Interactive Geometric Constraint Systems

Mark W. Brunkhart

Master's Project
under the direction of Carlo Séquin

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley

May 10, 1994

Abstract

Graphics and modeling systems allow users to place objects or primitives in space and permit the individual editing of these objects. A more powerful paradigm allows relationships expressed by constraints to be established and maintained between pairs of these primitives. For example, two lines might be constrained to be parallel or two points constrained to be coincident. These geometric constraints lead to a "live" model in which design changes to one part of the model may induce changes elsewhere thus permitting the encoding of the designer's intent into the model.

Constraints may describe complex non-linear algebraic relationships between the parameters of a model. To provide constraint satisfaction and maintenance in an interactive environment requires extremely fast and general equation solving techniques. We have developed a system called MechEdit for the interactive editing and animation of planar linkages to demonstrate techniques applicable to general geometric constraint-solving problems.

MechEdit combines analytic and iterative numerical techniques for constraint-system solving. A system of equations derived from a linkage is first examined and ordered based upon dependencies between parameters. Parameters that can be solved by propagating values are solved using backsubstitution. Among the remaining unsolved equations, a set of basic geometric reductions are used to identify parameters that can be solved analytically by combining pairs of equations. Finally, Newton-Raphson iteration is used to solve the remaining parameters. Singular value decomposition is used to solve systems of linear equations within the inner-loop of the Newton-Raphson iteration to provide quasi-physical motion for under-constrained and degenerate systems. By providing high-performance, interactive constraint maintenance and robust solving for complex systems, MechEdit demonstrates the power of constraints as a paradigm for specifying relationships between the primitives of a geometric model.

w

Contents

List of Figures

1 Introduction

The interactive animation of mechanisms provides a challenging domain for research in both user interfaces and constraint-based systems. Interactivity is a demanding goal. To achieve it, not only must a tool have a clean, consistent interface, but for computationally complex tasks such as geometric constraint solving, a tool must take advantage of every computational opportunity to improve performance. In the domain of design tools, however, the rewards of interactivity are outstanding. Users who can smoothly explore the domain in which they are designing by interactively editing and animating their constrained model can identify trends and achieve intuitions far more rapidly than the users of non-interactive systems that only permit the testing of a finite set of discrete designs. The modern graphics workstation could provide a virtual workshop for rapid prototyping in which hundreds of designs can be constructed, tested, and modified in the time it would take to get a single design back from a real-world machine shop. While this possiblity is intriguing, it is hindered by the computational requirements of the underlying constraint solver. This report examines constraint satisfaction and maintenance techniques in the limited domain of planar linkages, identifying many of the issues and difficulties surrounding constraint solvers and offering solutions where solutions have been identified.

1.1 MechEdit: A Planar-Linkage Editor

This report describes a tool for constructing and animating planar linkages called MechEdit. Planar linkages are constructed from rigid bodies called links connected by revolute and prismatic joints. Revolute joints are essentially hinges, allowing a change in the relative angle between two links. Prismatic joints are essentially sliders, allowing relative displacement between two links along a fixed line. The tool we have constructed can animate any planar linkage made out of combinations of such links and joints.

MechEdit is an interactive graphical tool. The user designs with the mouse, grabbing and moving links as though they were lying on a table in front of him. The partially constructed mechanism moves physically, that is, as it would if the designer had actually hooked the links together on his desk and were moving a particular link to see how his design responded. The design is always available to be animated. In fact, there is no distinction between construction and animation times so animation may be used to aid in construction. For example, a user may construct two separate linkages and then pull one according to its constrained mobility into contact with the other, creating a joint and a single mechanism. This realistic, physical workspace is a simple and powerful paradigm for design as it emulates the physical world and thus can draw upon the user's intuition.

MechEdit is an extension, in part, of the work done by Eric Enderton and reported in [ENDE90]. Enderton describes a method for rapidly calculating the positions of links when animating certain classes of mechanisms. Enderton's method is based upon a closed-form solver that finds small submechanisms for which predefined solution methods have been written. For example, given three links as shown in figure 1, where the position of points A and B are fixed and the links are joined at C, it is clear that the orientation of links I and II can be determined by intersecting two circles. In Enderton's terms, this is an RRR loop since three links (links I, II, and an implied ground link) are connected by three revolute (R) joints, and would be solved by an RRR solver which analytically computes the intersection of two circles.

Enderton's planar-linkage editor suffered a number of shortcomings. Only certain classes of mechanisms could be solved in closed-form. Other linkages requiring some form of numerical solution could not be animated. Also, Enderton's tool could animate only linkages that exhibited a single degree of freedom and were generic, that is, were not dependent upon special cases such as links which are precisely the same length or exactly parallel. Linkages with mobility greater than one or special degeneracies could not be animated. Finally, mechanisms could be animated only by altering the joint-angle of a fixed pivot. A user could not, for example, drag the coupler of a 4-bar linkage. All of these shortcomings have been corrected in our current work, primarily through the use of numerical techniques.

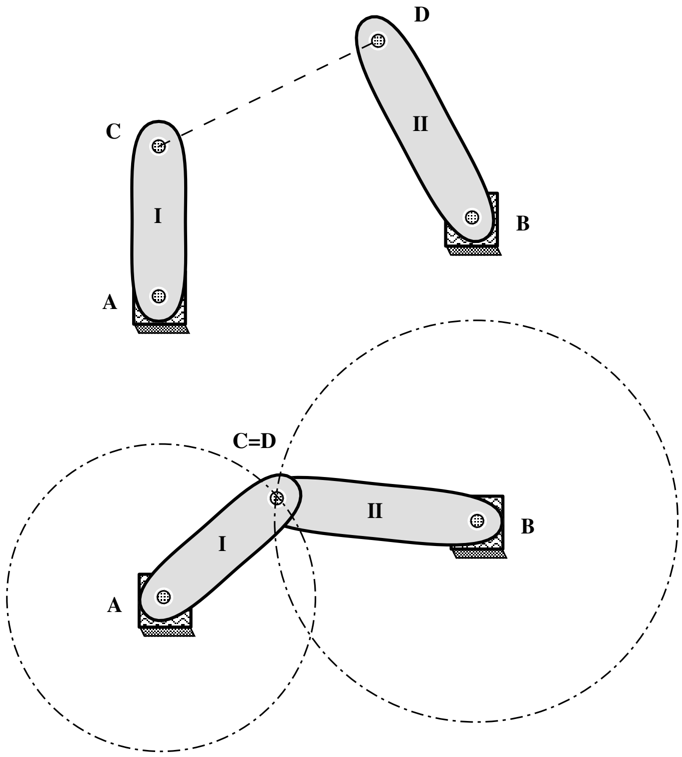
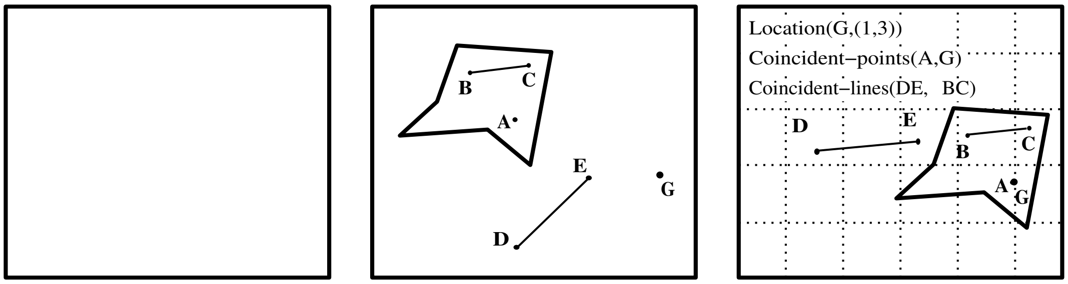


Figure 1: Solving an RRR loop: If points A and B are fixed, and a revolute joint binds points C and D, the orientation of the two links can be calculated by intersecting two circles.

1.2 Geometric Constraint Satisfaction

The animation of mechanisms is a subclass of the task known as geometric constraint solving. Geometric constraint problems involve a multidimensional space, objects within that space, and geometric relationships between these objects (figure 2). For example, for planar linkages, we have a two-dimensional space, a collection of links made up of points defining their geometry, and a collection of relationships between these points. To define mechanisms, we have relationships, also known as constraints, such as "point G is fixed in space at location (1,3)". For a revolute joint, one could specify "point A on link I is coincident with point G". For prismatic joints, one might use a constraint such as "line BC on link I and line DE are coincident", that is, these line segments lie on the same line. Finally, we must also have the notion of a ground link (or ground plane) in order to describe positions and orientations absolutely. The ground plane imposes a coordinate system on the two-dimensional space and was used implicitly when describing the location of G above.

Mechanisms require only a subclass of a more general class of geo-



A two−dimensional space S   Objects within S   Constraints and a ground plane

Figure 2: The elements of planar geometric constraint problems.

metric constraints. General geometric constraints include such notions as parallelism of lines, congruence of lines and angles, incidence, and intersection among others. We will be more precise about which constraints are required to define planar mechanisms later in the report; however, it is useful to recognize from the beginning that this work is a subclass of a more general problem in geometric constraint solving, a problem that was introduced to the world of computer science with Ivan Sutherland's "Sketchpad" ([SUTH63]).

Sketchpad and many systems since have allowed users to specify geometry declaratively; that is, a user specifies a relationship between two objects, and the computer attempts to position the objects so as to satisfy the relationship. As more and more relationships are specified, satisfaction of all constraints becomes increasingly difficult, and it is the burden of the constraint-solving program to search the rapidly decreasing space of valid solutions to find those that satisfy the added constraints. Specifying constraints and requiring the machine to satisfy them is a form of declarative programing. Declarative programming is contrasted with imperative programming in which the user must program an algorithm by which a solution to a problem can be found ([FETZ90]).

It is easy to herald declarative programming as a simpler method of solving problems than imperative programming as no knowledge of solution algorithms is required by the programmer; however, one must be careful to recognize that declarative programs are only as powerful as the solver that underlies them. Declarative programming is, in many ways, a misnomer. If a class of problems is solvable with a particular algorithm, then a general solver can be written to solve all problems in this class. We may then 'program' problems in this class in a declarative manner, simply stating the problem as input to our general solver. Thus, declarative programs are simply input to more general, problem solving algorithms. Geometric constraint systems happen to provide an extremely interesting form of declarative programming, and it is the focus of this document to describe the general, problem solving algorithms necessary to permit declarative programming in the setting of planar linkages.

With the arrival of powerful geometric constraint solving systems, constraints should easily find their way into many graphical applications. Constraints are the language of design. Constraint satisfaction is the primary task of designers. Simple examples of constraint satisfaction are plentiful in both engineering and architecture. An architect places a door in a wall. When the wall is moved, the door is expected to move as well. If a beam connects two walls, the beam should expand if the walls are moved further apart. The two walls of a hallway are expected to be parallel. If the hallway is rotated during the design process, both walls are expected to move. A mechanical engineer might indicate that a collection of gears must maintain contact in space, that a drill must be at a 45 degree angle to a base plate, or that no part may collide with any other part. With the exception of the last, these are all extremely simple constraints; however, they help to demonstrate some of the advantages of constraint-based systems, advantages that are discussed below.

Specification of geometry by constraints is far more powerful than specification of raw geometry in terms of point locations and connectivity of edges and faces. First, such specification is inherently easier both to construct initially and to modify. Consider the time it might take to specify all of the points that define two parallel walls containing numerous doors and windows in a building. If facilities exist to simply duplicate one wall and constrain the copy to lie parallel to the original, the designer's task is simplified substantially. While most systems allow such simple operations to construct hierarchical models, even greater benefit is gained if the constraint between the two walls is recorded and maintained as the design of the building is altered. Thus, if one of the two walls is moved, the second wall follows as well, enforcing the designer's original intent that the two walls be parallel. Constraints are a simple mechanism by which design intent can be stored within a model. The tedium of resatisifying a set of design requirements when one object is moved or altered is removed entirely if constraints exist that specify these requirements. A similar argument is obvious for the specification of geometry in computer-aided design for mechanical engineering.

While MechEdit is a constraint-satisfaction tool for a limited domain, the problems that it addresses are inherent in the more general problem of geometric constraint satisfaction. General constraint-solvers must recognize opportunities to use analytic solutions whenever possible to enhance performance. They must offer predictable solutions when numerical solutions are required and must deal robustly with problems of convergence and numerical instability. Precautions must be taken for over-constraint, and degeneracies must be handled in some reasonable manner. Many problems are extremely under-constrained. Constraint-solving programs must be capable of dealing with under-constraint and must assign values to unconstrained variables in a consistent and predictable way. Finally, interfaces to constraint systems must insure that the system is used correctly, that its weaknesses cannot be exploited, and that its capabilities are within easy reach of the user. In the limited domain of planar linkages, MechEdit satisfies these requirements.

1.3 Research Challenges for Constraint Systems

Mechanism animation, in particular, and interactive visualization tools in general present special challenges to constraint satisfaction systems that make research in this area particularly interesting. Interactivity requires exceptional performance. Users are quick to discard a system that makes them wait, regardless of the power it provides. If a tool is to be used throughout the design process including the early, conceptual stages of design, it must respond rapidly to the user. For a constraint system, this means that a user must be able to specify constraints quickly and then have these constraints maintained as he modifies the design. If constraints are to become a fluid part of the editing process, they must be resatisfied as quickly as a user can drag an object with a mouse. Unfortunately, constraint systems are characteristically plagued by poor performance. At the heart of most constraint systems, numerous nonlinear simultaneous equations must be solved. Such solutions are typically computationally expensive, making interactivity a challenging endeavor.

