinations of factors of $\hat{f}(t, Z)$ until we find one that yields a proper subfield of $E[\alpha]$. However, in the worst case this would require an exponential number of trials. Instead, we use a version of Landau and Miller's algorithm **BLOCKS** in [14] to find a non-trivial block, which will generate a proper subfield of $E[\alpha]$. As pointed out by Kozen and Landau [11], this algorithm only requires that the extension $E[\alpha]/E$ be separable. Kozen and Landau may need to examine as many as $O(n^{\log n})$ non-trivial blocks to find a decomposition. However, in our case, any non-trivial block will give a rational function decomposition. These techniques allow us to decide which factors of $\hat{f}(t, Z)$ should be recombined in polynomial time.

Furthermore, observe that Trager's polynomial time reduction of factorization over algebraic extensions [19], which was used by Landau to show that factoring over algebraic number fields is polynomial time [13] is applicable here also, so the factorization of $\hat{f}(t, Z)$ over the function field $E[\alpha]$ can be done in polynomial time.

The coefficients of such a factorization generate the intermediate field $E[\beta]$. Since we are seeking any intermediate field, a single coefficient that is not in E suffices. The minimal polynomial of for that coefficient can be determined using resultants and square free decompositions to give $E[\beta]/(p_\beta(t, \beta))$. $h(x)$ is then deduced from a linear factor of $p_\beta(f(x), Z)$, which need only be factored over k. (Factoring bivariate polynomials is polynomial time by Kaltofen [9].)

It is worth commenting on the practicality of this algorithm. Its dominant cost is the factorization of $\hat{f}(t, Z)$ over $k(t)[\alpha]$, which is about as costly as factoring a polynomial of degree $(\deg f)^2$. Given the practical difficulties of factoring polynomials of degree greater than about 100, it seems that it will be very difficult to determine the decomposition of $f(x)$ if the degree of $f(x)$ is greater than about 10.

3.3 Characteristic p case

Determining any decomposition, as opposed to determining a decomposition with a particular degree pattern over a field of characteristic p is only slightly more difficult than the characteristic 0 case, using the technique of Section 3.2. Assume that $\operatorname{char} k = p$ and $f(x)$ is a rational function over k. The decomposition of $f(x)$ may no longer be unique, but Proposition 4 shows that there is still a one to one correspondence between the inequivalent decompositions of $f(x)$ and the fields intermediate between $k(x)$ and $k(f(x))$.

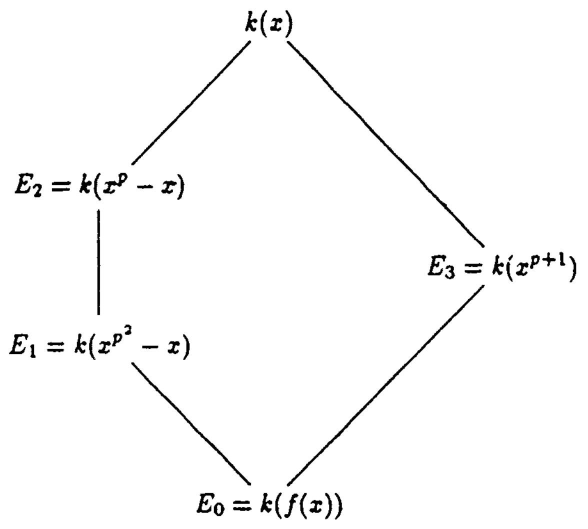


Figure 3: Field Structure for $f(x) = x^{p^3+p^2} - x^{p^3+1} - x^{p^2+p} + x^{p+1}$

Referring to Figure 2, let $\hat{f}(t, Z)$ be the (irreducible) minimal polynomial of α over E. If $\hat{f}(t, Z)$ is separable, then $E[\alpha]$ is separable over E and a subfield can be computed using the techniques of the previous section. If $\hat{f}(t, Z)$ is inseparable then it can be written as

$$\hat{f}(t, Z) = \bar{f}(t, Z^{p^\mu}),$$

for some positive value of μ. Furthermore, $\bar{f}$ is separable over E. Clearly, the field $E[\alpha^{p^\mu}]$ lies between $E[\alpha]$ and E and thus a linear factor of $\bar{f}(f(x), Z)$ will give a decomposition factor of $f(x)$. Since $E[\alpha^{p^\mu}]$ is separable over E, the techniques of the previous section can be used to find additional right decomposition factors. Left decompositions factors can be found from the fields $E[\alpha^{p^i}]$, which lie between $E[\alpha]$ and $E[\alpha^{p^\mu}]$ for $1 \leq i < \mu$.

It is worth noting that even the pathological example suggested by Dorey and Whaples [6]

$$\begin{aligned} f(x) &= x^{p+1} \circ (x^p + x) \circ (x^p - x), \\ &= (x^{p^2} - x^{p^2-p+1} - x^p + x) \circ x^{p+1}, \\ &= x^{p^3+p^2} - x^{p^3+1} - x^{p^2+p} + x^{p+1}, \end{aligned}$$

is can be handled straightforwardly, since the derived polynomial

$$\hat{f}(t, Z) = Z^{p^3+p^2} - Z^{p^2+p} - Z^{p^3+1} + Z^{p+1} - t$$

is separable. The fields associated with the two decompositions of $f(x)$ are shown in Figure 3.

In the case of polynomial decomposition, notice that $\hat{f}(t, Z)$ is inseparable if and only if $f(x)$ is a rational function of x^p. Thus the distinction made by von zur Gathen [21, 22] between "tame" and "wild" might more appropriately be made on whether or not $f(x)$ is a rational function in x^p.

Note that this approach only finds *some* decomposition of $f(x)$. It cannot find a prescribed one. In particular, if one is looking for a decomposition $f(x) = g(h(x))$ where $p | \deg g$ then the extension $k(x)/k(f(x))$ may be inseparable and we would thus have no algorithm for finding intermediate fields. This problem is raised in [22].

4 Conclusions

The technique is used to find the $h(x)$ in Section 3.2 is reminiscent of the technique proposed by Kozen and Landau [11] for decomposition over arbitrary fields. However, they studied intermediate fields between $k(\alpha)/(f(\alpha))$ and k. While there is an intermediate field between $k(\alpha)$ and k whenever $f(x)$ is decomposable, the existence of an intermediate field does not guarantee a decomposition. By using intermediate fields between fields $k(t)[\alpha]/(f(\alpha)-t)$ and $k(t)$, we avoid much of the complexity of their approach since any such intermediate field does lead to decomposition of $f(x)$.

It is tempting to conjecture that Propositions 5 and 6 can be generalized to more variables, but the straightforward generalization is not true, as pointed out in Section 2. It would be interesting to know in what way it can be generalized.

This work has benefited from discussions with Barry Trager and Dexter Kozen. Susan Landau's comments on an earlier version of this paper where quite helpful. The diagrams in this paper were typeset using Paul Taylor's commutative diagram macros for LaTeX.

References

[1] V. S. Alagar and M. Thanh. "Fast decomposition algorithms". In B. F. Caviness, editor, *Proceedings of Eurocal '85, Vol. II*, volume 204 of *Lecture Notes in Computer Science*, pages 150–153, Berlin-Heidelberg-New York, 1985. Springer-Verlag.

[2] D. R. Barton and R. E. Zippel. "Polynomial decomposition". In Jenks [8].

[3] ——. "Polynomial decomposition algorithms". *Journal of Symbolic Computation*, 1(2):159–168, June 1985.

[4] M. Dickerson. *The Functional Decomposition of Polynomials.* PhD thesis, Cornell University, Ithaca, NY, August 1989.

[5] ——. "The inverse of an automorphism in polynomial time". In *30th Symposium on Foundations of Computer Science*, pages 82–87. ACM, 1989.

[6] F. Dorey and G. Whaples. "Prime and composite polynomials". *Journal of Algebra*, 28:88–101, 1974.

[7] J. Gutiérrez, T. Recio, and C. Ruiz de Velasco. "Polynomial decomposition algorithm of almost quadratic complexity". In *Proceedings of AAECC-6, 1988.* Springer-Verlag, 1989.

[8] R. Jenks, editor. *Symbolic and Algebraic Computation '76*, New York, August 1976. ACM.

[9] E. Kaltofen. "Polynomial-time reductions from multivariate to bi- and univariate integral polynomial factorizations". *SIAM Journal of Computing*, 14:469–489, 1985.

[10] D. E. Knuth. *The Art of Computer Programming*, volume II. Addison-Wesley, New York, NY, 1971.

[11] D. Kozen and S. Landau. "Polynomial decomposition algorithms". *Journal of Symbolic Computation*, 7:445–456, 1989.

[12] H. T. Kung and J. F. Traub. "All algebraic functions can be computed fast". *Journal of the ACM*, 1978.

[13] S. Landau. "Factoring polynomials over algebraic number fields". *SIAM Journal of Computing*, 14(1):184–195, 1985.

[14] S. Landau and G. L. Miller. "Solvability by radicals is in polynomial time". *Journal of Computer and System Sciences*, 30(2):179–208, April 1985.

[15] P. Lüroth. "Beweis eines Satzes über rationale Curven". *Mathematische Annalen*, 9:163-165, 1876.

[16] R. J. McEliece and J. B. Shearer. "A property of Euclid's algorithm and an application to Padé approximation". *SIAM Journal of Applied Mathematics*, 34:611–615, 1978.

[17] A. Schinzel. *Selected Topics on Polynomials.* University of Michigan Press, Ann Arbor, MI, 1982.

[18] E. Steinitz. "Algebraische Theorie der Körper". *Journal für reine und angewante Mathematik*, 137:167–309, 1910.

[19] B. M. Trager. "Algebraic factoring and rational function integration". In Jenks [8], pages 219–226.

[20] B. L. van der Waerden. *Modern Algebra.* Fredrick Ungar, New York, NY, 1964.

[21] J. von zur Gathen. "Functional decomposition of polynomials: the tame case". *Journal of Symbolic Computation*, 9(3):281–299, March 1990.

[22] ——. "Functional decomposition of polynomials: the wild case". *Journal of Symbolic Computation*, 10(5):437–452, November 1990.

[23] J. von zur Gathen, D. Kozen, and S. Landau. "Functional decomposition of polynomials". In *28th Symposium on Foundations of Computer Science*, pages 127–131. ACM, 1987.

[24] H. Weber. *Lehrbuch der Algebra*, volume II. Chelsea Publishing Co., New York, third edition, 1961.

Automatic surface generation using implicit cubics

Baining Guo

Abstract

Modeling physical objects with low-degree algebraic surfaces shows promise for applications where manipulating and reasoning about physical objects are important. In this paper, we present an algorithm for free-form surface constructions using implicitly defined cubic surface patches. The input data for the algorithm is an arbitrary polyhedron with a normal prescribed at each vertex of the polyhedron. Using a Clough-Tocher like splitting scheme, the algorithm constructs a smooth piecewise cubic surface interpolating the vertices of the polyhedron and the prescribed normal at each vertex. The free-form surface construction in the algorithm is *local* and *quadratically precise.* In addition, the shape of the free-form surfaces can be manipulated through a set of intuitive *shape parameters* without knowing the details of the algorithm. The implementation results are reported.

Keywords: Geometric modeling, object representation, free-form surface, Bernstein-Bezier representation, implicit patch, design.