Interactive tools must also be predictable. Tools that do not respond in a predictable way to a user's actions are useless. Extremely powerful commands frequently go unused because the user cannot predict the result of the command or cannot easily control that result. Once again, this is a challenge for the constraint-satisfaction system. When solving even relatively small systems of nonlinear equations, many discrete solutions are possible. Consider for example the case described at the beginning of this report in which two circles must be intersected (figure 1). In general, two real solutions exist. Which one does the user expect to see as a solution? Constraint systems that use numerical methods may be hindered by problems with instability, that is, the tendency for the system to report one solution for a particular input but a different one if the input is perturbed even slightly. In a graphics application in which parameters are varied smoothly, such instability displays itself as a model oscillating between two different positions in space and is generally unacceptable to the user. Even more challenging is the task of creating a predictable response when there are an infinite number of possible solutions. This may occur quite frequently as it is difficult and not necessarily desirable to fully constrain an initial design. This will leave multiple degrees of freedom and, in turn, the space of possible solutions from which the constraint system must select is infinite.

Finally, interactive tools must be robust. Systems that are incapable of handling certain cases or provide incorrect solutions to some inputs will almost never be accepted by a user community. Constraint systems typically have been cursed by a combination of a limited underlying solver and an extremely descriptive constraint specification language, making it possible for the user to specify a multitude of problems that the underlying satisfaction system has little chance of solving. Declarative programming languages determine the amount of rope with which a user of the language can hang himself, and constraint system designers must be careful to limit the user's expressive power to the set of problems for which their system can provide solutions. Constraint systems in general have difficulty with over-constrained problems, under-constrained problems, and problems that exhibit mathematical degeneracies.

These problems may begin to clarify why very few constraint systems have made much progress beyond the work done by Ivan Sutherland 30 years ago and why constraints have failed to gain significant popularity within the graphics community. Interactivity requires exceptional performance, predictability, and generality, yet constraint systems are consistently plagued by poor performance, numerical instability, and the inability to handle many general cases. There is, however, reasonable hope that with proper attention, these problems can be overcome, and the advantages that constraints offer can be realized in modern interactive graphic applications. It is the purpose of this report to characterize many of these problems and recommend solutions where such solutions have been found.

2 Theory of Planar Linkages

Linkages have been used for centuries to alter motion. An input motion, perhaps the continuous rotation of a waterwheel, an engine, or a windmill, is converted into some presumably more useful form of motion. For example, using the very simple 4-bar mechanism shown in figure 3, rotation of the input crank is converted to an oscillatory motion in the output rocker. A tool may be attached to the output link, perhaps a punch or a drill, and a motor to the input link, and we have created a useful, though simple, mechanism.

Linkage designers are interested in many aspects of motion. Not only is the path of a particular output link important, but the complete kinematics and dynamics of the linkage may be significant to the usefulness of a

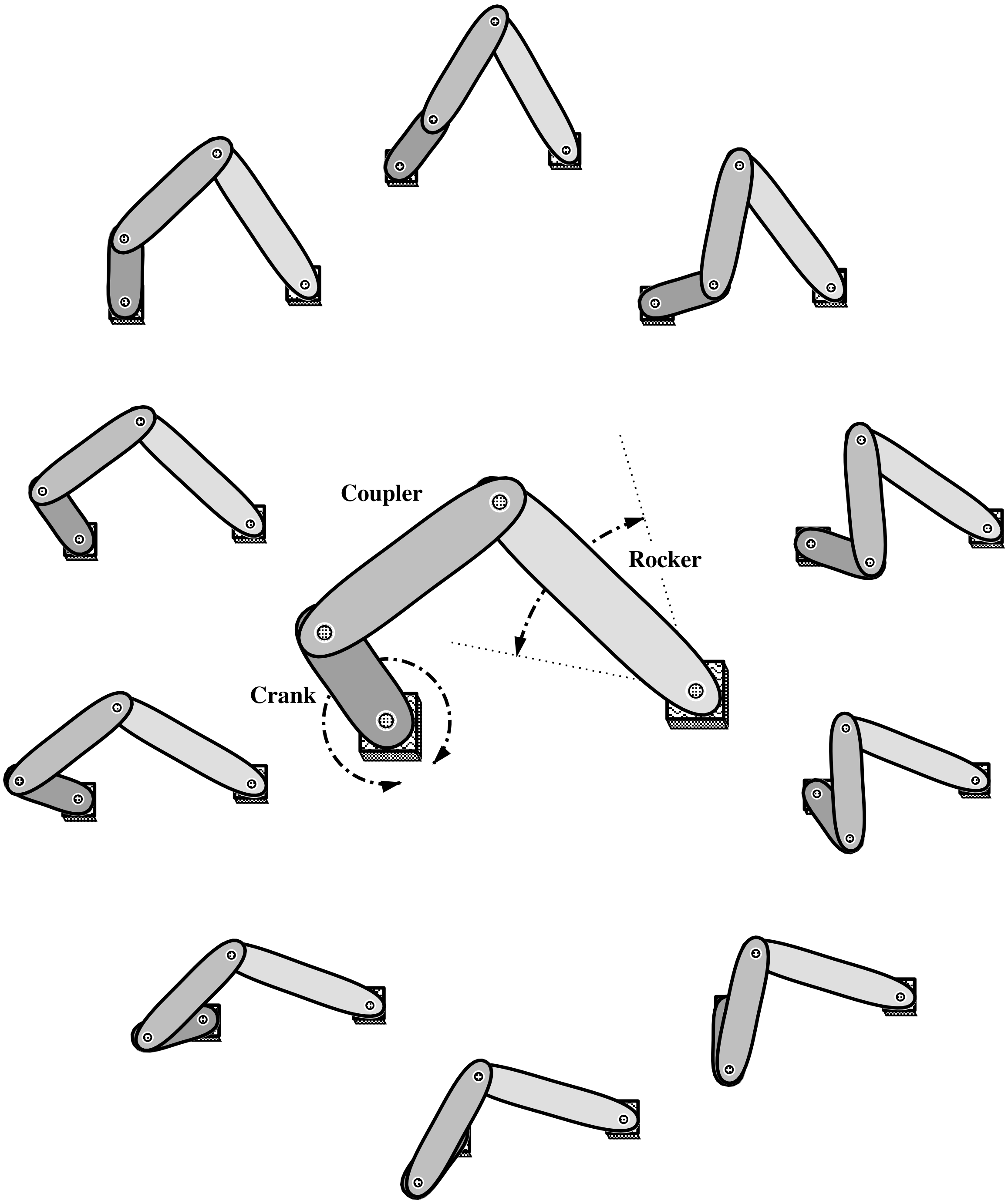


Figure 3: The classic 4-bar crank-rocker mechanism and its motion.

mechanism. Kinematics is a study of position with respect to time without consideration to mass and force, and it is this aspect of mechanisms on which this report will be focused. Dynamics is a broader study which includes mass and force and, in turn, notions such as momentum and energy. While simulation and animation of bodies and linkages in the context of dynamics is an interesting and ongoing topic of research, it is not one that will be considered here. Thus, our study involves primarily the positions of bodies in space and the change in these positions with respect to time.

2.1 Links, Joints, and Mobility

To understand how linkages relate to geometric constraint systems, it is useful to make some effort to equate the vocabulary of one discipline to that of the other. There is a straightforward correlation between, for example, the apparent degrees of freedom of a linkage and the number of independent equations that define its corresponding system of constraints. While the correlation is straightforward, algebraic insights are frequently overlooked by those attempting to simulate mechanisms and, in turn, some relatively simple concepts have been reinvented under many titles, and others appear to have been overlooked entirely. The relationship between planar linkages and the underlying algebraic systems will be emphasized throughout this report in the hope that the opportunity to reapply this work to general constraint-solving systems is not misplaced in the language of mechanisms.

2.1.1 Links and Rigidity

Linkages are constructed from rigid bodies connected by joints. A rigid body, by definition, is a set of points that are stationary with respect to all other points in the set. More concretely, every pair of points in a rigid body maintains a constant separation in space. A rigid-body transformation is a motion of points in the body that preserves these distances. Intuitively, the two rigid-body transformations are rotation and translation.

There is one other transformation that preserves distances but which is not considered a rigid-body transformation. A body may be mirrored in space, as indicated in two dimensions in figure 4. As can be seen, distances are preserved, but such mirroring cannot be effected by a physical motion of the body through two-dimensional space. A mathematical definition of rigid-body motion must contain both the distance constraint and a constraint that prevents mirroring; however, the mirroring constraint adds little more than mathematical complexity to the current discussion and will thus be ignored.

When defining linkages, one might choose to start by defining distance constraints between points on a rigid body. The triangular link in figure 5(a)

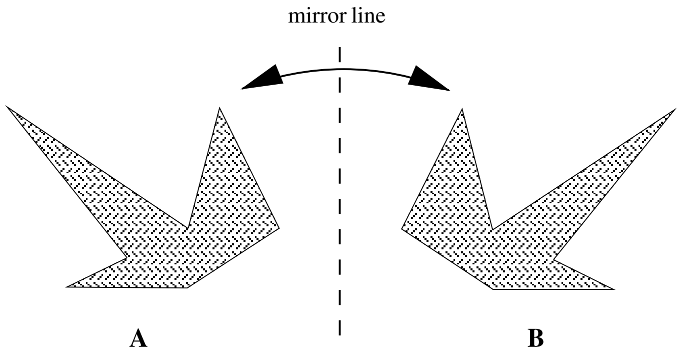


Figure 4: Distances are preserved when a body is mirrored but this does not constitute a rigid-body transformation. Notice that there is no way to move the link from its position in configuration A to its position in configuration B without lifting the link out of the plane.

for example, might be defined by three distance constraints as indicated. The quadrilateral link in figure 5(b) could be defined by as many as six distance constraints. Clearly, the more complex the geometry of an object, the more constraints can be defined specifying the distances between pairs of points.

Consider the complexity of the algebraic system created by defining rigid bodies in this manner. Each point in the system adds two variables, an x and a y coordinate. Each distance constraint adds one equation of the form $(x_1 - x_2)^2 + (y_1 - y_2)^2 = c_{12}^2$. For any set of n points, there are $\binom{n}{2}$ possible distance constraints. It is an interesting exercise, although not one taken on here, to prove that all but $2n-3$ of these possible constraints are redundant. Thus, a system of constraints describing a rigid body defined by n points can be reduced to a system of constraints defined by $2n-3$ independent distance constraints.

A more practical abstraction than that of points constrained by distance equations is to consider a rigid body to be a coordinate system and then to speak of the freedoms of that coordinate system in space. This leads to the notion of degrees of freedom of a rigid body. In two-dimensional space, a rigid body has three degrees of freedom; that is, to fix a body in two-dimensional space relative to a world coordinate system requires a minimum of three values. For example, we might specify the position of a body by specifying the location of its local coordinate system's origin in world coordinates and the angle between its x-axis and the x-axis of the world coordinate system. This particular representation of the three degrees of freedom, involving an offset from the origin and a rotation, leads to the notion that a rigid body in two dimensions has two translational degrees of freedom and one rotational degree of freedom (figure 6). In many ways, this is misleading since there is no inherent difference between a translational and rotational degree of freedom

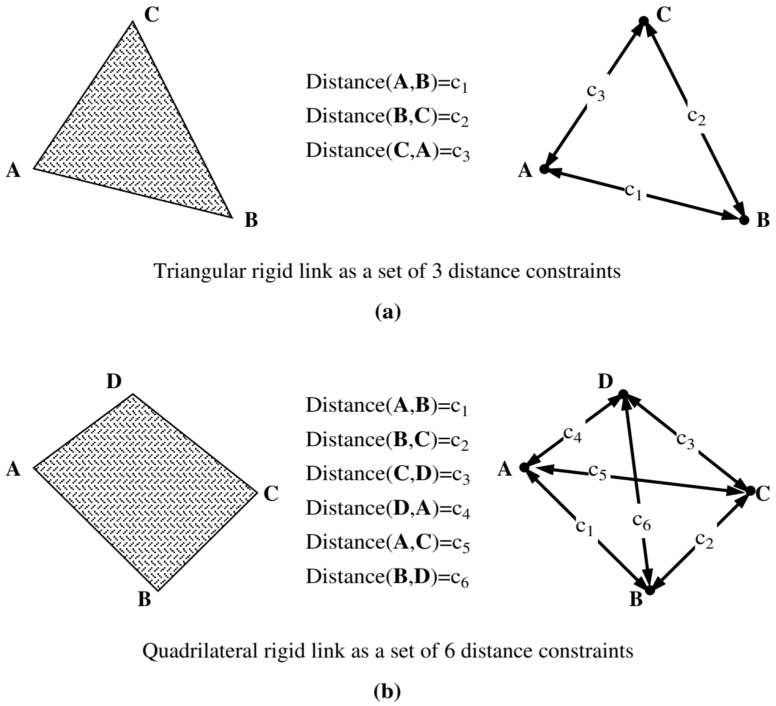


Figure 5: Rigid bodies as sets of distance constraints. Notice that any five constraints of the quadrilateral are independent, but the sixth is always redundant.

as the choice of the representation of position in space is arbitrary; however, these terms are commonly used and are helpful in conveying an intuitive understanding of degrees of freedom.

Rigid bodies are a practical abstraction because they reduce the complexity of the underlying constraint system. Rather than individually representing all of the n points and $2n$ freedoms that define the geometry of a body, we can take advantage of the fact that, in two dimensions, any number of consistent non-zero distance constraints between two or more points can, by change of variable, be reduced to a system of equations in three unknowns corresponding to the three degrees of freedom of the entire rigid body. From these three free variables, the locations of each of the individual points can be calculated directly and rapidly. Thus, if some set of points is defined or constrained to be rigid, it is to the advantage of a constraint solver to recognize that only three degrees of freedom exist for the entire set, regardless of the number of freedoms that individual points within that body appear to have or the number of individual constraints that define this rigidity. As will be shown later, the performance of a constraint-solving system is partially limited by the number of variables and number of equations for which the system must find solutions. Identifying rigid bodies and reducing the system

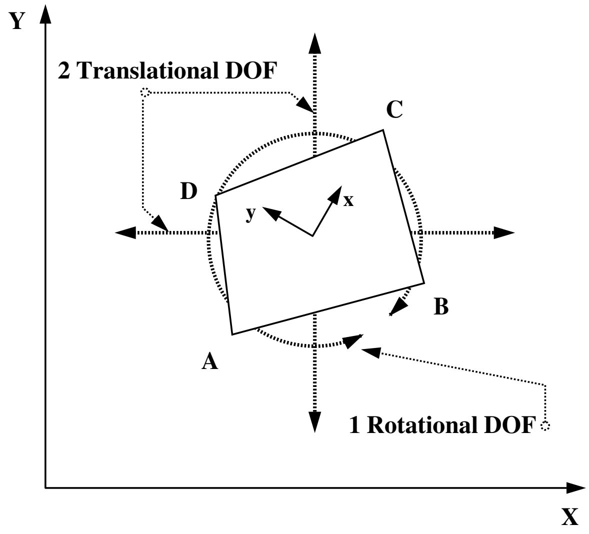


Figure 6: Three degrees of freedom of a rigid body in the plane.

of constraints accordingly reduces complexity substantially and thus is at the heart of most closed-form solvers.

2.1.2 Joints

Joints connect rigid bodies and, in turn, constrain the relative motion of the links that they connect. A revolute joint (or hinge), for example, prevents relative translation between two bodies. The relative motion of the two bodies is limited to rotation about the axis of the hinge. A prismatic joint (or slider) allows translation along a fixed straight line, but no relative rotation.

Generally speaking, a joint is the area of interaction where two links come into moving contact with one another. A distinction is generally made between two classes of joints: the lower pairs and the higher pairs. The lower pairs maintain *surface* contact throughout the motion of the joint, whereas the higher pairs maintain only *point* or *line* contact throughout the motion. Also, in the lower pairs, this surface of contact is fixed in location relative to one or both of the links. A revolute joint or hinge maintains contact along a surface of revolution between the two links and is thus a lower pair (figure 7). Similarly, a prismatic joint maintains a prismatic surface of contact across the pair and is a second lower pair. Cams, gears, tackle, and slotted pins are all examples of higher pairs in which a surface of contact is not maintained during motion. Lower pairs have certain properties that make them more practical in design, manufacture, and function ([PHIL90],[REUL76]). There are exactly six lower pairs in three dimensions (figure 8), but only two, the revolute and prismatic, in two dimensions, and it is these that are simulated in MechEdit.

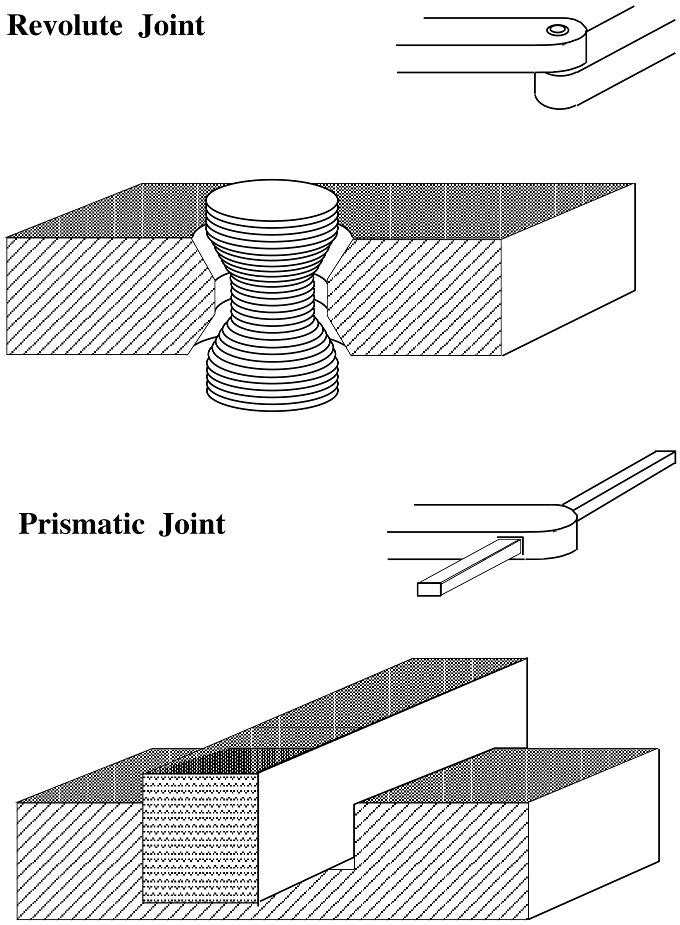


Figure 7: The two-dimensional lower pairs and their surfaces of contact.

It is instructive to examaine these two lower pairs and understand their effect on the relative degrees of freedom of the connected links. We will also show how the relative motion defined by these joints can be represented algebraically.

Joints constrain relative freedom between two bodies. A pair of links in two-dimensional space has three relative degrees of freedom. The relative degrees of freedom between two bodies are counted by taking one link to be fixed and observing the mobility of the remaining link. Both revolute and prismatic joints remove two relative degrees of freedom.

A revolute joint may be represented geometrically as the coincidence of two points on two rigid bodies. Algebraically, this corresponds to equating the x and y coordinates of corresponding points on two rigid bodies. Clearly by equating two independent values, two relative freedoms are removed. Since rigid bodies are represented by a local coordinate system, and their placement in space as a Δ_x and Δ_y offset and Θ rotation, the location of some point $P = (a, b)$ in rigid body I's local coordinate system is computed

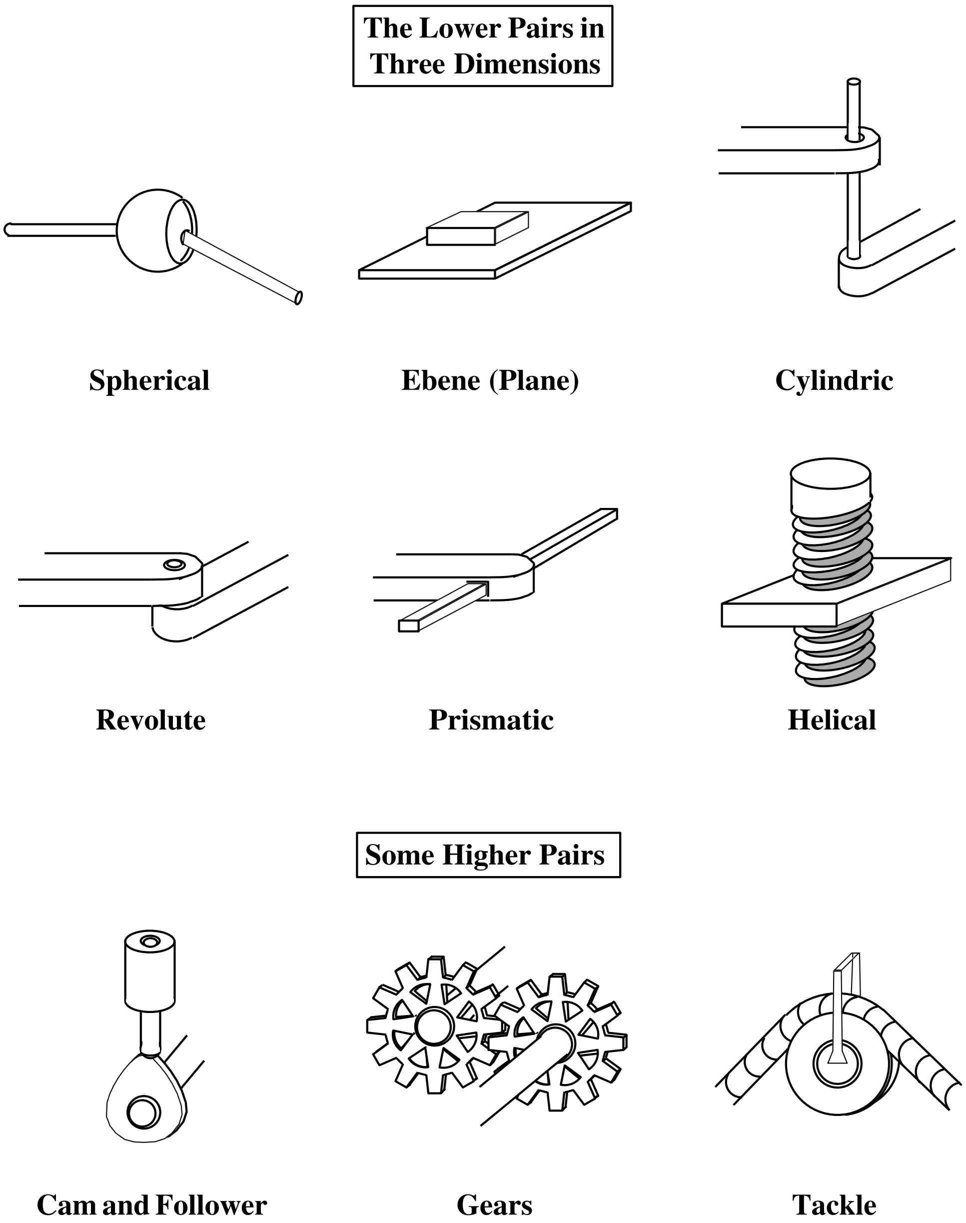


Figure 8: Lower v. Higher Pairs

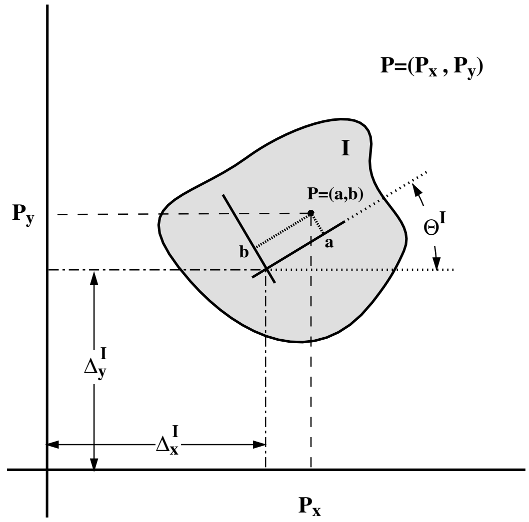


Figure 9: A point on rigid body I represented in local and global coordinates.

as:

$$\begin{aligned} P_x &= \Delta_x^I + a\cos\Theta^I \\ P_y &= \Delta_y^I + b\sin\Theta^I \end{aligned}$$

where the superscript I indicates a degree of freedom of rigid body I (figure 9). A revolute joint is algebraically represented as the equating of two such points, for example, if we form a revolute joint between point $P = (a,b)$ of link I and point $Q = (c,d)$ of link II, where link I and link II are parameterized by $(\Delta_x^I, \Delta_y^I, \Theta^I)$ and $(\Delta_x^{II}, \Delta_y^{II}, \Theta^{II})$, respectively (figure 10(a)), a revolute joint would be represented by (figure 10(b)):

$$P_x - Q_x = (\Delta_x^I + a\cos\Theta^I) - (\Delta_x^{II} + c\cos\Theta^{II}) = 0$$

$$P_y - Q_y = (\Delta_y^I + b\sin\Theta^I) - (\Delta_y^{II} + d\sin\Theta^{II}) = 0$$

A prismatic joint may be represented geometrically as the coincidence of two lines on two rigid bodies (figure 10(c)). A line on a rigid body may be represented by a point through which that line passes, $P = (a,b)$, and an angle, α. To equate such a line on rigid link I and another line on rigid link II represented by a point $Q = (c,d)$ and an angle, β, we may use the following equations:

$$\begin{aligned} \arctan\left(\frac{P_y - Q_y}{P_x - Q_x}\right) &= \alpha + \Theta^I \\ \alpha + \Theta^I &= \beta + \Theta^{II} \end{aligned}$$

The second equation states that the two angles in global coordinates are equal. The first equation states that a vector from a point on one line to