1 Introduction

While developing a geometric modeling system for representing, manipulating and reasoning about physical objects, we derived and implemented an algorithm for constructing geometric models for smooth objects of arbitrary shapes and topologies. Such geometric models are important for solid modeling, computer-aided design, visualization, computer graphics, and robotics.

The geometric models of arbitrary smooth objects are represented by closed free-form surfaces. The algorithm we drive generates a free-form surface from the input data of an arbitrary polyhedron with a normal prescribed at each vertex of the polyhedron. Using a Clough-Tocher like splitting scheme, the algorithm constructs a smooth piecewise cubic surface interpolating the vertices of the polyhedron and the prescribed normals.

The algorithm we derive has the following features. First, the algorithm is *local,* so modifying a piece of input data affects only nearby surface patches. Second, the algorithm has *quadratic precision,* which means that if the input data is taken from a quadric surface, the algorithm reproduces the quadric surface. Finally, the shape of the free-form surfaces produced by the algorithm can be controlled through a set of intuitive *shape parameters* without knowing the details of the algorithm.

An important motivation of our work is to construct geometric models that facilitate manipulating and reasoning about physical objects (Hopcroft and Krafft 1986; Hoffmann 1989). Traditionally, the building blocks for free-form surface constructions are parametric patches. As far as design and display are concerned, parametric patches are very successful. But when it comes to manipulating and reasoning about physical objects, parametric patches run into serious problems. Parametric patches are not closed under some elementary operations in geometric modeling, such as sweeping and Minkowski sum (Bajaj and Kim 1987). The intersection of two parametric patches is extremely difficult to represent and evaluate (Hoffmann 1989) because the algebraic degree of the intersection is prohibitively high. As an example, we notice that in general the intersection of two commonly used bicubic patches is a space curve of degree 324 (Hopcroft and Krafft 1986).

These problems can be avoided by building free-form surfaces from low-degree implicit patches. Implicit patches are closed under all common operations required by a geometric modeling system (Bajaj 1989), and the intersection of two degree n implicit patches has degree n^2, which is small if n is. Low-degree implicit patches also allow the use of algebraic techniques as opposed to numerical techniques in reasoning about physical objects (Hopcroft and Krafft 1986). These features make implicit patches a superior choice for applications where manipulating and reasoning about physical objects are important. In addition, from a practical point of view, implicit patches are compact to store and relatively easy to ray trace.

An inviting class of implicit patches for free-form surface constructions is the class of quadric patches. When the input data is a polyhedron without normals prescribed at its vertices, a free-form surface can be constructed using quadric patches. However, quadric patches have fundamental limitations that make it impossible to allow prescribing normals in the input data. Roughly speaking, when a free-form surface is constructed by replacing the facets of the input polyhedron with quadric patches, they introduce a correlation between the normals at the adjacent vertices of the input polyhedron. We have investigated the role of quadric patches as primitives for free-form surface constructions, and we hope to report the results elsewhere.

Being able to prescribe the normals in the input data is important. Prescribing normals is a measure to control the patches in the free-form surfaces so that only a few patches are needed for representing a smooth object that would otherwise requires thousands of polygons to approximate. One way to overcome the limitations of quadric patches is to split the edges of the input polyhedron, as was done by Dahmen (Dahmen 1989). However, from a theoretical point of view, Dahmen's method cannot handle arbitrary input polyhedron because his method requires the existence of "transversal systems", which no one knows how to construct in general; from a practical point of view, splitting the edge of the input polyhedron causes oscillations in the free-form surfaces, making it impossible to produce free-form surfaces of pleasing shapes. In this paper, we show that the limitations of quadric patches can be overcome by cubic patches.

1.1 Previous work

There is a rich literature on surface constructions using parametric patches, and a recent survey can be found in (Mann et al. 1990). Modeling complex objects with implicit patches was introduced in recent years and is becoming an increasingly prominent area of research. General techniques for implicit modeling are developed by researchers worldwide (Nishimura et al. 1985; Bloomenthal and Wyvill 1990; Dahmen 1989). In particular, many authors have demonstrated the power of implicit patches in deriving blending surfaces (Blinn 1982; Middleditch and Sears 1985; Hoffmann and Hopcroft 1987; Rockwood and Owen 1987) and in surface fitting and approximation (Bajaj and Ihm 1989; Patrikalakis and Kriezis 1989).

Sederberg proposed using Bernstein-Bezier representation of implicit patches in free-form surface constructions (Sederberg 1985). Subsequently, various techniques are developed for constructing free-form surfaces using implicit patches (Patrikalakis and Kriezis 1989; Bajaj and Ihm 1989; Moore and Warren 1990; Sederberg 1990). In particular, Patrikalakis et al., Sederberg, and Bajaj et al. demonstrated the complications and pitfalls of modeling with implicit patches (Patrikalakis and Kriezis 1989; Bajaj and Ihm 1989; Sederberg 1990).

Dahmen (Dahmen 1989) gave an algorithm for constructing free-form surfaces from quadric patches. But the algorithm cannot handle arbitrary polyhedra, and the splitting scheme in the algorithm prevents it producing pleasing shapes. There are also algorithms for constructing free-form surfaces with implicit patches of degree six and degree five (Moore and Warren 1990; Bajaj 1990).

2 Conceptual overview

The algorithm described in this paper builds free-form surfaces from the input data of a polyhedron with a normal vector prescribed at each vertex of the polyhedron. The input data is denoted by $(\mathcal{P}, \mathcal{N})$, where $\mathcal{P}$ is an arbitrary polyhedron with vertex set $\{\mathbf{x}_1, \cdots, \mathbf{x}_k\}$, and $\mathcal{N}$ is a set of normals $\{\mathbf{n}_1, \cdots, \mathbf{n}_k\}$ with $\mathbf{n}_i$ being the normal vector prescribed at $\mathbf{x}_i$. The facets of $\mathcal{P}$ are assumed to be triangular.

The basic idea of the algorithm is very simple. The free-form surface to be built must be in the neighborhood of the input polyhedron $\mathcal{P}$, so we construct a neighborhood Σ of $\mathcal{P}$ using tetrahedra and creat a cubic polynomial for each tetrahedron used. By ensuring C^1 conditions between adjacent tetrahedra, we obtain a global C^1 function that is a cubic polynomial in each tetrahedron. The zero contour of this global C^1 function within the neighborhood Σ is the free-form surface to be generated.

The following three aspects are crucial to the success of the algorithm.

1. The construction of a neighborhood Σ of the input polyhedron using tetrahedra. The neighborhood must locally contain the tangent plane determined by the prescribed normal at each vertex of the input polyhedron $\mathcal{P}$, and the neighborhood must have the same topology as the polyhedron $\mathcal{P}$.

2. A scheme for defining a globally C^1 function which is a cubic polynomial over each tetrahedron within the neighborhood Σ. The scheme must leave *free control points* in the definition of each cubic polynomial so that the zero contour of the cubic polynomial can be controlled by these free control points.

3. A mechanism to control the cubic polynomial defined for each tetrahedron so that the zero contour of the cubic polynomial inside the tetrahedron is a single-sheeted cubic patch without holes, extraneous sheets, self intersections, or other topological anomalies.

These three aspects will be stressed throughout the development of the algorithm.

3 Algorithm details

Now we address the three aspects of the algorithm in detail. In this paper, we use $[\mathbf{x}_1 \cdots \mathbf{x}_n]$ to denote the convex hull of point set $\{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$.

3.1 The construction of the neighborhood Σ

The basic spatial elements used to build the neighborhood Σ of the polyhedron $\mathcal{P}$ are tetrahedra. Tetrahedra are chosen for two reasons. one, tetrahedra are simple and flexible three dimensional

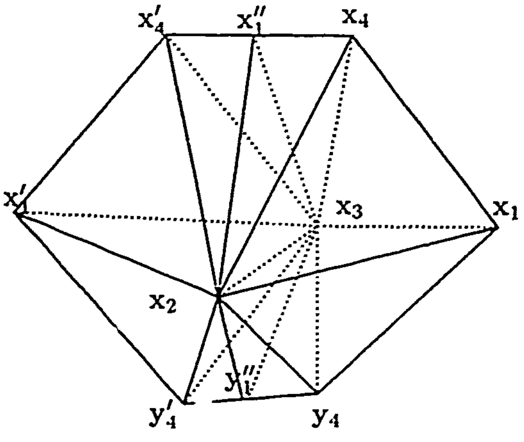


Figure 1: Filling gaps between two double tetrahedra

spa? units; two, tetrahedra facilitate the use of Bernstein-Bezier representation, which is the base for this work.

The neighborhood Σ is constructed as follows. For each facet $F = [x_1x_2x_3]$ of the polyhedron $\mathcal{P}$, two _ ints x_4 and y_4 off each side of the facet are chosen, and they determine two tetrahedra, $[x_1x_2x_3x_4]$ and $[x_1x_2x_3y_4]$. These two tetrahedra form a *double tetrahedron* denoted by $([x_1x_2x_3x_4], [x_1x_2x_3y_4])$. Consider an adjacent facet $F' = [x_1'x_2x_3]$ and its double tetrahedron $([x_1'x_2x_3x_4'], [x_1'x_2x_3y_4'])$. Between the double tetrahedra of facets F and F', there are two gaps. One gap is between the tetrahedra $[x_1'x_2x_3x_4']$ and $[x_1x_2x_3x_4]$; the other is between $[x_1x_2x_3y_4]$ and $[x_1'x_2x_3y_4']$. The first gap is filled with a pair of tetrahedra $[x_1''x_2x_3x_4]$ and $[x_1''x_2x_3x_4']$, and the second gap is filled with another pair of tetrahedra. $[y_1''x_2x_3y_4]$ and $[y_1''x_2x_3y_4']$. Here x_1'' and y_1'' are points on the line segments $[x_4x_4']$ and $[y_4y_4']$ respectively. All these are shown in Figure 1.

As an auxiliary geometric structure for the free-form surface construction, the neighborhood Σ must satisfy the following condition. At each vertex x_i of the polyhedron $\mathcal{P}$, the neighborhood Σ should locally contain the tangent plane defined by n_i. In other words, there is a disk D around the vertex x_i in the tangent plane at x_i such that

$$D \subset \Sigma.$$

3.2 A scheme for enforcing C^1 conditions over Σ

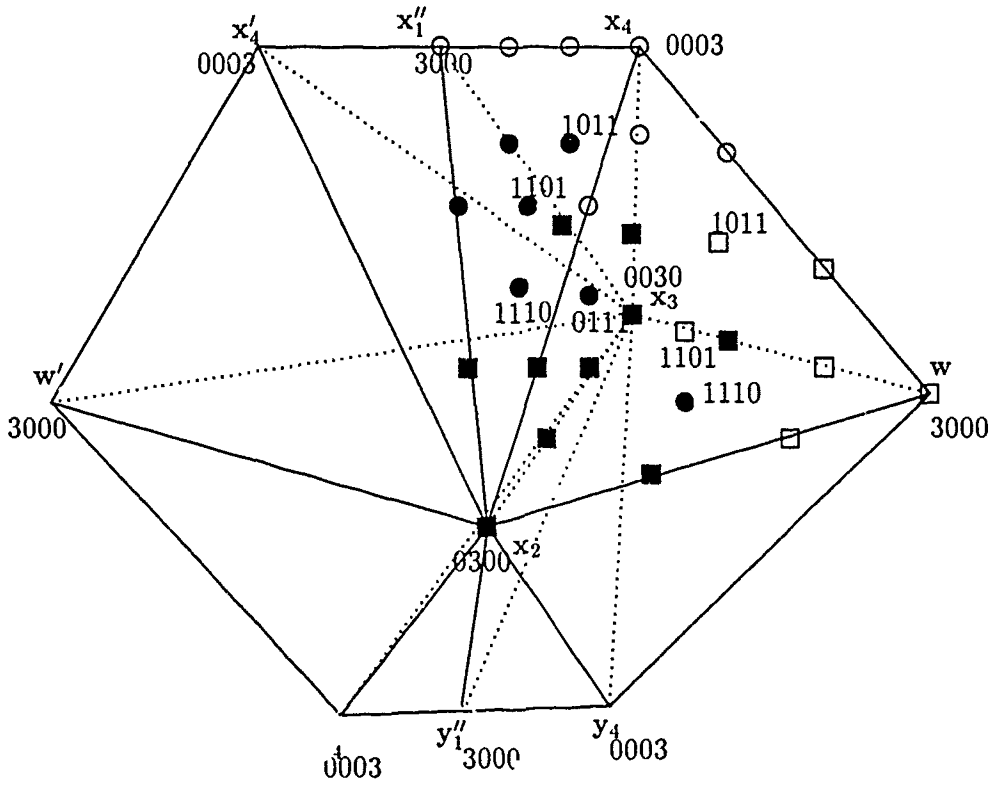
Having built a neighborhood Σ of the polyhedron $\mathcal{P}$, we construct a C^1 function f over the neighborhood Σ so that

$$f(x_i) = 0, \ \nabla f(x_i) = n_i, \ i = 1, \cdots, k. \tag{1}$$

The zero contour of f within Σ is the free-form surface to be generated.

The construction of f can be outlined as follows. First, we split the the tetrahedra that have facets of $\mathcal{P}$ as faces: the neighborhood Σ is kept the same except some of its tetrahedra are split. Then, the function f is defined by constructing a cubic polynomial for each tetrahedron within the neighborhood Σ.

To show the splitting scheme, we take a facet $[x_1x_2x_3]$ and its double tetrahedron $([x_1x_2x_3x_4]$, $[x_1x_2x_3y_4])$ as an example. Let w be a point in the facet $[x_1x_2x_3]$. We split the double tetrahedron



□ control points to be decided

○ control points that are free or are decided by free control point.

■ control points from the input data

● control points from C^1 conditions

Figure 2: The C^1 conditions between two adjacent double tetrahedra

into six tetrahedra: $[x_ix_jwx_4]$ and $[x_ix_jwy_4]$ for $1 \le i < j \le 3$. For symmetry and robustness reasons, w is often chosen to be the centroid of triangle $[x_1x_2x_3]$, while y_4 and x_4 are chosen to be on the line that passes through w and is perpendicular to $[x_1x_2x_3]$.

The construction of cubic polynomials over the tetrahedra within Σ takes two steps. Consider a facet $F' = [x_1'x_2x_3]$ adjacent to facet $F = [x_1x_2x_3]$ and the double tetrahedron of F', $([x_1'x_2x_3x_4'],$ $[x_1'x_2x_3y_4'])$. The facet F' and its double tetrahedron are split at the centroid w' of F' in the same way that $([x_1x_2x_3x_4],[x_1x_2x_3y_4])$ is split at w. For the facets F and F', the first step takes place over tetrahedra $V_1 = [x_2x_3x_4w]$, $V_2 = [x_2x_3x_4'w']$, $W_1 = [x_2x_3x_1''x_4]$, $W_2 = [x_2x_3x_1''x_4']$, $V_1' = [x_2x_3y_4w]$, $V_2' = [x_2x_3y_4'w']$, $W_1' = [x_2x_3y_1''y_4]$, and $W_2' = [x_2x_3y_1''y_4']$ as in Figure 2. We construct the cubic polynomials over tetrahedra W_1, W_2, W_1', and W_2'. At the same time, the cubic polynomials over tetrahedra V_1, V_2, V_1', and V_2' are partially determined through C^1 conditions. The same process is carried out between every pair of adjacent facets of $\mathcal{P}$, so at the end of the first step, the cubic polynomials over the tetrahedra $[x_ix_jx_4w]$ and $[x_ix_jy_4w]$ are partially constructed for all $i < j \le 3$. Then, the second step completes the construction of these cubic polynomials according to C^1 conditions.

Now we describe the first step in detail. Throughout this description, we assume $i = 1,2$ whenever i appears. By doing so we are taking advantage of the symmetry of the problem in consideration.

Let the cubic polynomials f_i over V_i, f_i' over V_i', g_i over W_i, and g_i' over W_i' be expressed in Bernstein-Bezier forms as follows.

$$f_i(x) = \sum_{|\lambda|=3} a_\lambda^i B_\lambda^3(\tau_i), \tag{2}$$

$$g_i(x) = \sum_{|\lambda|=3} b_\lambda^i B_\lambda^3(\rho_i), \tag{3}$$

$$f_i'(x) = \sum_{|\lambda|=3} c_\lambda^i B_\lambda^3(\tau_i'), \tag{4}$$

and

$$g_i'(x) = \sum_{|\lambda|=3} d_\lambda^i B_\lambda^3(\rho_i'), \tag{5}$$

where τ_i, τ_i', ρ_i' and ρ_i are the barycentric coordinates on V_i, V_i', W_i', and W_i respectively. We call the a_λ's, b_λ's, c_λ's, and d_λ's the *control points* of the cubic polynomials f_i, f_i', g_i, and g_i' respectively. Our task is to determine these control points.

For notational convenience, if two tetrahedra sharing a common face, we equal the control points of the associated cubic polynomials on the common face to ensure C^0 continuity. Hence such control points will be defined only once.

All the control points over tetrahedra V_i, V_i', W_i, and W_i' that can be determined from the input data are as follows. The fact that the zero contours of f_i, f_i', g_i, and g_i' pass through x_2 and x_3 implies

$$a_{0300}^i = a_{0030}^i = 0,$$

$$c_{0300}^i = c_{0030}^i = 0,$$

$$b_{0300}^i = b_{0030}^i = 0,$$

and

$$d_{0300}^i = d_{0030}^i = 0.$$

More control points are determined by the normals at the vertices x_2 and x_3. For example,

$$a_{2e^j+e^1}^1 = \frac{1}{3}(\mathrm{n}_j, \mathrm{w} - \mathrm{x}_j),\ j = 2,3.$$

Similar expressions are used to determine the control points $a_{2e^j+e^k}^i$ for $j = 2,3$ and $k = 1,4$, $c_{e^j+e^4}^i$ for $j = 2,3$, $b_{2e^j+e^1}^i$ for $j = 2,3$, and $d_{2e^j+e^1}^i$ for $j = 2,3$.

Before determining the rest of the control points according to C^1 conditions, we have to choose some control points to be *free control points* whose values will be left unspecified at this point. This is because creating a piecewise cubic C^1 function f over the neighborhood Σ is only an intermediate step in the free-form surface construction. Having defined a cubic polynomial whose zero contour passing through some vertices of a tetrahedron does not guarantee the existence of a taut cubic

Figure 3: A handle on a cubic patch

patch inside the tetrahedron. There may not be a cubic patch inside the tetrahedron at all, or even there is, the cubic patch can have self intersections, holes, and extra sheets. Figure 3 is a more dramatical example: a handle appears on an otherwise nice cubic patch. If this cubic patch is in a free-form surface constructed from the input polyhedron $\mathcal{P}$, the topology of the free-form surface is bound to be different from that of $\mathcal{P}$.

We choose the following free control points $a^i_{2e^4+e^j}$ $(j = 1, 2, 3, 4)$, $c^i_{2e^4+e^j}$ $(j = 1, 2, 3, 4)$, b^i_{2001}, and d^i_{2001} for f_i, f'_i, g_i, and g'_i respectively. The intuition of these control points are as follows. Control points $a^i_{2e^4+e^j}$ $(j = 1, 2, 3)$ are equivalent to the function values and gradients at x_4 and x'_4, and the control point b^i_{2001} enables us to have complete control of the function values of g_i along the line segment $[x_4 x'_4]$. The same statement can be made about control points $c^i_{2e^4+e^j}$ $(j = 1, 2, 3, 4)$ and d^i_{2001}. In 3.3, we will explain how these free control points affects the associated cubic patches.

Now we determine the rest of the control points to ensure C^1 conditions. Consider the C^1 conditions across faces $[x_2 x_3 x_4]$ and $[x_2 x_3 x'_4]$. Suppose

$$x''_1 = \beta^1_1 x_1 + \beta^1_2 x_2 + \beta^1_3 x_3 + \beta^1_4 x_4$$

and

$$\mathrm{x}_1'' = \beta_1^2 \mathrm{x}_1' + \beta_2^2 \mathrm{x}_2 + \beta_3^2 \mathrm{x}_3 + \beta_4^2 \mathrm{x}_4'.$$

Then, the C^1 conditions are the following.

$$b^i_{1002} = \beta^i_1 a^i_{1002} + \beta^i_2 a^i_{0102} + \beta^i_3 a^i_{0012} + \beta^i_4 a^i_{0003},$$

$$b^i_{1101} = \beta^i_1 a^i_{1101} + \beta^i_2 a^i_{0201} + \beta^i_3 a^i_{0111} + \beta^i_4 a^i_{0102}, \tag{6}$$

$$b^i_{1011} = \beta^i_1 a^i_{1011} + \beta^i_2 a^i_{0111} + \beta^i_3 a^i_{0021} + \beta^i_4 a^i_{0012}, \tag{7}$$

and

$$b^i_{1110} = \beta^i_1 a^i_{1110} + \beta^i_2 a^i_{0210} + \beta^i_3 a^i_{0120} + \beta^i_4 a^i_{0111}. \tag{8}$$

The first three equations can be viewed as the definitions for the control points b^i_{1101}, b^i_{1011}, and b^i_{1110}, leaving a^i_{1011} and a^i_{1101} to be determined. Equation (8) will be treated later.

Moving on to the C^1 conditions across $[\mathrm{x}_2\mathrm{x}_3\mathrm{x}_1'']$, we see that if

$$\mathrm{x}_1'' = \mu_1 \mathrm{x}_4 + \mu_2 \mathrm{x}_4',$$

then the C^1 conditions are the following.

$$b^i_{3000} = \mu_1 b^1_{2001} + \mu_2 b^2_{2001}, \tag{9}$$

$$b^i_{1020} = \mu_1 b^1_{1101} + \mu_2 b^2_{1101}, \tag{10}$$

$$b^i_{1200} = \mu_1 b^1_{1011} + \mu_2 b^2_{1011}, \tag{11}$$

and

$$b^i_{1110} = \mu_1 a^1_{0111} + \mu_2 a^2_{0111} \tag{12}$$

Again, the first three equations can be viewed as definitions for control points b^i_{3000}, b^i_{1020}, and b^i_{1200}; and the last equation will be treated later. Notice that b^i_{1011} and b^i_{1101} in the above equations are defined earlier by the equations (6) and (7).