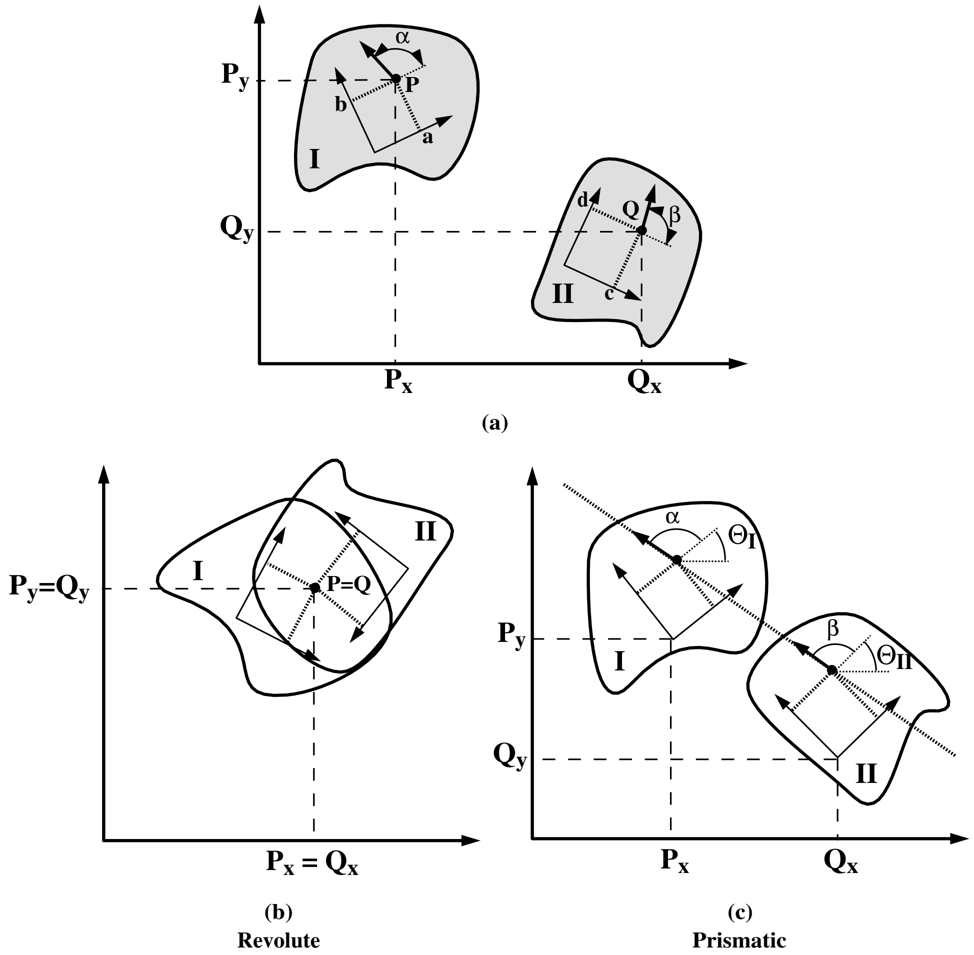


Figure 10: The algebra necessary to represent the lower pairs in two dimensions.

a point on the other line must lie in the line. An implementation of this formulation must insure that if $P_x - Q_x = 0$, the inverse tangent computes the angle of the vector correctly. Wtih two such independent equations, it is clear that prismatic joints also remove two relative degrees of freedom.

2.1.3 Mobility

The number of degrees of freedom of a linkage, also known as the linkage's mobility, is the number of input values that must be supplied to fully constrain the position of all links in the mechanism. The classic 4-bar linkage (3 moving links and a fixed ground link) has one degree of freedom (figure 3). Each of the three moving links has three degrees of freedom giving a total of nine freedoms. These are constrained by four joints, each removing two degrees of freedom for a total of eight, leaving the mechanism with a single degree of freedom. Therefore, in order to fully specify the position of the

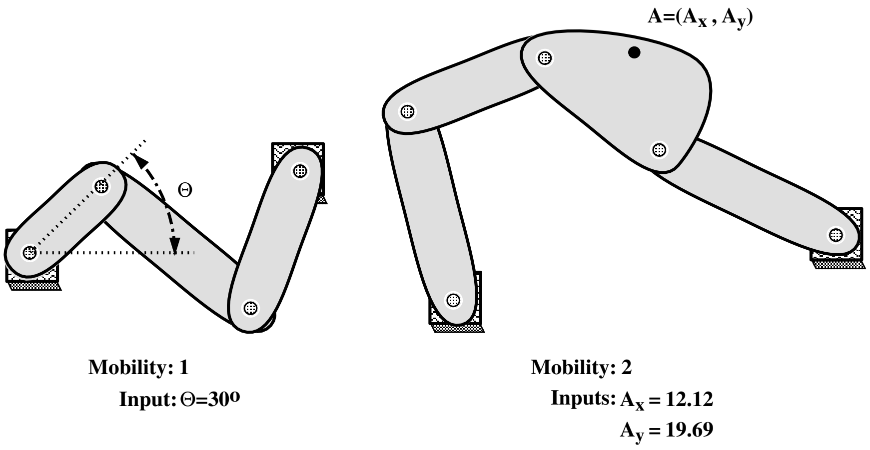


Figure 11: A 4-bar linkage requires one input, θ to fully constrain its position. A 5-bar linkage requires two input constraints, for example the x and y coordinates of point $\mathbf{A}$.

linkage, a single input value must be given. This might be given by specifying the angle between links at one of the revolute joints. If, for example, the crank of a crank-rocker mechanism is driven by a motor, the motion of the linkage is fully determined. A five bar mechanism can be fully constrained by specifying the location of a point on one of the two couplers. This provides two input values, an x-coordinate and a y-coordinate, and thus fully constrains the five-bar's two degrees of freedom (figure 11).

The fact that both lower pairs in two dimensions remove two relative degrees of freedom, combined with the observation that every rigid body intrinsically has three degrees of freedom leads to a very simple formula, known as Grübler's formula, for computing the degrees of freedom of a planar linkage:

$$D = 3(L - 1) - 2J$$

D represents the number of degrees of freedom of the mechanism, L is the number of links including a fixed ground link which contributes zero degrees of freedom (hence the minus one), and J is the number of joints.

Algebraically, an independent variable is a degree of freedom. Each independent variable in a system of constraints adds one degree of freedom. A constraint is an independent equation. A system is fully constrained, or has zero degrees of freedom, when the number of independent variables is equal to the number of independent equations. The number of degrees of freedom of a system of constraints is the number of independent equations subtracted from the number of independent variables defining the system. Where above we spoke of additional input values required to fully constrain a system, we can now see these additional 'input' values to be simply additional constraints that set the value of a particular variable to a constant.

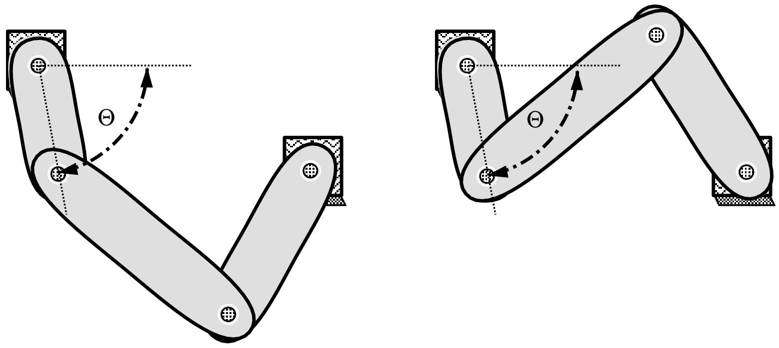


Figure 12: Two solutions for given input θ corresponding to the two real roots of the system of equations

When a linkage has as many input parameters as it has degrees of freedom, the linkage is said to be well-constrained. Algebraically, this corresponds to having an equal number of independent equations and unknowns. A well-constrained mechanism has a finite number of positions that satisfy both the constraints imposed by the geometry of the links and joints and the constraints imposed by the inputs. If there are multiple solutions to the constraint system, these solutions are sometimes called *branches*. Figure 12 shows two possible configurations of a 4-bar mechanism driven from one of its ground sites. Both configurations have equal driving angles, but as can be seen, there are two valid solutions. Each of these solutions is a real root of the constraint and input equations. The presence of multiple real roots requires that the constraint-solving system insure a consistent selection of one of these roots. This may be difficult if unstable iterative numerical techniques are used by the solver.

A linkage with fewer input parameters than degrees of freedom is said to be under-constrained. An under-constrained mechanism has an infinite number of solutions that satisfy both the inputs and the constraints imposed by the structure of the mechanism. Algebraically this corresponds to having fewer independent constraint equations than unknowns. Since the position of the mechanism for a given input cannot be determined kinematically, the constraint system must either report this situation or select one of the infinite number of positions of the mechanism to display. While most useful mechanisms are well-constrained, the ability to animate under-constrained mechanisms is extremely useful in the design process. As a mechanism is assembled, it passes through many states that are not fully constrained. A user may desire to adjust each of the degrees of freedom of such a mechanism individually, not simultaneously. Also, a user may simply wish to explore how the addition of individual constraints restricts and alters motion. These restrictions may not fully determine the position of the mechanism but may be helpful for editing or visualization. To create an acceptable imitation of a

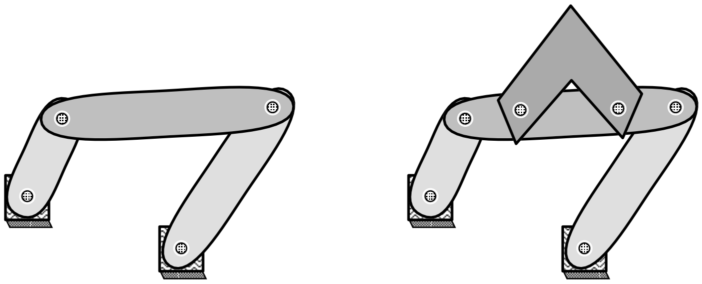


Figure 13: Over-constraint and genericity. The holes for the revolute joint in the coupler must be precisely aligned with the holes in the V-shaped link. Any random epsilon change to these holes will make the linkage unconstructible.

physical workbench, a geometric-constraint solver must provide some facility for handling under-constrained systems.

If a linkage has more independent constraint equations than variables, the linkage is said to be over-constrained. Generically, over-constrained systems have no solutions; however, it is possible to create over-constrained systems that do have solutions by carefully defining the equations. Typically, this occurs by constructing the over-constrained system in its solved position. The mechanism in figure 13 shows a well-constrained linkage to which is added a link and two joints. The link adds an additional three freedoms and the two joints remove a total of four freedoms. By Grübler's formula, the mobility of the system is zero; however, it is clear that the linkage can move as before. This particular system is over-constrained but is not generic.

2.1.4 Genericity

Genericity is a quality that is difficult to specify precisely. A linkage is described as generic if a random epsilon change to any joint site on any of the links would not affect the ability to assemble the mechanism and would not change its mobility. The previous example is non-generic because an epsilon change to one of the joint sites on the V-shaped link could make it impossible to assemble the linkage. One could assemble the linkage only by carefully selecting the position of the joint sites on the V-shaped link to match those of the coupler. Another classic example of non-genericity is the appearance of coincident points or of parallel lines. The linkage in figure 14(a) becomes non-generic when point A becomes coincident with point B. When this occurs, the mobility of the linkage increases by one and the pair of links is free to rotate around the ground joint. Similarly, the linkage in figure 14(b)

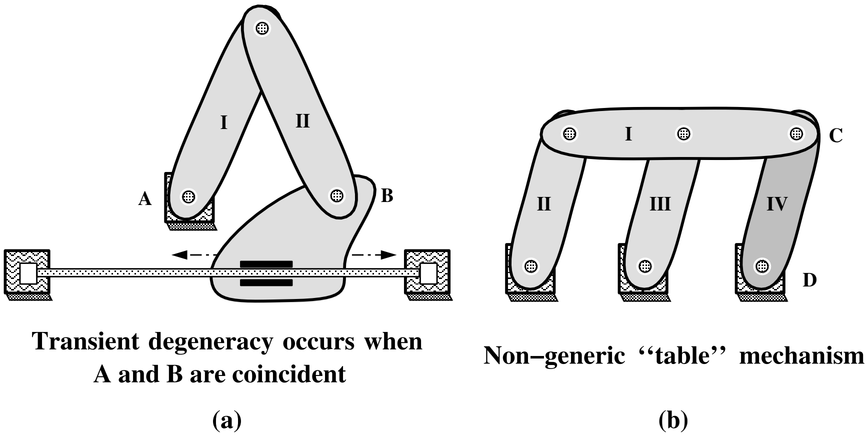


Figure 14: Non-generic mechanisms

is non-generic because of the parallelism of three links. While by Grübler's formula, the mobility of this linkage is zero, the apparent mobility is one. In both of these linkages, if the positions of the joints were offset by some random epsilon, they would become generic, and Grübler's formula would once again predict the mobility correctly. Over-constrained mechanisms that still exhibit positive mobility are non-generic.

Algebraically, genericity has long been discussed and is closely related to the concept of independence. A system of independent equations corresponds to a generic system of constraints. Independence implies that one equation in a system is neither an implication of the other equations, that is, it cannot be derived through proper combination of other equations in the system, nor a contradiction. In the case of the "table" mechanism in figure 14(b), the additional constraints implied by joints C and D are not independent. The location of joint C will be the same regardless of the existence of link IV and joint D. The position of joint C is determined by links I, II, and III, thus the equation that constrains the position of joint C imposed by link IV is redundant with the system of constraints imposed by the submechanism consisting of links I, II, and III. Genericity can be quantified algebraically by stating that a system is generic if all of its constraint equations are independent. Genericity can be forced by adding a random epsilon to each of the constraint equations.