Finally, we consider the C^1 conditions across faces $[\mathrm{x}_2\mathrm{x}_3\mathrm{y}_4]$, $[\mathrm{x}_2\mathrm{x}_3\mathrm{y}_1'']$, and $[\mathrm{x}_2\mathrm{x}_3\mathrm{y}_4']$. All the control points of g_i' and some of the control points of f_i' can be fixed in the same way as the control points of f_i and g_i. In doing so, we also have two equations left untreated.

$$d^i_{1110} = \gamma^i_1 a^i_{1110} + \gamma^i_2 a^i_{0210} + \gamma^i_3 a^i_{0120} + \gamma^i_4 c^i_{0111} \tag{13}$$

and

$$d^i_{1110} = \eta_1 c^1_{0111} + \eta_2 c^2_{0111}, \tag{14}$$

where the coefficients η's and γ's come from the following relations:

$$\mathrm{y}_1'' = \gamma_1^2 \mathrm{x}_1' + \gamma_2^2 \mathrm{x}_2 + \gamma_3^2 \mathrm{x}_3 + \gamma_4^2 \mathrm{y}_4'$$

and

$$\mathrm{y}_1'' = \mu_1 \mathrm{y}_4 + \mu_2 \mathrm{y}_4'.$$

Now we collectively treat the equations (8), (12), (13), and (14) as promised. These equations can be rewritten as

$$\eta_1 c^1_{0111} + \eta_2 c^2_{0111} = \gamma^i_1 a^i_{1110} + \gamma^i_2 a^i_{0210} + \gamma^i_3 a^i_{0120} + \gamma^i_4 c^i_{0111} \tag{15}$$

and

$$\mu_1 a^1_{0111} + \mu_2 a^2_{0111} = \beta^i_1 a^i_{1110} + \beta^i_2 a^i_{0210} + \beta^i_3 a^i_{0120} + \beta^i_4 a^i_{0111}. \tag{16}$$

Here c^i_{0111} can be determined from a^i's through the C^1 conditions across $[\mathrm{x}_1\mathrm{x}_2\mathrm{x}_3]$ and $[\mathrm{x}'_1\mathrm{x}_2\mathrm{x}_3]$,

$$c^i_{0111} = \alpha^i_1 a^i_{1110} + \alpha^i_2 a^i_{0210} + \alpha^i_3 a^i_{0120} + \alpha^i_4 a^i_{0111}, \tag{17}$$

where the α's come from

$$\mathrm{y}_4 = \alpha^1_1\mathrm{x}_1 + \alpha^1_2\mathrm{x}_2 + \alpha^1_3\mathrm{x}_3 + \alpha^1_4\mathrm{x}_4$$

and

$$\mathrm{y}'_4 = \alpha^2_1\mathrm{x}'_1 + \alpha^2_2\mathrm{x}_2 + \alpha^2_3\mathrm{x}_3 + \alpha^2_4\mathrm{x}'_4.$$

The equations (15), (16), and (17) form a system of six linear equations with six unknowns, a^i_{0111}, a^i_{1110}, and c^i_{0111}. When the points x_4, x'_4, y_4, and y'_4 are in general position, the system always has a solution. It may happen that there is a family of solutions, in which case we choose a solution as follows. Using the degree-elevation property of Bernstein-Bezier representation, we can compute default values for a^i_{1110} and c^i_{0111} from the prescribed normals. A solution to the system can be selected from the family of solutions according to these default values.

This finishes the first step in the construction of the cubic polynomials over the tetrahedra within the neighborhood Σ. Now g_i and g'_i are completely constructed, while f_i and f'_i are partially constructed.

The second step completes the construction of f_i and f'_i. For this purpose, we shift our focus to the double tetrahedron ($[\mathrm{x}_1\mathrm{x}_2\mathrm{x}_3\mathrm{x}_4]$, $[\mathrm{x}_1\mathrm{x}_2\mathrm{x}_3\mathrm{y}_4]$), which has been split into tetrahedra $[\mathrm{x}_1\mathrm{x}_2\mathrm{x}_4\mathrm{w}]$, $[\mathrm{x}_1\mathrm{x}_3\mathrm{x}_4\mathrm{w}]$, $[\mathrm{x}_3\mathrm{x}_2\mathrm{x}_4\mathrm{w}]$, $[\mathrm{x}_3\mathrm{x}_2\mathrm{y}_4\mathrm{w}]$, $[\mathrm{x}_3\mathrm{x}_2\mathrm{y}_4\mathrm{w}]$, and $[\mathrm{x}_3\mathrm{x}_2\mathrm{y}_4\mathrm{w}]$.

Consider the problem of completing the construction of the partially constructed cubic polynomials for tetrahedra $U_1 = [\mathrm{x}_2\mathrm{x}_3\mathrm{x}_4\mathrm{w}]$, $U_2 = [\mathrm{x}_1\mathrm{x}_3\mathrm{x}_4\mathrm{w}]$, and $U_3 = [\mathrm{x}_1\mathrm{x}_2\mathrm{x}_4\mathrm{w}]$. Denote the cubic polynomials for U_i by

$$h_i(\nu_i) = \sum_{|\lambda|=3} a^i_\lambda B^3_\lambda(\nu_i),$$

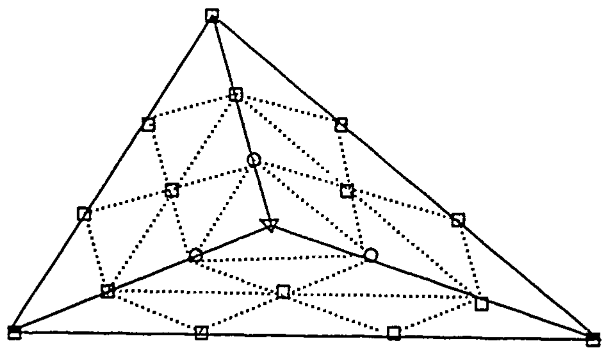
where ν_i is the barycentric coordinate of U_i. It is easy to recognize that polynomial h_1 is the same as f_1 in the first step. More generally, the partially constructed functions h_i is the result of carrying out the first step for $[\mathrm{x}_1\mathrm{x}_2\mathrm{x}_3]$ and the facet sharing edge $[\mathrm{x}_m\mathrm{x}_n]$ ($m \neq i, n \neq i, 1 \leq m, n \leq 3$) with $[\mathrm{x}_1\mathrm{x}_2\mathrm{x}_3]$. We denote f_1, V_1, and τ_1 by h_1, U_1, and ν_1 in the second step to reflect the new symmetry.

The task of ensuring C^1 conditions between U_i's is greatly simplified by taking advantage of the fact that $\mathrm{w} \in [\mathrm{x}_1\mathrm{x}_2\mathrm{x}_3]$. The control points over U_i can be divided into four groups. The i-th group, called i-th *layer*, is the set of $a^i_{\lambda_1\lambda_2\lambda_3\lambda_4}$ such that $\lambda_4 = i$. Because $\mathrm{w} \in [\mathrm{x}_1\mathrm{x}_2\mathrm{x}_3]$, the C^1 conditions between U_i and U_j only involve control points from the same layer. So we can satisfy the C^1 conditions by examining each layer as if we were working on bivariate polynomials.

For the 0-th layer, the control points $a^i_{\lambda_1\lambda_2\lambda_3 0}$ are defined previously for all $\lambda_1 \leq 1$. Determining the rest of the control points in this layer is exactly the famous Clough-Tocher interpolation in finite element analysis. Figure 4 illustrates a standard solution (Farin 1986).

For the 1-th layer, the control points $a^i_{\lambda_1\lambda_2\lambda_3 1}$ are defined earlier for all $\lambda_1 = 0$. Since this layer can be viewed as a bivariate quadratic function, the known control points uniquely determine the rest of the control points within the layer through the C^1 conditions (Farin 1986).

The control points in the 2-th and 3-th layers are trivially determined by the the function value and gradient at x_4.



o centroid of surrounding boxes
□ constructed from previous step
▽ centroid of surrounding circles

Figure ' The Clough-Tocher bivariate splines

To complete the second step, we carry out the same argument for tetrahedra $[x_1x_2y_4w]$, $[x_2x_3y_4w]$, and $[x_1x_3y_4w]$. As for the C^1 conditions across $[x_1x_2x_3]$, notice that these conditions only involve control points from the 0-th layer and the 1-th layers. From equation (17) and the way the control points in the 1-th layer are determined, it is easy to see that the C^1 conditions across $[x_1x_2x_3]$ are indeed satisfied.

Therefore, we have constructed the global C^1 function f satisfying (1). If the free control points are chosen so that a "nice" cubic patch is obtained inside each tetrahedron within the neighborhood Σ, then the zero contour of f inside the neighborhood Σ is the free-form surface to be constructed.

3.3 Obtaining and controlling the cubic patches

As we mentioned earlier, creating a C^1 function f over the neighborhood Σ according to (1) is only an intermediate step. In general, such a function rarely yields the free-form surface we expect. The problem is that some of the control points of a cubic polynomial strongly affect the zero contour of the cubic polynomial inside the associated tetrahedron. If we let these control points be decided by the C^1 conditions, then the zero contour of the cubic polynomial inside the tetrahedron exhibits various behaviors undesirable for free-form surface constructions.

The following situations may occur for the zero contour of a cubic polynomial inside a tetrahedron.

1. There is no zero contour in th interior of the tetrahedron even though the zero contour is known to pass through several vertices of the tetrahedron.
2. There are self-intersection points, or singular points on a cubic patch.
3. There are holes on a cubic patch caused by the zero contour of the cubic polynomial leaving and coming back to the tetrahedron. See the left figure in Figure 5.
4. There are multiple sheets of the zero contour inside the tetrahedron. See the left figure in Figure 6.

Figure 5: Avoiding holes in a cubic patch

Figure 6: Avoiding extra sheets in a cubic patch

5. More dramatically, there may be even handles etc. on a cubic patch. See Figure 3.

Notice that we listed singular points together with self intersection because for implicit patches, singular points appear where self intersections occur.

We use tetrahedra $[x_1x_2wx_4]$ in Figure 2 as an example to explain how the situations listed above can be avoided by controlling the free control points we have chosen. In this example, the free control points are the function value $h_3(x_4)$ and the gradient $\nabla h_3(x_4)$. The same argument with minor modifications applies to the cubic polynomials defined for other tetrahedra.

Situation one can be avoided by properly choosing the function value at x_4. Consider the line segment from the centroid of $[x_1x_2w]$, p, to x_4. If the function value $h_3(x_4)$ is chosen to be have a sign opposite to that of the function value at p, then there must be a point on the line segment $[x_4p]$ where the cubic polynomial is zero. In other words, the zero contour passes through the interior of the tetrahedron $[x_1x_2wx_4]$.