2.1.5 Breaking and Branching

Even well-constrained linkages may not have any real solution for some particular set of inputs, perhaps even all inputs. Intuitively, the non-existence of solutions can be physically interpreted as the breaking of the mechanism. Values of the input parameters beyond which the system of constraints defin-

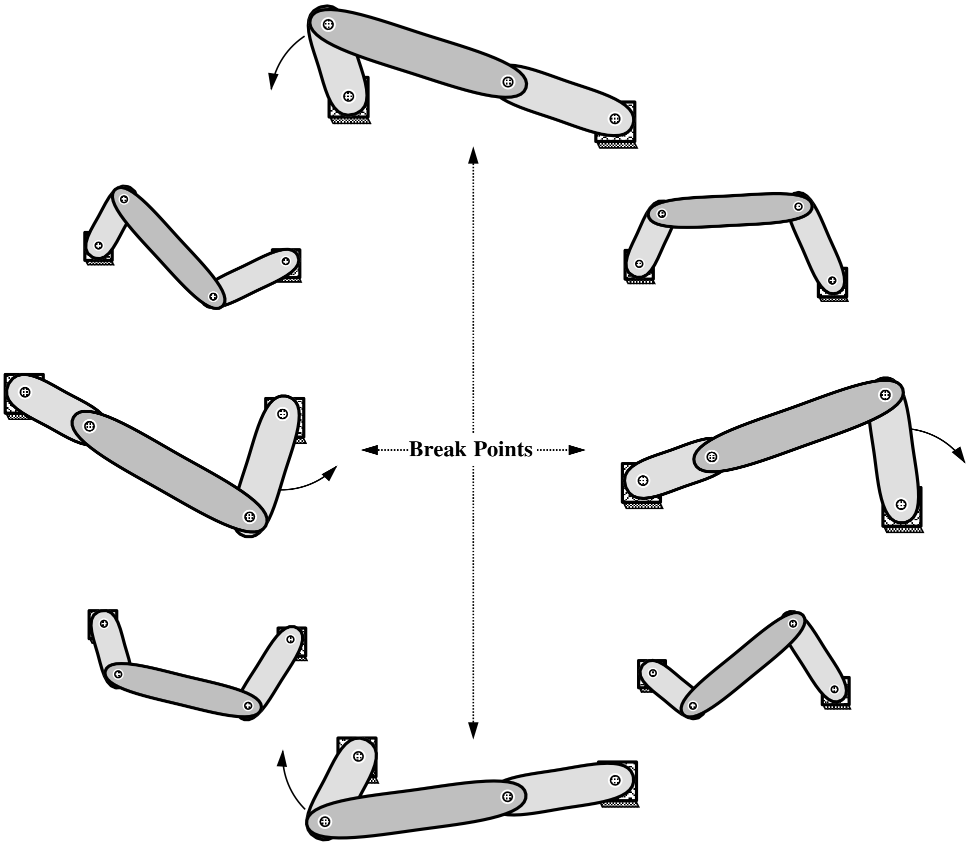


Figure 15: The break points of a rocker-rocker mechanism. If the driving input constraint were to continue moving along the arrows as indicated, the underlying constraint system would become unsatisfiable, and the linkage would break.

ing the linkage cannot be satisfied are termed *break points*. The four-bar rocker-rocker mechanism in figure 15 is shown at its four break points. Moving the input joint beyond the range of motion indicated would cause the system of constraints to be unsatisfiable and the mechanism to break. Identifying such breakpoints is a non-trivial task, particularly in cases where closed-form solutions to the system of equations do not exist. Once again, a useful geometric constraint system must respond to this situation in some way, either by relaxing some of the constraints, halting the motion at the break point, or some other reasonable response.

One final concept that should be mentioned is the concept of branching in a well-constrained linkage. At some break points, two or more distinct solutions to a system of equations converge; physically, the two branches of the linkage share a common position. The four-bar example in figure 16 is at such a position. Clearly upon returning from this position to some intermediate state, the choice of mode, that is, whether the coupler angles up or down,

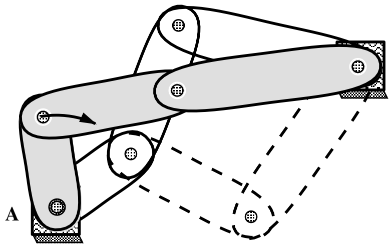


Figure 16: Returning from a limit point, two branches exist for possible motions. The limit point is a double root of the system of equations.

is non-deterministic. In fact, driven from ground joint A, any motion at all would require infinite force and is thus impossible. In a kinematic system however, some choice as to which branch is to be taken must be made. As the break point of the mechanism is approached, the two distinct solutions representing the two different modes converge. In turn, numerical instability is particularly likely near break points.

2.2 Summary

Linkages make an interesting application of constraint-based systems precisely because of the simple correlation between the motion of the linkage and the underlying algebraic system. Mechanism mobility corresponds neatly to the dimension of the solution space of the algebraic system. Genericity corresponds to algebraic independence. Branches of the mechanism correspond to separate solutions in a finite solution space. Double roots manifest themselves as branch points of the mechanism. Finally, points at which a linkage changes its mobility correspond to points at which the underlying system exhibits a degeneracy. This clean correspondence lends substantial insight into the kinematic analysis and, as will be seen, most of the methods for such analysis that have been presented in the literature have similar correspondences to well-known algebraic techniques.

3 Constraint-System Solving

Efficient constraint-system solving presents significant research challenges and essentially will determine the efficacy of constraints in interactive environments. Algebraically, solving a system of constraints refers to finding a set of real values for the unknowns in the system that satisfy the set of constraint equations. This is equivalent to finding the roots of a system of equations, or in the context of mechanical linkages, locating positions for all of the links so that they satisfy the input parameters and do not cause the mechanism to break. While in some applications, it is desirable to locate all of the real roots, we will only seek to locate a single solution, in particular, a solution that is, in some sense, desirable. Normally, a desirable solution is one which is closest to the previous position of the linkage. In the context of interactive animation of mechanisms, the inputs are varied continuously, for example by a mouse, and thus we seek a series of solutions that follow a smooth, continuous path. Because of the desire for interactivity, such solutions must be found in a fraction of a second, ideally at the frame rate of the display.

In the context of interactive graphics, there are also a number of other challenges that have been mentioned previously. Over- and under-constrained systems must be handled robustly, special considerations must be made for transient degeneracies, and the system must respond in a meaningful way to systems of equations with no real solutions. Each of these situations, while perhaps avoidable through careful specification of the system in non-interactive applications, appears quite frequently in interactive settings. When animation is controlled by the mouse, a constraint solver may be required to solve as many as 30 constraint systems per second to keep up with frame rate. With the power to construct 1800 systems of equations per minute, a user will inevitably stress the ability of the solver to handle each of the above challenges.

Since Sutherland's "Sketchpad", numerous techniques have been developed for constraint solving. It is useful to consider the limited progress in solvers that has been made during the last 30 years, examine the advantages and shortcomings of each technique, and finally show how proper combination of these techniques leads to a robust and relatively efficient system. Also, we will discuss how to use standard numerical techniques to handle under-constraint and degeneracies.

3.1 A 6-bar Example

To make the discussion of various methods for constraint solving more concrete, the 6-bar linkage shown in figure 17 will be used as an example. This particular mechanism is interesting because the ability to solve the underlying constraint system in closed-form is dependent upon which ground joint is driven as an input. As will be discussed below, when driven from ground joint A or C, the system can be solved analytically, but when driven from ground joint B, the solver must resort to an iterative numerical method.

This 6-bar mechanism is generic and by Grübler's formula can be seen to have a mobility of one. Links I, II, and III together with the ground link form a simple 4-bar crank-rocker mechanism. Link I is the crank, that is, it can be rotated through a complete circle without breaking the mechanism.

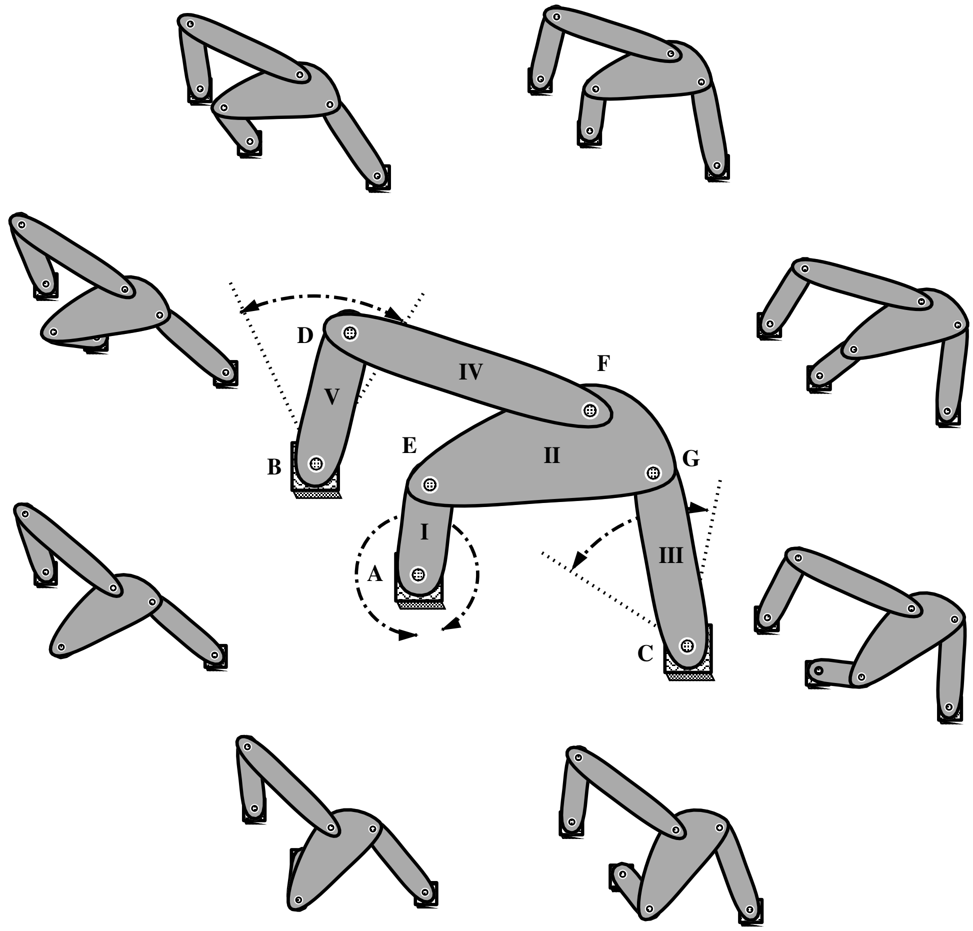


Figure 17: A 6-bar mechanism and its motion when cranked at joint A.

Links III and V both have a limited range of motion which is indicated in the figure and will oscillate through this range if link I is driven through 360 degrees. The figure shows selected frames from an animation of this motion.

3.2 Local Propagation

An extremely simple method for solving constraint systems is that of local propagation. A very limited class of systems may be solved completely using this technique; however, local propagation is necessary at some level in every constraint solver. Algebraically, the method begins with a set of equations in a set of unknowns. When the value of one of the unknowns is provided as an input, the value is propagated to all constraints that depend upon that unknown. Having substituted the value into certain constraint equations, the system now examines each of these equations to see if only one unknown remains. If so, the equation is solved and the value of the unknown found. This newly acquired value is then propagated in the same manner. The process is repeated until no more constraints can be solved or all unknowns are found (figure 18).

Local propagation is extremely limited as it is incapable of examining two constraints simultaneously. Thus if two equations have interdependencies as do the simple line equations

$$\begin{aligned} y - x &= 5 \\ y + x &= 7 \end{aligned}$$

local propagation will be unable to solve the system. Local propagation is only effective for systems of equations that can be rearranged into triangular form.

An algebraic system is considered triangular if the system can be solved using only backsubstitution. Mathematically, an ordering of a system of n consistent, independent equations $E_1, ..., E_n$ in a set of n unknowns, U, is triangular if each E_k is dependent on some $U_k \subseteq U$ such that $|U_k| \leq k$. Thus, the first equation in the ordering must be of the form $x_1 = c_1$ where x_1 is a variable and c_1 is a constant. Equations of this form are called input constraints. The second equation might be $x_1^2 + x_2^2 = c_2$. Notice that the second equation, E_2, has two variables, x_1 and x_2, of which only one, x_2, cannot be determined from the first $k - 1 = 1$ equations. Thus, backsubstitution and simple algebra allows a value to be assigned to x_2. A third equation could contain up to three variables x_1,x_2, and x_3 (figure 19). Since x_1 and x_2 are now known, x_3 can be found easily. This is a simple extension of the familiar notion of an upper triangular matrix in linear algebra. Because of the ease and speed with which a triangular system of equations can be solved, mechanisms that identify when this is possible and construct the necessary ordering are extremely important in constraint systems.

Local propagation can be implemented in two phases, a planning phase and an execution phase. The planning phase identifies the ordering on the system of equations that places it in triangular form. During the execution phase, the constraint solver performs the backsubstitution necessary to solve the system of equations. Notice that if the set of constraints remains constant with changes being made only to the input constraint, the planning phase need not be repeated.

Local propagation is an inherently one-dimensional method when considered geometrically. In two dimensions, the simplest unknowns are points which, of course, have two degrees of freedom, say an x and a y value. All interesting relationships that can be specified between sets of points, such as distance, colinearity, or cocircularity, will inherently relate these x and y values simultaneously. Except for the simplest task of setting one of these values to a constant, two-dimensional relationships involve the simultaneous solution of multiple constraints on at least two unknowns, surpassing

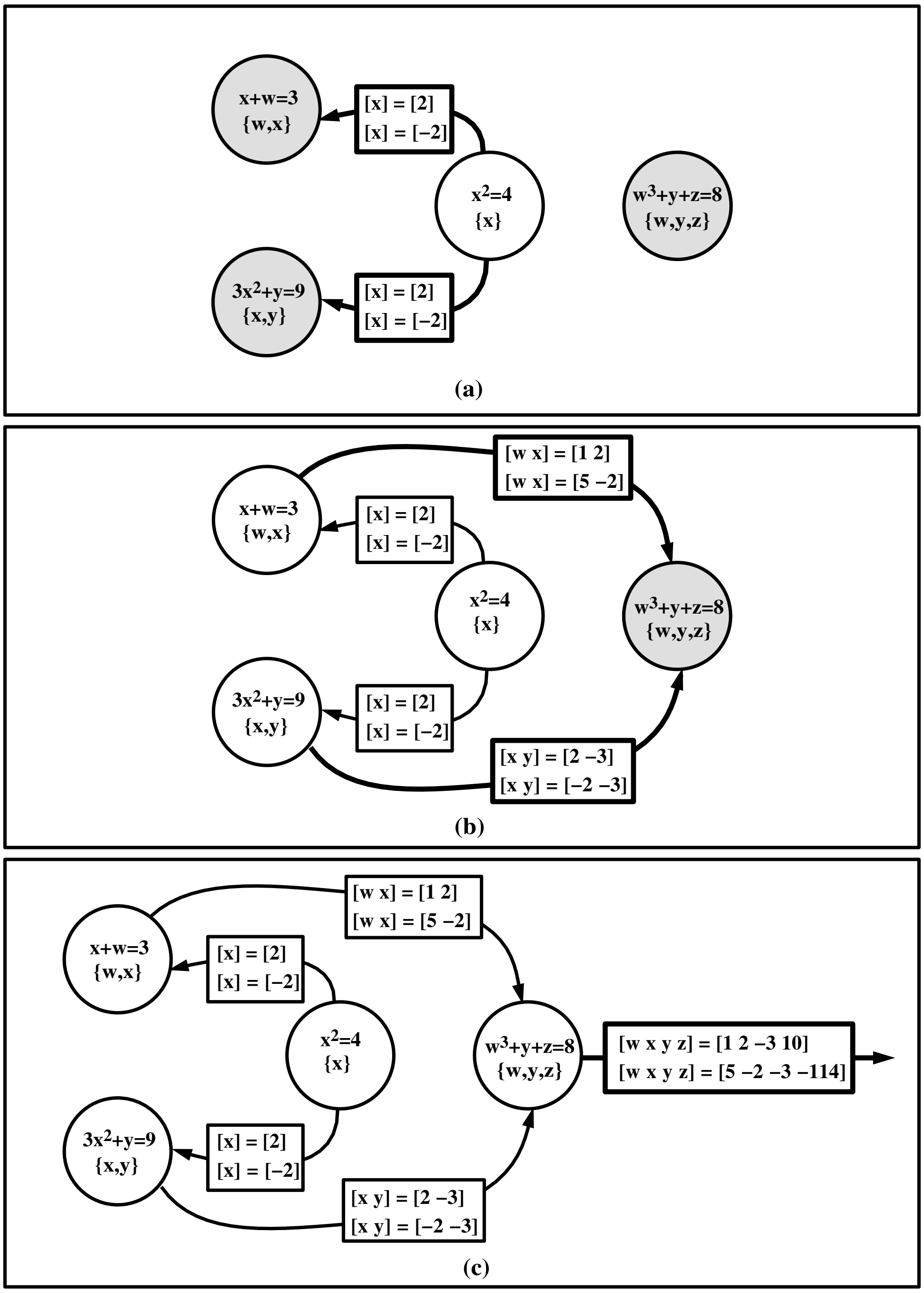


Figure 18: Using propagation to solve a system of four equations in four unknowns. The input equation, $x^2 = 4$, causes a sequence of propagation steps to occur incrementally solving for the vector of unknowns.

$$
\begin{array}{r}
x_4 + 3x_3 + 5x_2 - 2x_1 = 8 \\
-x_3 + 2x_2 + 5x_1 = 2 \\
3x_2 + 2x_1 = 1 \\
x_1 = 5
\end{array}
\qquad
\begin{array}{r}
x_4^2 + 2x_3^3 + 5x_2^2 - 4x_1 = 0 \\
x_3^2 + \quad\quad x_1 = 3 \\
x_2^3 \quad\quad = 5 \\
x_1^2 = 2
\end{array}
\qquad
\begin{array}{lllll}
x_4 & & & & = 3 \\
 & x_3 & & & = 2 \\
 & & x_2 & & = 1 \\
 & & & x_1 & = 0
\end{array}
$$

(a) **(b)** **(c)**

Figure 19: Triangular Systems

the capacity of local propagation to solve the system. Because local propagation inherently does not support two dimensional constraints, graphical applications must resort to more complex algorithms.

3.3 Contraction

3.3.1 Method

A method presented many times in the literature with slight variations in name and description involves constructing and examining a symbolic graph of the mechanism and identifying how a solution can be built up constructively in closed form. Enderton [ENDE90], Kramer [KRAM92], and recently Romdhane [ROMD92] have each developed similar algorithms for finding solutions in this manner. Algebraically, all of these methods, like propagation, may be seen as finding an ordering for the system of constraints that makes the algebraic system, in some sense, triangular.

Contraction, also known as the method of dyads, when considered algebraically, is a relatively straightforward extension of propagation. Contraction involves two phases, a planning phase in which a solution path through the mechanism graph is determined and an execution phase during which this plan is used to assign values to unknowns. These correspond to the processes of ordering the equations in a semi-triangular form and performing the backsubstitution, respectively. The name *contraction* is derived from the idea that rigidity is propagated through the mechanism graph, and after each discrete propagation, the mechanism graph may be reduced in size. Thus the mechanism graph is iteratively contracted to a single node. Starting with the rigid ground link, the location of mobile links are calculated by examining pairs of equations. After the location of a link is established, it may be regarded as part of the fixed ground link, thus contracting the mechanism graph. As two links are fixed relative to one another in space, they become a single rigid body. The two links' six degrees of freedom can thus be reduced to the three degrees of freedom inherent in a single rigid body. This process is demonstrated on the 6-bar mechanism of figure 20.

3.3.2 6-bar Example

If the 6-bar mechanism is driven from joint A, contraction may be used to position the mechanism in space. The initial mechanism graph has 6 nodes, representing each of the 6 links including ground, and edges betweeen these nodes as shown in figure 20(a). The addition of the input constraint fixes the angle of link I. Together with the constraint imposed by joint A which fixes the position of the axis of joint A, this fully constrains the position of link I. Thus, the three degrees of freedom associated with link I may be fixed, and link I may be viewed as part of the rigid ground link. In the mechanism graph this is represented as the contraction of link I and the ground link into the single node Fix_1 (figure 20(b)).

With link I fixed, thus fixing the location of joint E, the interaction of links II and III may now be solved. The position of joint E constrains joint G to lie on a circle centered at E and of radius determined by the fixed distance between E and G. The position of joint C constrains G to lie on a circle centered at C and of radius determined by the fixed distance between C and G. With these two constraints, the two degrees of freedom of point G are fully determined by the intersection of two circles (figure 20(c)). In turn, the location of joint G combined with the locations of joints E and C fully constrains the positions of links II and III.

In terms of contraction, a loop consisting of three nodes and three edges may be contracted to a single node as is shown in the transition from figure 20(b) to figure 20(c). The mechanism graph now consists of a single loop of three nodes and may thus be contracted in a similar way (figure 20(d). After this last contraction, the graph is a single fixed node indicating that the position of the mechanism is fully determined (within a finite number of discrete branches).

3.3.3 Application

The primary difference between propagation and contraction is the number of nodes that are eliminated simultaneously. Algebraically, propagation is simple backsubstitution which solves a single variable at a time; contraction requires slightly more complicated algebra and the use of the quadratic equation to simultaneously solve two equations. With propagation, a single constraint is used, in combination with all of the currently known variables, to solve for a single unknown, contracting two nodes and one edge into a single node. The presence of a single input value at joint A allowed the contraction of nodes G and node I in the transition between figure 20(a) and

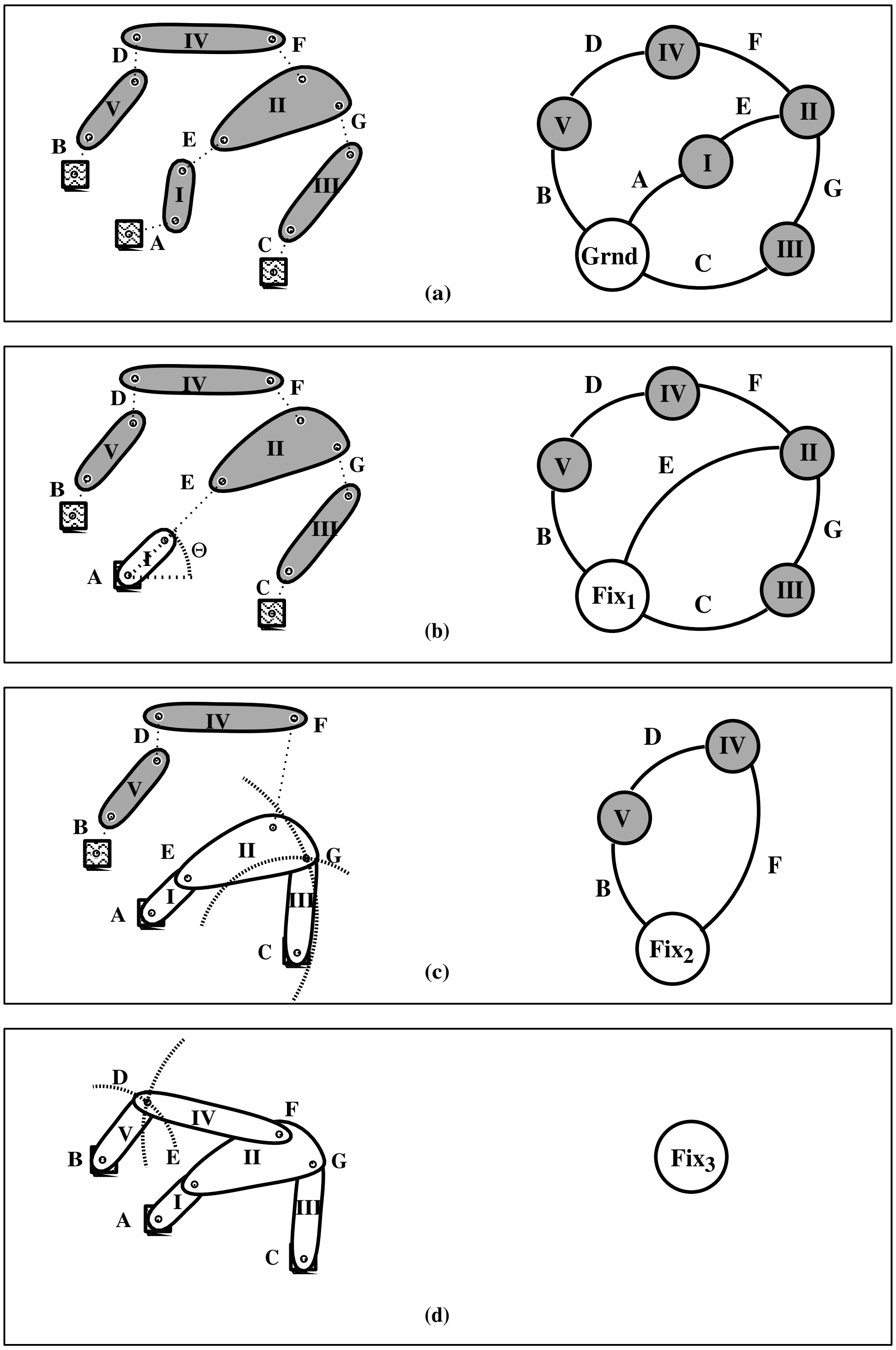


Figure 20: The position analysis of the 6-bar mechanism driven from joint A and the corresponding contraction of the mechanism graph

figure 20(b) and as such represents an example of propagation.

Contraction involves the reduction of three nodes connected by three edges. Algebraically, this corresponds to simultaneously solving two equations in two unknowns as was done geometrically in the computation of the intersection of two circles. Contraction allows the solution of systems of equations that are almost triangular but require a minimal set of manipulations to make them truly triangular.

While in principle, contraction can be extended to reduce equations in three variables for three dimensional applications, the complexity of completely enumerating such reductions is extreme as demonstrated by Kramer ([KRAM92]). Given an extensible set of constraints on n variables, automated methods for enumerating these reductions and identifying where such reductions are useful will be required to make contraction a general purpose method for constraint satisfaction.

3.3.4 Preprocessing for Efficient Animation: Plan and Move Time

The two phases of the solving process, ordering the equations into semi-triangular form and backsubstituting, may be separated to gain substantial improvement in performance. These two phases correspond directly to Enderton's "plan" and "move" times ([ENDE90]). Notice that once a system of equations has been put into triangular form, the input constraint, which corresponds to the first equation, E_1 in the triangular system, may be given a new input value and the backsubstitution repeated using the original ordering of the equations. For a given mechanism and a given input site, the planning phase need only be performed once.

This ability to decompose the solution process is extremely helpful in providing interactive response times. When a user selects a given link as a driving input, the constraint solver immediately calculates and stores the triangular form for the system of constraints. The ordering in this semi-triangular form corresponds to a plan for solving the mechanism in closed form. The user can then move the link and watch the animation of the mechanism. During this animation, the system need only perform the required backsubstitution according to the stored plan. The planning phase need not occur at interactive speeds. For example, when a user clicks on a new link from which to animate a constrained mechanism, a half second delay will go unnoticed; however, once he begins animating, half second delays between frames are unacceptable. Essentially, the ordering of the equations into semi-triangular form is a compilation of the solution. Fast solutions that can be executed quickly for interactive animation are substantially more important than fast compile times.