Situations two through five can be avoided by enforcing monotonicity conditions on the cubic polynomial along the direction from w to x_4. A function is monotone in direction α if the directional derivative along α is positive. Let the cubic polynomial in $[x_1x_2x_4w]$ be

$$h_3(\nu_3) = \sum_{|\lambda|=3} a_\lambda B^3_\lambda(\nu_3).$$

A sufficient condition for the cubic polynomial h_3 to be monotone along the direction form w to x_4 within the tetrahedron is that

$$a_{\lambda-e^1+e^4} - a_\lambda \geq 0, \text{ for all } \lambda \text{ with } \lambda_1 \geq 1. \tag{18}$$

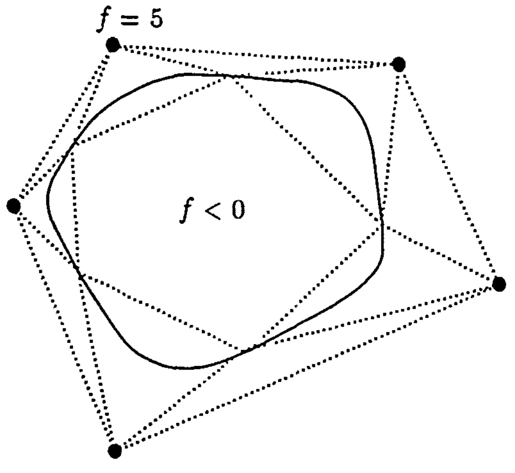


Figure 7: The shape control scheme

When $\lambda_1 > 1$, the condition (18) can be enforced by the function value and gradient at x_4. As for $\lambda_1 = 1$, the control points involved in (18) are completely determined from the prescribed normals in the input data, so the monotonicity conditions may not be satisfied for certain input data no matter how the free control points $h_3(x_4)$ and $\nabla h_3(x_4)$ are chosen. But remember the prescribing normals in the input data is only a measure to control the behavior of each cubic patch. If we choose these normals within proper ranges, the condition (18) can be enforced.

In practice, the free control points are computed using the degree-elevation property of Bernstein-Bezier representation. The idea is to extend the effects of prescribed normals to the free control points. A quadric polynomial q over the tetrahedron $[x_1x_2x_3x_4]$ can be determined from the fact

$$q(x_i) = 0, \ \nabla q(x_i) = n_i, \ i = 1,2,3.$$

and the value $q(x_4)$ which is referred to as a *shape parameter*. If this is done for all facets of the input polyhedron $\mathcal{P}$, then quadric polynomials over tetrahedra such as $[x_2x_3x_1''x_4]$ can be determined also. These quadric polynomials are then degree elevated to cubic polynomials, whose control points corresponding to the free control points are given to the free control points. This method of choosing free control points works very well in practice. From our experience, the ranges of free control points within which the cubic patches behave well are fairly large. As long as the free control points are not in the relative small "bad" ranges, the cubic patches are in good shape.

Figure 5 and Figure 6 are two examples of how the above method works in the setting of Figure 2. In Figure 6, the left figure has an extra sheet due to badly chosen free control points. In the right figure, the badly chosen free control points are corrected using the above method. Figure 5 is similar except the problem is the hole in the left figure.

3.4 Features of the algorithm

The above free-form algorithm has several features. From the description of the algorithm, it is easy to see that the free-form construction in the algorithm is local. In the following, we discuss the

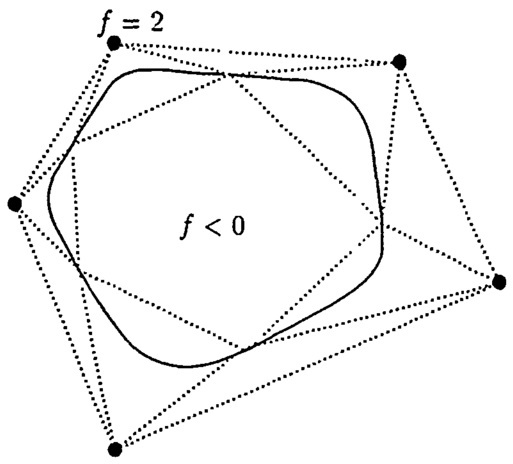


Figure 8: Figure 7 with a shape parameter decreased

quadratic precision of the algorithm, and how to control the shape of the free-form surface without knowing the details of the algorithm.

Quadratic precision is a measure of accuracy of free-form surface algorithms in terms of how well the algorithms can reproduce a known surface if the input data is taken from the surface. A question that users often ask about a free-form surface algorithm is that if the input data is taken from a sphere, can the algorithm reproduces the sphere. For the algorithm we derive, the answer is yes. In fact, the algorithm reproduces all quadrics.

Notice that the input polyhedron $\mathcal{P}$, prescribed normals at the vertices of $\mathcal{P}$, and the shape parameters completely determine the free-form surface. If the input data is taken from a quadric surface and the shape parameters are from the quadric surface, then the algorithm will produce the same quadric surface. To ensure the shape parameters are properly chosen so that all quadric surfaces can be reproduced, we must give certain default values to the shape parameters. For example, an easy way to do so is as follows. Randomly choose enough vertices of $\mathcal{P}$ so that these vertices determine a quadratic polynomial q such that the zero contour of q passes though the chosen vertices, then compute the the shape parameters by evaluating q.

An important feature of the algorithm we derive is that it allows the users to control the shapes of the free-form surfaces produced by the algorithm without knowing the details of the algorithm. This feature is very important for applications like CAD/CAM, where the designers manipulate the shape of the free-form surfaces to achieve functional or aesthetic design objectives.

Recall that for each facet, the cubic polynomials over the double tetrahedron containing the facet is not completely fixed. A double tetrahedron has a vertex outside $\mathcal{P}$, and we call the vertex *the apex* of the double tetrahedron. At the apex of the double tetrahedron of each facet, the value of the cubic polynomials is left as a shape parameter, as was shown in 3.3.

If we think of the algorithm as producing the global function f over the constructed neighborhood Σ of $\mathcal{P}$, then the value of f at each apex is a shape parameter. Since the interior of the free-form surface is exactly the region where $f < 0$, decreasing a shape parameter at a apex pulls the free-form surface towards the apex. Moreover, only nearby quadric patches are affected by

Figure 9: An example of shape control

this shape parameter because the free-form surface construction in the algorithm is local. So, the apexes form a net which controls the shape of the free-form surface through the shape parameters at the apexes.

Figure 7 and Figure 8 illustrate a two dimensional analogy of this shape control scheme. The situation in the three dimension is the same but harder to draw. Figure 9 is an example of two free-form surface having everything identical except the shape parameters.

4 Conclusions

We have presented an algorithm for generating free-form surfaces from the input data of an arbitrary polyhedron with a normal prescribed at each vertex of the polyhedron. The algorithm constructs a smooth piecewise cubic surface interpolating the vertices of the input polyhedron and the prescribed normal at each vertex. The free-form surface construction is local and quadratically precise. In addition, the free-form surface produced can be manipulated through a set of intuitive shape parameters without knowing the details of the algorithm.

Figure 10: A skewed dodecahedron

Figure 11: A tea pot

Figure 10 and Figure 11 illustrate some implemented results. Figure 10 is a skewed dodecahedron with 12 points, 20 facets, and 80 patches; Figure 11 is a tea pot with 45 points, 72 facets, and 266 patches. These two pictures, as well as the pictures shown earlier, are generated by polygonizing the cubic patches and rendering the resultant polygon using Gouraud shading.

We hope to incorporate the free-form surface algorithm into a geometric modeling system and to experiment designing, manipulating, and reasoning about complex smooth objects.

Acknowledgements

I am grateful to John Hopcroft for the support. guidance, and encouragement for this work. I also would like to thank Professors Chris Hoffmann. Joe Warren, and Doug Moore for providing polygonizers which were used for implementing our results. This work is supported by DARPA under ONR contract N00014-86K-0591. NSF Grant DMC-86-17355. and ONR Grant N00014-89J-1946.

Baining Guo is currently a doctoral candidate at Cornell University. He works in the Modeling and Simulation Project in the Department of Computer Science. His research interests include computer graphics, numerical analysis, and theoretical computer science.
Guo received his BS in mathematics from Beijing University in 1982, and he received his MS in computer science from Cornell University in 1989.
Address: Department of Computer Science. Upson Hall, Cornell University, Ithaca. New York 14853. USA

References

[Bajaj C (1990)] Surface fitting using implicit algebraic surface patches. Technical Report CSD-TR-1001, Department of Computer Science. Purdue University, 1990.

[Bajaj C (1989)] Geometric modeling with algebraic surfaces. Technical Report CSD-TR-825. Department of Computer Science. Purdue University, 1989.

[Bajaj C, Ihm I (1989)] Hermite interpolation using real algebraic surfaces. In *Proceedings of the ACM Symposium on Computational Geometry.* West Germany. pages 94-103.

[Bajaj C, Kim M (1987)] Compliant motion planning with geometric models. In *Proceedings of the ACM Symposium on Computational Geometry.* Waterloo. Canada, pages 171-180.

[Blinn J (1982)] A generalization of algebraic surface drawing. *ACM Transactions on Graphics.* 1:235–256, 1982.

[Bloomenthal J. Wyvill B (1990)] Interactive techniques for implicit modeling. *Computer Graphics*, 2:109–116.

[Dahmen W (1989)] Smooth piecewise quadric surfaces. In T. Lyche and L. Schumaker. editors. *Mathematical Methods in Computer-aided Geometric Design.* pages 181-193. Academic Press.

[Farin G (1986)] Triangular Bernstein-Bezier patches. *Computer-aided Geometric Design.* 3:83-127.

[Hoffmann C (1989)] *Solid and Geometric Modeling.* Morgan Kaufmann Publishers, Los Altos, California.

[Hoffmann C, Hopcroft J (1987)] The potential method for blending surfaces and corners. In *Geometric Modeling: Algorithms and New Trends*, pages 347–366.

[Hopcroft J, Krafft D (1986)] The challenge of robotics for computer science. In C. Yap and J. Schwartz, editors, *Advances in Robotics, Vol. 1:Algorithmetic and Geometric Aspects of Robotics.*

[Mann S, Loop C, Lounsbery M, Meyers D, Painter J, DeRose T, Sloan K (1990)] *A survey of parametric scattered data fitting.* Department of Computer Science, University of Washington. Preprint.

[Middleditch A, Sears K (1985)] Blend surfaces for set volume modeling systems. *Computer Graphics*, 19:161–170.

[Moore D, Warren J (1990)] *Adaptive approximation of scattered contour data using piecewise implicit surfaces* Department of Computer Science, Rice University. Preprint.

[Nishimura H, Hirai A, Kawai T, Kawata T, Shirakawa I, Omura K (1985)] *Object modeling by distribution function and a method of image generation.* Journal of Papers Given at the Electronics Communications Conference 1985, J68-D(4).

[Patrikalakis N, Kriezis G (1989)] Representation of piecewise continuous algebraic surfaces in terms of B-splines. *The Visual Computer*, 5:360–374.

[Rockwood A, Owen J (1987)] Blending surfaces in solid geometric modeling. In G. Farin, editor, *Geometric Modeling: Algorithms and New Trends.*

[Sederberg TW (1985)] Piecewise algebraic surface patches. *Computer-aided Geometric Design*, 2:53–59.

[Sederberg TW (1990)] Techniques for cubic algebraic surfaces, *IEEE Computer Graphics and Applications*, 4:14-25.