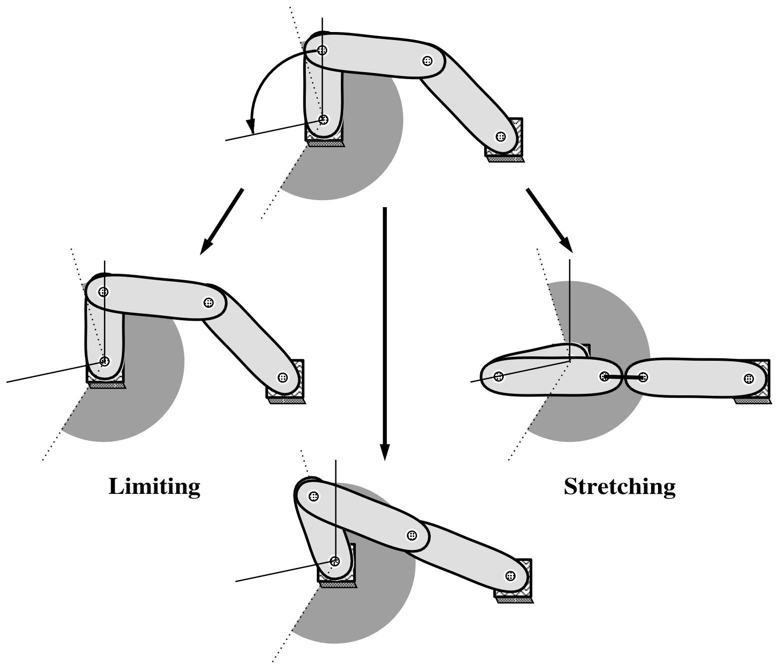


Figure 21: Three policies for handling an input that would drive the mechanism into an unsatisfiable position.

3.3.5 Inconsistent Systems: Breaking

During an animation, the user may drive the mechanism beyond its physical limits. Algebraically, the user is asserting an input that makes the system of equations inconsistent. Enderton prescribes four possible policies for handling this situation at the user interface level of which three are described here: limiting, searching, and stretching (figure 21).

Limiting prevents motion that would move the mechanism into an inconsistent state. If a solution cannot be found for the new input, a new frame is not generated. The difficulty with this approach is that the samples taken from the mouse are not continuous, so the last valid constellation generated may be some distance from the limiting point of the linkage. A more robust policy is *searching* in which the break point of the mechanism is found using a binary search or some other algorithm that permits the linkage to move to the limit of its motion and stop. While Enderton presents a simple explanation of searching, locating break points is a non-trivial task. In fact, recognizing that any particular motion is valid is a research issue in itself. Consider a ground joint that may be driven through all but two degrees of arc (figure 22). In these remaining two degrees, the underlying constraint system is inconsistent. A user who moves the mouse four degrees may completely miss this region if the mouse is not sampled frequently enough. The

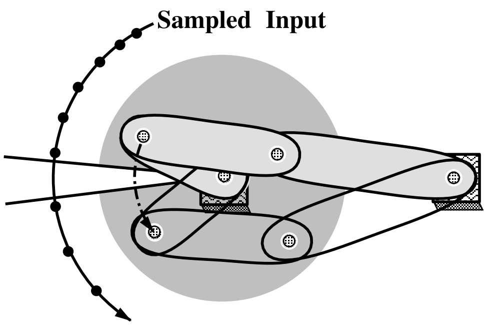


Figure 22: A mechanism may pass through a break region if the mouse is not sampled frequently enough. Determining these break regions for complex mechanisms is non-trivial.

mechanism will appear to have passed through this region creating an animation that is not physically realizable. Searching for break points in all but the most trivial cases is not a simple task.

Stretching allows joints to break, making joints in some sense, rubbery. Stretching is a relaxation of one or more of the constraints that define the linkage. For example, a revolute joint which constrains points on different links to be coincident may be relaxed so as only to constrain these points to be as close to one another as possible. In a closed-form solver, this policy is easily implemented. If a step of the plan cannot be executed, a relaxed version of that step is performed. Notice that the searching policy described above is simply a relaxation of the input constraint. If the user drives the input into an inconsistent state, the input constraint is relaxed, and the input link only approximates the angle requested by the user. Finally, notice that it is unclear which joint constraint should be relaxed. This choice might be made arbitrarily by the solver when it reaches a point in the plan where it can no longer satisfy the next constraint equation. This is the primary difference between searching and stretching. Stretching permits the constraint solver to determine which constraint to relax; searching mandates that the input constraint be relaxed. Searching is a more difficult policy to implement in closed form as the occurrence of a break may be unknown until a substantial amount of the plan has been executed. This break must then be translated into a limitation on the range of the driving input, essentially forcing the constraint solver to establish the exact geometric relationship between the driving input and the constraint that cannot be satisfied.

3.3.6 Genericity

Degenerate mechanisms may still be animated using contraction provided that the number of degrees of freedom of the mechanism is consistently one. Transient degeneracies are ignored. The extra degrees of freedom introduced by transient degeneracies cannot be controlled. Thus, the linkage in figure 14(a) would be animated as though it were generic, the passive degree of freedom being ignored. The linkage in figure 14(b) could be animated correctly.

Degeneracy that results in a single-degree-of-freedom mechanism generally appears as a loop containing two nodes and two edges. If one of these nodes is fixed, the remaining mobile node contributes three degrees of freedom and the two edges remove a total of four degrees of freedom, thus overconstraining the linkage. If the mechanism still exhibits positive mobility, the two edges must thus be non-generic; that is, one of the four constraints is degenerate. In a closed-form solver, checking whether mobility still exists is simple. Each of the constraints is satisfied using contraction; however, after three of the constraints are satisfied using contraction, no degrees of freedom will remain to satisfy the fourth constraint. The remaining constraint is simply checked for consistency. If the fourth constraint has been satisfied in the process of satisfying the first three, then this remaining constraint is viewed as degenerate. If, after satisfying the first three constraints, the fourth constraint is checked for consistency but has not been satisfied, the system of constraints is inconsistent and a breaking policy must be invoked.

3.3.7 Limitations of Contraction

The success of contraction is predicated on the ability to progressively decompose the mechanism graph into loops of three nodes and three edges. These three-loops are also known as dyads or class II groups ([ROMD92]). Driving the 6-bar mechanism from joint B is an example of a situation that does not permit position analysis using contraction. This driving input collapses the ground node and V into a single node (Fix_1) as indicated in figure 23. There are, however, no solvable dyads in the remaining mechanism. The smallest loop in the remaining graph has four nodes and four edges, and thus, contraction cannot continue at this point. Notice that the solving of the remaining system requires us to place the triangle EFG in such a way that each of the vertices of the triangle lie on a particular circle, as shown in figure 24. No simple closed-form solution for performing this task is known, and thus an iterative numerical technique must be used.

Another primary difficulty with contraction methods is that when approached without an underlying symbolic algebra system, the complexity of handling each of the sets of contractions involve different algebraic manipulations and, in turn, separate lines of code that spell out the analytic solution. For two-dimensional linkages, there are only six solving modules corresponding to the six combinations of three lower-pair joints (RRR, RPR, PRR, PPR, PRP, and PPP); however, the number of combinations of con-

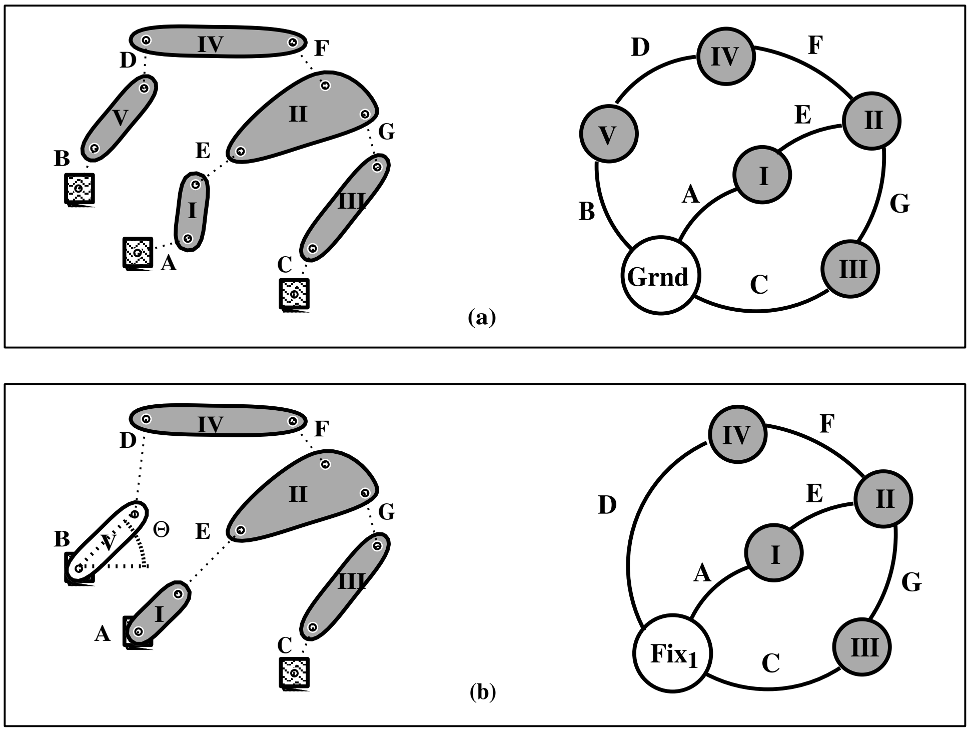


Figure 23: The position analysis of the 6-bar mechanism driven from joint B. In subfigure (b), no further contraction is possible as the smallest loop in the graph contains four nodes and four edges.

straints grows exponentially in the number of dimensions and the number of constraints admitted. Glenn Kramer presents 82 'plan fragments' which do geometric reductions on seven types of basic constraints in three dimensions. Each of these reductions has an algebraic equivalent. Simply examining one of Kramer's plan fragments (figure 25) reveals the programming complexity involved in this task and reveals the need for simpler, more general abstractions if extensible constraints systems are to be constructed.

Finally, contraction methods do not address the problem of under-constrained systems. While it is clear that one could add constraints in some way until the system is fully constrained and then use a contraction method, it is not clear where such constraints should be introduced. Also, motion constrained by introducing extra constraints in this manner is unrealistic in that it displays a determinism inappropriate to multiple-degree-of-freedom mechanisms. For example, straight line motion may be introduced where none is implied by the structure of the mechanism. While contraction offers much in terms of efficiency, it is not a general enough mechanism to handle many of the needs of a complete constraint solver. As will be seen, numerical techniques fulfill many of these needs and, when combined with efficient closed-form techniques, provide a relatively complete system.

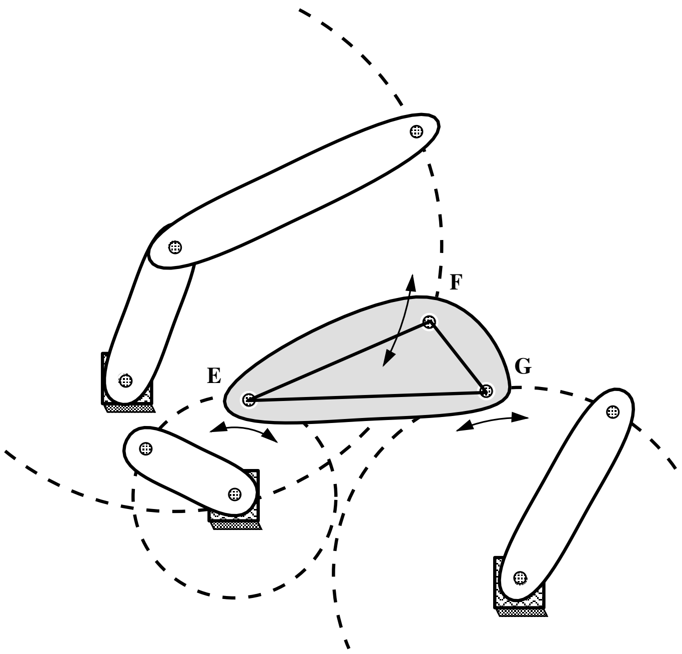


Figure 24: The position analysis requires the placement of the triangular link such that each of its three vertices lie on three triangles and, in turn, an iterative numerical solution.

```
PFT entry: <1,1,in-plane> (?M_1 fixed)
        Initial status:
          1-TDOF(?geom, ?point, ?line, ?lf)
          1-RDOF(?geom, ?axis, ?axis_1, ?axis_2)
        Plan fragment:
          begin
          R[0] = line(?point, ?axis_1);
          R[1] = ⊥-dist (gmp(?M_1), R[0]);
          R[2] = cylinder(?line, ?axis, mag(R[1]));
          R[3] = plane(gmp(?M_2), gmz(?M_2));
          R[4] = intersect(R[2], R[3], 0);
          unless ellipse?(R[4])
            error(⊥-dist(R[2], R[3]), estring[6]);
          R[5] = a-point(R[4]);
          1t-1r/p-p(?geom, ?point, ?line,
            ?axis, ?axis_1, ?axis_2,
            R[5], gmp(?M_1), ?lf, 0);
          R[6] = ⊥-dist(?point, R[5]);
          R[7] = ellipse+r(R[4], gmz(?M_2), R[5]);
          end;
        New status:
          1/2-TDOF(?geom,R[5],R[7], R[7])
          1/2-RDOF(?geom, ?axis, ?axis_1, ?axis_2)
        Explanation:
          Geom ?geom has only one rotational and one
          translational degree of freedom. First, a point on
          the plane defined by ?M_2 and its z axis which
          will coincide with ?M_1 is found; then, the geom is
          translated and rotated to make that point
          coincident with ?M_1. A single degree of freedom,
          combining rotation and translation, remains.
```

Figure 25: One of 82 plan fragments that geometrically compute reductions that can be performed on collections of constrained geometry in Kramer's geometric constraint solver.