Beyond Keyframing: An Algorithmic Approach to Animation

A. James Stewart
James F. Cremer

TR 91-1207
May 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

Beyond Keyframing: An Algorithmic Approach to Animation

A. James Stewart
James F. Cremer
Computer Science Department
Cornell University

Originally written January 1989

Abstract

The recent explosion of interest in physical system simulation may soon lead to realistic animation of passive objects, such as sliding blocks or bouncing balls. However, complex *active* objects (like human figures and insects) need a control mechanism to direct their movements. We present a paradigm that combines the advantages of physical simulation and algorithmic specification of movement. The animator writes an *algorithm* to control the object and runs this algorithm on a physical simulator to produce the animation. Algorithms can be reused or combined to produce complex sequences of movements, eliminating the need for tedious keyframing. We have applied this paradigm to control a walking biped. The walking algorithm is presented along with the results from testing with the *Newton* simulation system.

1 Introduction

This paper describes a new paradigm for the control and animation of complex active objects such as the human figure. This approach allows the animator to control an object through an algorithm which specifies certain "intuitive" variables as a function of time and of world state. In the case of human figure walking, the animator might write an algorithm which controls the acceleration of the figure's center of mass at one point in the animation, and which controls the angle of the knees at another point. The algorithmic approach to animation allows this to be done with ease, as demonstrated by the walking algorithm presented in Section 6.

Witkin and Kass [WK88] have combined physical simulation and key-framing to produce realistic animation of their jumping Luxo lamp. With their approach the animator uses *spacetime constraints* to specify several key points for selected variables. These variables may be positions, velocities, forces and so on. Combining spacetime constraint equations with the Lagrangian equations of motion and discretizing over time yields a system of equations that are solved to produce the motion. Since the system is generally underconstrained (having multiple solutions) a solution can be chosen to minimize the power, fuel comsumption and so on.

Our algorithmic approach is similar in that the animator can control accelerations and forces, but differs in that the constraints can be added or removed "on the fly" as the algorithm sees changes in the world state which might not be predictable. In the case of human figure walking the algorithm might, as the foot touches the ground, remove a foot positioning constraint and add a leg stiffening constraint. The exact point of contact is not predictable in advance. Additionally, the algorithmic approach frees the animator from considering the dynamics of impact and other changes in kinematic relationships, which are handled automatically by the simulation component of our system. Incorporating impact into the work of Witkin and Kass would require either guessing the impact points beforehand or incorporating a "force field" approach as described in Section 2.

Other work on combining control and simulation has been done by Barzel and Barr [BB88]. Their method of *dynamic constraints* adds fictitious forces which pull the simulated objects into specified positions. By doing this in the framework of a simulation system, the movement of complex physical objects can be simulated with little work on the part of the animator. A limited form of control is achieved by attaching forces to points on the object and dragging these points.

Various other approaches to combine control and physical simulation have been explored. Wilhelms [Wil87] blends kinematic and dynamic formulations, Isaacs and Cohen [IC87] incorporate inverse dynamics in their simulation system, and Brotman and Netravali [BN88] use dynamics and optimal control to interpolate between key frames.

Some further insights on control can be gained from examining the current literature in the field of robotics. While this field deals with controlling real, physical objects, some of the techniques can be applied to produce simpler animation.

Researchers in robotics have taken various approaches to reduce the complexity of control programs for physical objects. The computed torque method (see [Cra86]) for robot arms can be viewed as simplifying control by reducing the gripper to a unit mass. The control program can ignore the dynamics of the robot arm, only concerning itself with the position of the end effector as a function of time.

In building his one-legged hopping machine, Raibert [Rai86] partitioned control along three intuitive degrees of freedom: hopping, forward speed and body posture. This resulted in surprisingly simple control programs for the hopping robot. For multi-legged machines, Raibert introduced the idea of a "virtual leg" which was defined in terms of the robot's physical legs. This again led to simplified control programs.

Both the computed torque method and Raibert's virtual leg demonstrate that a proper choice of control variables can lead to simplified control programs. The problem with this approach is that there is often no simple closed-form mapping of these control variables onto the forces and torques needed to control the object. In some cases a complete system of equations must be numerically solved to make this mapping. This is called "inverse dynamics" and is typically rejected by robotics researchers as being too expensive to use in real-time control. For the purposes of animation, however, it is ideal.

This is the basis of our algorithmic approach to control. This approach advocates the selection of a small set of intuitive variables which are used by the algorithm in controlling the object. The algorithm constrains these variable with *constraint equations*, which, when combined with the standard Newton-Euler equations of motion, produce a system of equations describing the motion of the simulated object. The system of equations is maintained by our general purpose physical simulator, called *Newton*. The *Newton* simulator is responsible for integrating the motion of the simulated objects over time to produce the animation. As described in the next section, *Newton* also automatically updates the system of equations as kinematic relationships in the simulation change (one such change would occur as the biped's foot touches the ground). Finally, *Newton* provides an interface to allow the algorithm to add and remove constraint equations to and from the system of motion equations.

In the event that the control algorithm underconstrains the motion of the object, constrained optimization techniques are used to choose a motion that optimizes some criterion while satisfying the constraints imposed by the algorithm. Our decision to allow control programs to underconstrain the controlled object – necessitating the use of constrained optimization techniques – is based on the realization that control algorithms often require many fewer control variables than there are degrees of freedom in the controlled object. A robot modeled after the human figure may have as many as two hundred degrees of freedom [Zel82], while the control program for such a robot would only require twenty or thirty degrees of freedom to accomplish its task. In programming our walking biped we used at most eleven of its sixteen degrees of freedom at any given instant.

In summary, the algorithmic approach presented in this paper allows the algorithm to constrain a small set of intuitive variables. The algorithm is allowed to underconstrain the motion of the object, in which case a motion is chosen which optimizes some criterion while obeying the constraints. The *Newton* simulator incorporates the constraint equations into its automatically maintained system of motion equations and integrates over time to produce realistic animation.

Section 2 outlines the relevant background of the *Newton* simulation system. Section 3 describes in detail the algorithmic approach, while Section 4 looks at some low-level controllers used by the walking algorithm. Following this, Sections 5 and 6 outline the biped model and the walking algorithm. and present results from testing the algorithm.

2 Overview of Newton

The walking algorithm described in this paper has been designed and tested using the *Newton* simulation system, part of a large research effort in modeling and simulation at Cornell University. The development of *Newton* was inspired by the need for more general-purpose, flexible simulation systems.

Extensive mechanical engineering research has led to many developments in physical system simulation. The ADAMS [Cha85] and DADS [HL87] systems are examples of large state-of-the-art systems from the mechanical engineering domain. In many ways such systems are very sophisticated: efficient formulations of mechanism dynamics are supported, fancy numerical techniques for solving equation systems are used, object flexibility and elasticity are often handled, and so on. Recent work by graphics and animation researchers [BB88,IC87,MW88,Hah88] in what is termed *physically-based modeling* has generally been less sophisticated but has placed greater emphasis on animation of interesting high-degree-of-freedom mechanisms.

A number of things are still lacking in all of these systems. Typically they have almost ignored geometric considerations and represented objects simply as point masses with associated inertias and coordinate systems. Geometric modeling techniques have matured enough to allow object representations used by dynamic simulations to include a complete geometric description usable by a geometry processing module. Furthermore, impact, contact, and friction are typically handled by current systems in an *ad hoc* or rudimentary manner, if at all. In some cases, for instance, any possible impacts must be specified in advance; in others, a kind of "force field" technique is used, in which between every pair of objects there is a repelling force that is negligible except when objects are very close together. In addition, the desire to manipulate high-degree-of-freedom objects suggests that a module for specification of control algorithms should be a significant part of a dynamics system.

2.1 Newton Architecture

Using *Newton*, a designer can define complex three-dimensional physical objects and mechanisms and can represent object characteristics from a wide range of domains. An object is made up of a number of "models," each responsible for organization of object characteristics from a particular domain. In most simulations the basic domains of geometry, dynamics, and controlled behavior are modeled. A dynamic modeling system, for example, is responsible for maintaining an object's position, velocity, and acceleration, and for automatically formulating the object's dynamics equations of motion. A geometric modeling system is responsible for information about an object's shape, distinguished features on the object, and computation of geometric integral properties such as volume and moments of inertia. It also detects and analyzes object interpenetrations so that an interference modeling system can deal with collisions between objects.

Newton is composed of three main components: the definition and representation module, the analysis module and the report system. The definition module analyzes high level language descriptions of *Newton* entities and organizes the corresponding data structures. The analysis component implements the top-level control loop of simulations and coordinates the working of various analysis subsystems. The report system handles generation of graphical feedback to users during simulations as well as recording of relevant information for later regeneration of animations.

2.2 Dynamic Analysis in Newton

A complex physical object is modeled as a collection of rigid bodies related by constraints. Newton-Euler equations of motion are associated with each individual rigid body.[1] At the time an object is created the equations are of the form

$$m\ddot{r} = 0$$

$$J\dot{\omega} + \omega \times J\omega = 0.$$

where m is the mass, $\ddot{r}$ is the second time derivative of the position (ie. the acceleration), J is the 3×3 inertia matrix, and ω and $\dot{\omega}$ are the rotational velocity and acceleration, respectively.

A specification that two objects are to be connected with a spherical hinge is met by the addition of one vectorial constraint equation and the addition of some terms to the motion equations of the constrained objects. For a holonomic constraint such as this one, the second derivative of the constraint equation can be used along with the modified motion equations

[1] *Newton* is capable of using dynamics formulations other than the one outlined here. We are also working on incorporating non-rigid bodies into the system.

to solve for object accelerations and reaction forces. Thus, the equations above become

$$\begin{aligned} m_1\ddot{r}_1 &= F_{hinge} \\ J_1\dot{\omega}_1 + \omega_1 \times J_1\omega_1 &= c_1 \times F_{hinge} \\ m_2\ddot{r}_2 &= -F_{hinge} \\ J_2\dot{\omega}_2 + \omega_2 \times J_2\omega_2 &= c_2 \times -F_{hinge} \end{aligned}$$

$$\ddot{r}_1 + \dot{\omega}_1 \times c_1 + \omega_1 \times (\omega_1 \times c_1) = \ddot{r}_2 + \dot{\omega}_2 \times c_2 + \omega_2 \times (\omega_2 \times c_2),$$

where c_i is the vector from object i's center of mass to the location of the hinge and F_{hinge} is the constraint force that keeps the objects together. Note that the last equation above is the second time derivative of the holonomic constraint equation $r_1 + c_1 = r_2 + c_2$ for spherical joints. Other kinds of hinges commonly used in *Newton* include revolute or pin joints, prismatic joints, springs and dampers, and rolling contacts.

If gravity is present during the simulation the system will automatically add gravitational force terms to the objects' translational motion equations. The system keeps track of the constraints responsible for the various terms in the motion equations. Thus, constraints, and their corresponding motion equation terms, can be removed at any time without necessitating complete rederivation of the system of motion equations.

Using this method of dynamics formulation, closed-loop kinematic chains are handled as simply as open chains. Though the formulation does lead to a large set of equations, the matrices are very sparse and often symmetric. Thus, acceptable efficiency is achieved by the use of sparse matrix solution techniques.

2.3 Event handling, impact and contact

Newton, unlike many other simulation systems (though see [Fea85]), can automatically and incrementally reformulate the motion equations as exceptional events occur during simulations. One kind of exceptional event is a change in kinematic relationship between objects. Figure 1 shows a block that was initially sliding along a table top. After some time the edge of the table is reached and the contact relationship changes from a plane-plane contact to a plane-edge contact. Still later the contact is broken altogether. These changing contact relationships are automatically detected by *Newton*. The system of motion equations and the related constraint equations are automatically maintained by *Newton* to reflect these changing relationships.

During the course of a simulation, a variety of events can occur that require special processing. *Newton*'s event handler is primarily responsible for detection and resolution of impacts, for analysis of continuous contacts

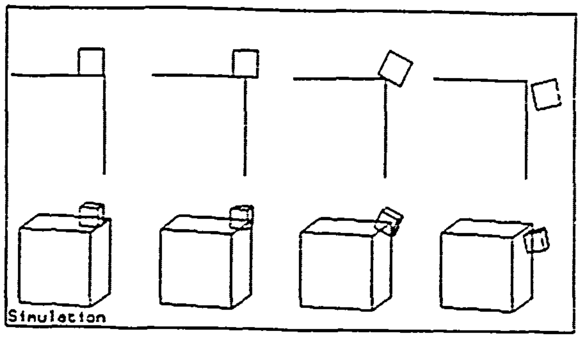


Figure 1: Changing Kinematic Relationships

between objects and corresponding maintenance of *temporary hinges*, special kinds of hinges that model one sided constraints between objects in contact, and for handling of events specified by control programs that necessitate changes in the constraint set. For example, the walking algorithm might tell the event handler to notify it when the biped's foot touches the ground so that it can change the constraint equations.

The geometric modeling subsystem is responsible for detecting and analyzing impacts and interpenetrations. In the usual method of handling impacts, the dynamic analysis module formulates impulse-momentum equations in a manner completely analagous to the formulation of the basic dynamics equations, and solves these equations to produce the instantaneous velocity changes caused by the impact. The details of *Newton*'s methods for handling impact, contact and other exceptional events are given in [HH87,HH88,CS88,Cre89].

3 The Algorithmic Approach

In *Newton's* automatically-generated equations of motion certain quantities are considered to be *unknowns*. A system of simultaneous linear equations is solved at each time step to produce values for the unknowns. These values are integrated over time to produce the simulated motion. Typically, the unknowns consist of accelerations and joint constraint forces, while positions, velocities and joint control torques are *knowns*.

In the algorithmic approach, the programmer controls "intuitive" quantities defined as linear combinations of the unknowns. The programmer might, for example, want to control the acceleration of the center of mass of a biped without explicitly controlling each component of the biped. To do this, the algorithm must define the acceleration of the center of mass in terms of the accelerations of the centers of mass of the primitive components of the ob-

```
procedure initialize

  begin
  add-equation " $\ddot{r}_{cm} = \frac{1}{M} \sum m_i \ddot{r}_i$ "
  end

procedure controller( time )

  begin
  $\ddot{r}_{cm}$ = $f$( time )
  end
```

Figure 2: The Format of an Algorithm

ject. Over the course of execution, the algorithm must supply the desired acceleration of the center of mass at each point in time.

Figure 2 shows the format of a control algorithm. For the sake of clarity the algorithms will be described in a Pascal-like notation[2]. Two procedures are always present: one to initialize the algorithm (called `initialize`) and one to be executed repeatedly over the course of the task (called `controller`). The `controller` procedure has access to the complete state of the system. The algorithm of Figure 2 trivially defines and controls the acceleration of the center of mass of an object (the function f must be defined elsewhere).

Defining and controlling a three-dimensional vectorial quantity like the acceleration of the center of mass has the effect of adding three constraint equations to the system of simultaneous linear equations that describe the instantaneous motion of the object. By considering joint torques as unknowns in this augmented system of equations, the system can be solved to produce motion that satisfies the additional constraint equations. This is a simple application of inverse dynamics.

For an object with n degrees of freedom the control algorithm can define and control up to n independent scalar quantities[3]. If fewer than n equations are added the system of motion equations is underdetermined, and many different solutions could satisfy the constraints of the control algorithm. In this case the algorithm must guide the selection of a solution by providing a cost function which is quadratic in the unknowns. A standard numerical optimization technique is used to compute a solution that instantaneously (for each point in time) minimizes the cost function while obeying the algorithm's constraints. This is different from the approach of Witkin and Kass

[2]The algorithms are, for now, written in Lisp.

[3]The additional definitional equations could make the system of motion equations inconsistent. This would be an error on the part of the control algorithm.

[WK88], who optimize over the whole animation. This reflects the different philosophies of the two systems: Witkin and Kass specify all of the information beforehand, while we let the control algorithm make decisions *during* the animation. Such "on the fly" decisions make it impossible to do global optimization, but allow much more versatility in the control algorithm by not requiring *a priori* knowledge of impacts and other exceptional events.

In summary, the programmer designs an algorithm in a high-level computer language to control intuitive degrees of freedom of the object. These degrees of freedom are defined as linear combinations of the unknowns in the object's equations of motion. An augmented linear system of equations describes the instantaneous behavior of the object; this system can be solved to produce the object's configuration at each point in time. If the system is underdetermined, the algorithm can provide a cost function to guide the choice of a solution.

In the remaining sections we describe the application of this approach to the design of a simple walking algorithm.

4 Low-level Controllers

In designing algorithms with *Newton* we found ourselves frequently using PD controllers[4] and curve-fitting controllers to control the "trajectory" of many of the defined quantities. In controlling the biped, for example, a quintic interpolation was used to plot the trajectory of the heel, and a PD controller was used to orient the foot before it struck the ground. A small library of these controllers is used in the biped algorithm, and will be described here.

PD controllers are used in the biped algorithm to control orientation, position and joint angle. Each controller adds an equation to the system of motion equations which defines the second derivative of the quantity in terms of the first derivative and the quantity itself. The procedure in Figure 3 produces accelerations to move an `object` to within 1% of a position `x-desired` within a given time `delta-time`. The quantities x, v and a are data structures representing state variables of the controlled object. These data structures are used by the `add-named-equation` function to create the appropriate equation.

Execution of the procedure in Figure 3 causes a named equation to be

[4]A PD controller (Proportional, Derivative), also known as a "spring and damper" controller, relates the second derivative of a variable linearly to the error in the variable's first derivative and to the error in the variable itself. The equation is $\ddot{x}+\frac{2}{\tau}\dot{x}+\frac{1}{\tau^2}(x - x_{\mathrm{desired}}) = 0$ for some appropriate τ. PD controllers are used extensively in robotics to move robot joints into specified positions by calculating the joint acceleration as a function of the position and velocity errors. A good explanation can by found in [Cra86]. Barzel and Barr [BB88] use a form of PD controller to achieve their dynamic constraints.

```
procedure position-with-PD( constraint-name, object,
                            x-desired, delta-time )

  var $x$, $v$, $a$:  quantity
      $\tau$:  real

  begin
  $x$ = get-position-quantity( object )
  $v$ = get-velocity-quantity( object )
  $a$ = get-acceleration-quantity( object )

  $\tau$ = - delta-time / log( .01 )

  add-named-equation( constraint-name,
                      " $a + \frac{2}{\tau} v + \frac{1}{\tau^2}(x$ - x-desired) = 0 " )
  end
```

Figure 3: PD Controller Used in Positioning

added to the system of motion equations. This equation will continue to affect the motion of the object until it is explicitly removed by the control algorithm.

A complete list of controllers available to the bioed walking algorithm is shown in Figure 7 at the end of the paper. Those with `quintic` in their name do quintic interpolation to achieve the desired position and velocity in the desired time. Quintic interpolation was chosen over cubic interpolation to eliminate "jerk" (discontinuous acceleration) from the beginning and end of the trajectory.

5 The Biped Model

The simulated biped is composed of a torso, two legs with knee joints and two feet with toe joints. This model was adapted from a description in [McM84] and is shown in Figure 4. The hips and ankles are three degree of freedom spherical joints, while the knees and toes are one degree of freedom revolute joints, making a total of sixteen degrees of freedom. The biped is about six feet tall with moments approximating those of a human being.

We hope to improve this model by incorporating joint limits and elastic tendons. McMahon suggests that, during walking, energy is stored in stretched tendons and is released when the stretched leg swings forward [McM84]. This idea might be used to simplify the walking algorithm described in the next section.

Newton's impact handling capabilities have not yet been extended to

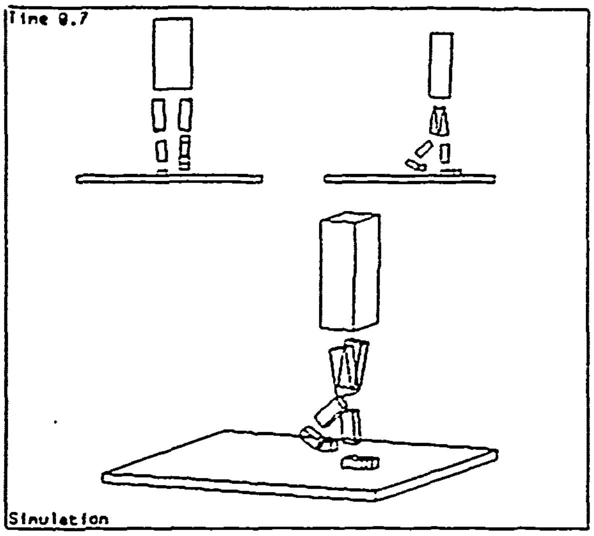


Figure 4: Simulated Biped Model

accurately model the impact of the feet upon the ground. Instead, impact is simulated by adding an external force and torque to the feet that holds them level with the ground until they are released with an explicit command from the control algorithm. This is as though the biped was walking with magnetic shoes on a steel plate. Very shortly we expect to adapt the algorithm to incorporate realistic impact.

6 The Walking Algorithm

An abbreviated version of the walking algorithm is shown in Figures 8 and 9, which can be found at the end of this paper. The algorithm cycles through a set of six states: swing the right leg, land the right foot, lift the left foot, swing the left leg, land the left foot, lift the right foot and then repeat the cycle. In the *swing* phase, a quintic trajectory is plotted for the swing foot with `move-heel-to-target`, while the stance leg is stiffened with `set-angle-with-PD` and the foot is oriented for landing with `orient-with-PD` (shown under START in Figure 9). In the *landing* phase, the leading leg is stiffened as the foot nears the ground. Following this, the *takeoff* phase flexes the trailing leg, causing the trailing foot to lift from the ground. Once the trailing toe is bent to 10° the flexing constraint is removed and the *swing* phase begins for the trailing leg.

The largest number of constraints are applied during the *swing* phase, as shown in Table 1. Since the biped has sixteen degrees of freedom (DOF) it remains underconstrained at all times. A quadratic cost function is therefore defined (in `initialize` of Figure 9) in order to fully determines the motion

Constraint Name	DOF	Constrained Item
TORSO-CONSTRAINT	3	torso orientation in 3 dim
L-KNEE-ANGLE	1	angle of revolute knee joint
R-HEEL-TRAJ	3	heel acceleration in 3 dim
R-FOOT-ORIENTATION	3	foot orientation in 3 dim
R-TOE-ANGLE	1	angle of revolute toe joint

Table 1: Swing Phase Constraints

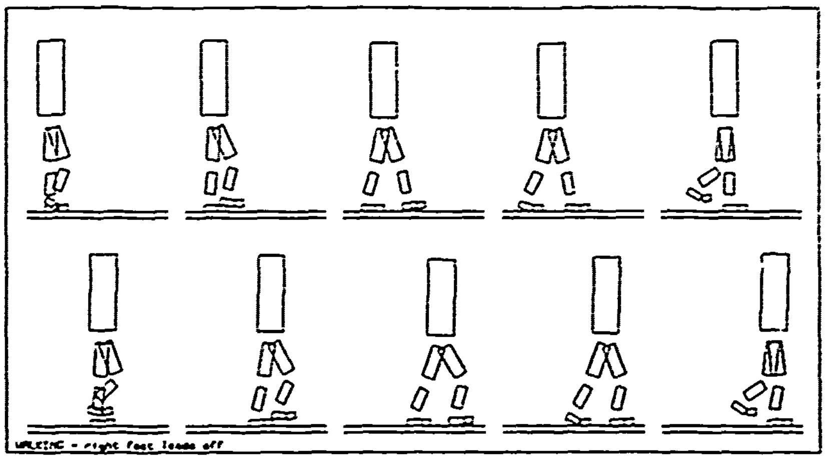


Figure 5: Walking Cycle

of the biped. The cost function is a weighted sum of the translational and angular accelerations, and of the difference between the torso translational acceleration and some acceleration defined by a function F which tries to keep the torso mid-way between the two feet.

We found that a cost function which minimizes instantaneous translational and rotational acceleration usually produces smooth motion. In the case of the simulated biped, the cost function causes the constrained heel acceleration to be achieved by a linear combination of small accelerations of many components of the body, rather than a few large accelerations of those components which are near the heel. We have observed that the combination of many small accelerations yields more stable motion than large, local accelerations.

The walking algorithm was tested with the *Newton* simulation system. Figure 5 shows ten frames in which the biped completes a full cycle of the six phases described above. The full simulation consisted of twenty seconds of straight-line walking on a flat surface and generated the statistics shown in Figure 6. The version of the algorithm that produced these statistics had the biped increase speed at 4.0 seconds, as can be seen on the graphs.

7 Summary

We have presented an algorithmic approach to control. This approach allows the animator to choose intuitive degrees of freedom by which to control an object. The control algorithm adds and removes constraint equations "on the fly" as the world state changes; *a priori* knowledge of the exact moment of each state change is not required. With the algorithmic approach, all consideration of dynamics and impact is left to the *Newton* simulation system. The burden on the animator is further reduced by allowing underdetermined specification of motion through the use of constrained optimization techniques.

We have presented an algorithm to control a simulated biped, along with results from its execution on the *Newton* simulation system. The algorithm has the advantage of being intuitive, simple to program, and reusable.

Unlike keyframing, the algorithmic approach does not require the animator to repeat the work of creating new key frames for every walking sequence. Unlike keyframing, the algorithmic approach allows various algorithms to be combined to produce long animated sequences. We believe that in the future, animating complex physical objects will require a structured, algorithmic approach similar to that presented in this paper.

8 Future Work

We will incorporate elastic tendons and joint friction into the *Newton* simulation system and modify the walking algorithm accordingly. From there we hope to develop a suite of algorithms to allow a biped to walk, turn, climb stairs, manipulate objects, and so on. In keeping with the structured approach presented in this paper we will attempt to combine these algorithms to have the biped perform complicated tasks. In carrying an object up a flight of stairs the high-level algorithm would combine subroutines to pick up the object, walk to the stairs, climb the stairs and deposit the object.

Acknowledgements

This work was supported in part by NSF grant DMC 86-17355, ONR grant N0014-86K-0281 and DARPA grant N0014-88K-0591. Support for James Stewart is provided in part by U.S. Army Mathematical Sciences Institute grant U03-8300 and NASA training grant NGT-50327. The *Newton* system is being developed in Common Lisp on Symbolics Lisp Machines and can be used on other machines supporting Common Lisp.

References

[BB88] Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constraints. In *Computer Graphics (SIGGRAPH 88)*, pages 179–188. ACM, August 1988.

[BN88] Lynne S. Brotman and Arun N. Netravali. Motion interpolation by optimal control. In *Computer Graphics (SIGGRAPH 88)*, pages 309-315. ACM, August 1988.

[Cha85] M. Chace. Modeling of dynamic mechanical systems. Presented at the CAD/CAM Robotics and Automation Institute and International Conference, Tuscon, Arizona, February 1985.

[Cra86] John J. Craig. *Introduction to Robotics: Mechanics and Control.* Addison Wesley, 1986.

[Cre89] James F. Cremer. PhD thesis, Cornell University, in preparation, 1989.

[CS88] James F. Cremer and A. James Stewart. Using the *newton* simulation system as a testbed for control. In *Proceedings of the 3rd IEEE International Symposium on Intelligent Control*, 1988.

[Fea85] Roy Featherstone. The dynamics of rigid body systems with multiple concurrent contacts. In O. D. Faugeras and G. Giralt, editors, *Robotics Research: The Third International Symposium*, pages 191-196. The MIT Press, 1985.

[Hah88] James K. Hahn. Realistic animation of rigid bodies. In *Computer Graphics (SIGGRAPH 88)*, pages 299–308. ACM, August 1988.

[HH87] C. M. Hoffmann and J. E. Hopcroft. Simulation of physical systems from geometric models. *IEEE Journal of Robotics and Automation*, RA-3(3):194–206, June 1987.

[HH88] C. M. Hoffmann and J. E. Hopcroft. Model generation and modification for dynamic systems from geometric data. Presented at the NATO Workshop on CAD-based Programming for Sensor-based Robots, Il Ciocco, Italy, July 1988.

[HL87] E. J. Haug and G. M. Lance. Developments in dynamic system simulation and design optimization in the center for computer aided design: 1980-1986. technical report 87-2, University of Iowa, February 1987.

[IC87] Paul M. Isaacs and Michael F. Cohen. Controlling dynamic simulation with kinematic constraints, behavior constraints and inverse dynamics. In *Computer Graphics (SIGGRAPH 87)*, pages 215-224. ACM, July 1987.

[McM84] T. A. McMahon. Mechanics of locomotion. *The International Journal of Robotics Research*, 3(2):4-28, 1984.

[MW88] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. In *Computer Graphics (SIGGRAPH 88)*, pages 289-298. ACM, August 1988.

[Rai86] M. H. Raibert. *Legged Robots That Balance.* The MIT Press, 1986.

[Wil87] J. Wilhelms. Using dynamic analysis for realistic animation of articulated figures. *IEEE Computer Graphics and Applications*, 7(6):12-27, 1987.

[WK88] Andrew Witkin and Michael Kass. Spacetime constraints. In *Computer Graphics (SIGGRAPH 88)*, pages 159-168. ACM, August 1988.

[Zel82] D. Zeltzer. Motion control techniques for figure animation. *IEEE Computer Graphics and Applications*, 2(9):53-59, 1982.

```
position-with-PD( constraint-name, object, $x_d$, $\Delta t$ )
position-point-with-PD( constraint-name, object, point-on-object, $x_d$, $\Delta t$ )
orient-with-PD( constraint-name, object, $\Phi_d$, $\Delta t$ )
set-angle-with-PD( constraint-name, joint, $\theta_d$, $\Delta t$ )

position-with-quintic( constraint-name, object, $x_d$, $v_d$, $\Delta t$ )
position-point-with-quintic( constraint-name, object, point-on-object, $x_d$, $v_d$, $\Delta t$ )
orient-with-quintic( constraint-name, object, $\Phi_d$, $\dot{\Phi}_d$, $\Delta t$ )
set-angle-with-quintic( constraint-name, joint, $\theta_d$, $\dot{\theta}_d$, $\Delta t$ )
```

Figure 7: Low-level Controllers

```
const   time-in-air             = 0.5 s
        stride                  = 0.5 m
        direction               = (1 0 0)
        inside-step-fraction    = 20 %
        heel-Y-strike-speed     = -0.05 m/s
        heel-X-strike-speed     = 0.02 m/s
        foot-strike-orientation = 10° about (0 0 1)
        torso-orientation       = -10° about (0 0 1)


var     phase:  (start r-swing r-land l-lift l-takeoff l-swing l-land r-lift r-takeoff )


procedure move-heel-to-target( constraint-name, foot, other-foot, hip, other-hip )

  var target-x, target-v, hip-to-hip:  vector

  begin
  hip-to-hip = get-position( TORSO, hip ) - get-position( TORSO, other-hip )

  target-x = get-position( other-foot, HEEL ) + stride × direction
             + inside-step-fraction × hip-to-hip

  target-v = heel-Y-strike-speed × (0 1 0) + heel-X-strike-speed × direction

  position-point-with-quintic( constraint-name, foot, HEEL, target-x, target-v, time-in-air )
  end
```

Figure 8: Definitions for the Walking Algorithm

```
procedure initialize

  let F = $K_p(\frac{1}{2}(r_{l-foot} + r_{r-foot}) - r_{torso}) + K_v(\frac{1}{2}(\dot{r}_{l-foot} + \dot{r}_{r-foot}) - \dot{r}_{torso})$

  begin
  quadratic-cost = $\sum \dot{\omega}^2 + \sum \ddot{r}^2 + 20\,(\ddot{r}_{torso} - \mathbf{F})^2$
  phase = START
  end

procedure controller( time )

  begin
  case phase of

    START:
      phase = R-SWING
      orient-with-PD( TORSO-CONSTRAINT, TORSO, torso-orientation, 2.0 s )
      move-heel-to-target( R-HEEL-TRAJ, R-HEEL, L-HEEL, R-HIP, L-HIP )
      set-angle-with-PD( L-KNEE-ANGLE, L-KNEE, 175°, 0.1 s )
      orient-with-PD( R-FOOT-ORIENTATION, R-FOOT, foot-strike-orientation, time-in-air )
      set-angle-with-PD( R-TOE-ANGLE, R-TOE-JOINT, 0°, time-in-air )

    R-SWING:
      if distance-to-target( R-FOOT ) < 0.01 m then

        phase = R-LANDING
        remove-constraint( R-HEEL-TRAJ )
        set-angle-with-PD( R-KNEE-ANGLE, R-KNEE, 175°, 0.05 s )

    R-LANDING:
      if heel-has-touched( R-FOOT ) then

        phase = L-TAKEOFF
        remove-constraints( R-FOOT-ORIENTATION, R-TOE-ANGLE, L-KNEE-ANGLE )
        set-angle-with-PD( L-KNEE-ANGLE, L-KNEE, 160°, 0.1 s )

    L-TAKEOFF:
      if joint-angle( L-TOE-JOINT ) > 10° then

        phase = L-SWING
        remove-constraint( L-KNEE-ANGLE )
        move-heel-to-target( L-HEEL-TRAJ, L-HEEL, R-HEEL, L-HIP, R-HIP )
        orient-with-PD( L-FOOT-ORIENTATION, L-FOOT, foot-strike-orientation, time-in-air )
        set-angle-with-PD( L-TOE-ANGLE, bL-TOE-JOINT, 180°, time-in-air )

    Cases L-SWING, L-LANDING, and R-TAKEOFF
    are analogous to the preceding three cases.

    end
  end
```

Figure 9: Abbreviated Walking Algorithm