3.4 Numerical Techniques

Iterative numerical techniques are frequently discounted as unacceptable for interactive equation solving for a varety of reasons which, while having some foundation, do not generally undermine the value that such techniques offer. The primary benefit of numerical solutions is their generality, and as will be seen, this generality can be extended to handle under- and over-constrained systems, systems that cannot be made triangular, and systems that exhibit transient degeneracies: precisely those systems that contraction does not address.

3.4.1 Description

Solving systems of constraints is essentially the mathematical task of finding the roots of a system of equations. Thus, in one dimension, we seek a value of x such that

$$f(x) = 0$$

for some function (or constraint) f. In multiple dimensions, if there are n free variables and k constraints, we seek values for $x_1 ... x_n$ that satisfy the k equations

$$\begin{aligned} f_1(x_1, ..., x_n) &= 0 \\ \vdots \quad & \quad \vdots \\ f_k(x_1, ..., x_n) &= 0 \end{aligned}$$

or in vector notation

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}.$$

Numerical solutions begin with some starting guess for a solution and then iteratively improve that solution by moving closer to an actual root of the system. There are a number of problems that can hinder iterative methods; however, most of these are avoidable if proper precautions are taken:

- *The solver may fail to converge to a solution.* A solver will fail to converge if no real solution exists or may fail to converge simply because the rate of convergence is exceedingly slow. In either case, after a fixed number of iterations of the solver, no solution to the system of equations has been found. One difficulty with numerical solvers is that there is no obvious way to delineate between these two very different cases. Most applications require a different response when an inconsistent system has been created from that when the solver is converging slowly.

- *The solver may converge to the wrong solution.* As the user varies the input by moving the mouse, it is important that the 'same' solution is tracked and the animation not oscillate between two different branches of a mechanism's motion. If the system converges to the wrong solution, physically unrealizable motion will be displayed.

- *The solver may have difficulty with over- and under-constrained systems caused by redundant constraints or extra degrees of freedom.* Most numerical solvers, as will be seen, have as part of the inner loop, a linear equation solver. Systems of constraints that are singular or nearly singular may cause the linear equation solver to fail or to provide inaccurate results.

- *Numerical solvers are computationally complex.* Numerical solvers have complexity no less than quadratic in the number of unknowns, limiting the size of the problems that may be solved and animated interactively. This limitation, while serious, is frequently overstated. Substantial systems can be animated interactively on relatively slow workstations. While it is simple to create constraint systems that have extremely complex interactions, we doubt whether such systems are generally useful. Intricate constraints systems that do not have a straightforward solution path tend to be extremely difficult for designers to understand and, in turn, are less likely to be considered during the design process. The order of growth of the numerical solver's running time with respect to the number of constraints, while important, may not be as important as its ability to handle most smaller systems of constraints robustly.

These difficulties will be discussed in the context of the numerical methods implemented in MechEdit. Some of these difficulties do not arise because of the nature of the application, and others are easily overcome if proper numerical techniques are used.

3.4.2 Newton's Method

Newton's method, also known as the Newton-Raphson method, is a well-known technique for finding the roots of a function in one variable and can be generalized easily to root-finding in multiple dimensions. While we assume that the reader is familiar with Newton's method in one dimension, we present a brief review to assist the reader with an understanding of the extension to multiple dimensions.

A continuous function in the neighborhood of some point x may, by Taylor's theorem, be approximated by

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{1}{2}f''(x)\delta^2 + \cdots$$

If δ is small enough, the non-linear terms are negligible and the function may be approximated by

$$f(x+\delta) \approx f(x) + f'(x)\delta$$

To locate roots, we set $f(x+\delta) = 0$ which yields, in terms of δ

$$\delta = -\frac{f(x)}{f'(x)}$$

The algorithm for root-finding is thus to begin with some initial guess for the value of x, x_0, and iteratively calculate

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

until $|f(x_n)| < \epsilon$. Graphically, this equates to approximating the function $f(x)$ at some point x by a straight line with slope $f'(x)$ passing through the point $(x, f(x))$. The zero crossing of this line is then used as the next guess for x. Figure 26 demonstrates Newton's method graphically. Note that the method also requires that $f'(x) \neq 0$ for the initial guess and every following guess including the solution.

The key assertion which is made in the above description is that for the method to work correctly δ must be 'small enough'. This is to say that for any continuous function with continuous first derivative, there exists an initial guess for x, x_0, such that $|f(x_0)| \geq \epsilon$ but, using x_0 as an initial guess, Newton's method will converge to some x_n such that $|f(x_n)| \leq \epsilon$ for arbitrary $\epsilon > 0$. Insuring a good initial guess, and in turn, a sufficiently small δ is not as difficult as it may seem for many applications. By making the user specify constraint systems using a physical paradigm of dragging parts and connecting them, systems of constraints are constructed in a solved state in which values for all unknowns are supplied. Once created, constraints are altered only by moving the mouse along a continuous path giving, in theory, a smooth path through which the various unknown values travel continuously. If the mouse is sampled frequently enough, the corresponding δ values will be sufficiently small to permit Newton's method to converge.

The power of Newton's method lies in its very rapid local convergence toward a solution. Its primary detractor is its poor global convergence. Numerous techniques exist for improving upon the basic algorithm including, among others, the use of a hybrid of Newton-Raphson and standard bisection techniques for bracketing a root; however, the basic concept behind the algorithm is the same in every implementation including its expansion to multiple dimensions ([KAHA89]).

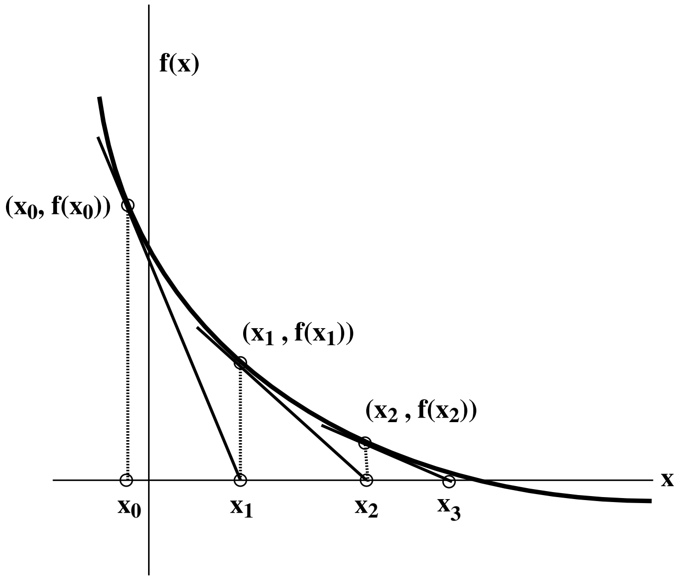


Figure 26: Simple Newton-Raphson iteration for one-dimensional root-finding

3.4.3 Newton's Method in Multidimensional Space

For constraint problems involving systems of equations in multiple variables, Newton's method is modified in a straightforward way. A root is now sought for a set of n functions $f_i, i = 1, ..., n$ in n variables, $x_1, ..., x_n$. The vector $x_1, ..., x_n$ is represented by $\mathbf{x}$, and the vector $f_1(\mathbf{x}), ...f_n(\mathbf{x})$ is represented $\mathbf{f}(\mathbf{x})$, thus a value for $\mathbf{x}$ is sought that satisfies

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

The Taylor series for each $f_i(\mathbf{x})$ is

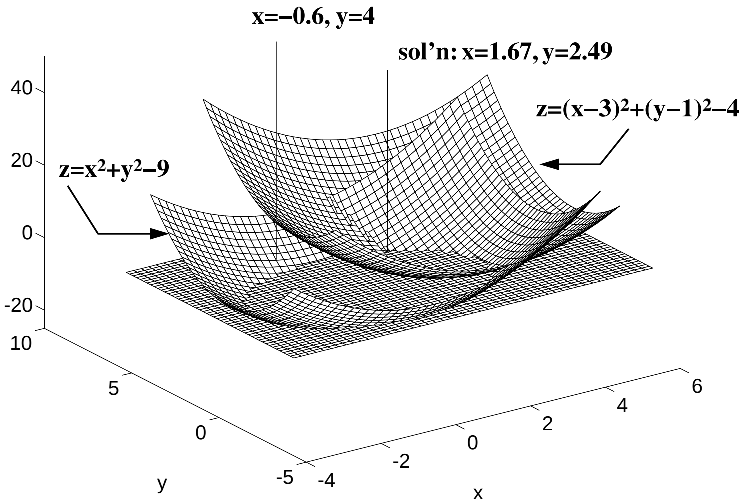
$$f_i(\mathbf{x} + \delta\mathbf{x}) \approx f_i(\mathbf{x}) + \sum_{j=1}^{n} \frac{\partial f_i}{\partial x_j} \delta x_j + O((\partial\mathbf{x})^2)$$

where $O((\partial\mathbf{x})^2)$ represents the higher order terms of the expansion. The terms $J_{ij} = \frac{\partial f_1}{\partial x_j}$ can be understood as the elements of the Jacobian matrix, J, in which each row represents a particular constraint equation, f_i, and each column represents a particular variable x_j.

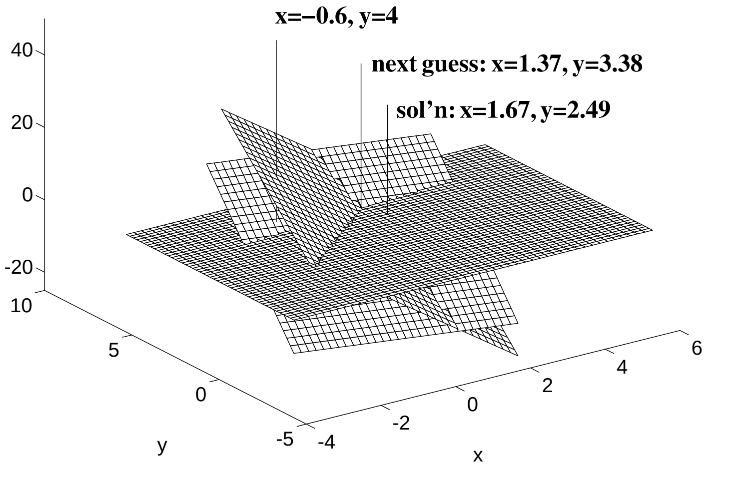
Neglecting higher order terms and taking advantage of matrix notation gives the equation

$$\mathbf{f}(\mathbf{x} + \delta\mathbf{x}) \approx \mathbf{f}(\mathbf{x}) + \mathbf{J} \cdot \delta\mathbf{x}$$

Graphically, this equation represents n hyperplanes tangent to the n continuous hypersurfaces $f_i(\mathbf{x})$ for some point $\mathbf{x}$ (figure 27). As in the one-



(a)



(b)

Figure 27: Newton-Raphson iteration in two dimensions solving the intersection of two circles. Subfigure (a) shows the two continuous surfaces whose intersection with the zero plane are the circles of interest and a starting guess, $(-0.6, 4)$. *Subfigure (b) shows the two planes tangent to the surfaces in (a) at point* $(-0.6, 4)$. *The intersection of these planes with the zero plane represents the next guess in the iteration. Notice the rapid convergence toward a solution.*

dimensional case, roots of these approximating hyperplanes are located by setting $\mathbf{f}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{0}$. Unlike the one-dimensional case, extra complexity is added because a vector must be located which simultaneously zeroes all of these hyperplanes. This yields a set of linear equations that must be solved for the corrections, $\delta\mathbf{x}$, to the original guess $\mathbf{x}$:

$$\mathbf{J} \cdot \delta\mathbf{x} = -\mathbf{f}$$

Once this system of linear equations has been solved, the corrections, as in the one-dimensional case, are added to the original guesses:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \delta\mathbf{x}$$

This process is repeated until the variables, $\mathbf{x}$, or the functions, $\mathbf{f}(\mathbf{x})$, have converged.

Notice that throughout this explanation, it was assumed that the number of variables and the number of equations are the same. This is necessary to make the solving of the linear equations simple. Notice that if any set of constraint equations are not independent, the resulting Jacobian matrix will be singular, and the linear equation solving step will be complicated. This leads to the claim made previously that numerical techniques are ill-suited to systems with redundant constraints or under-constrained systems. It will be shown later that the requirement that the number of independent equations and number of variables be equal may be relaxed if a more robust linear equation solver is used.

3.4.4 6-bar Example

If the 6-bar mechanism from figure 17 is driven with link V, the mechanism cannot be solved using contraction. Using Newton's method, each of the five mobile links can be assigned a set of variables describing their positions, and each of the joints define equations that constrain these variables. Each of the mobile links has three variables, $\{\Theta, \Delta_x, \Delta_y\}$, describing its three degrees of freedom. Since all of the joints are revolute, all of the constraints (with the exception of the driving input constraint) will be of the form $P_x - Q_x = 0$ or $P_y - Q_y = 0$ where (P_x, P_y) and (Q_x, Q_y) are the global coordinates of points P and Q which lie on different links and are constrained to be coincident. As described in section 2.1.2, if P and Q are described in their local coordinate frames by the constant coordinates (a, b) and (c, d) respectively, the functions for which a zero must be found are of the form:

$$f(\Delta_x^I, \Theta^I, \Delta_x^{II}, \Theta^{II}) = P_x - Q_x = (\Delta_x^I + a\cos\Theta^I) - (\Delta_x^{II} + c\cos\Theta^{II})$$

$$g(\Delta_y^I, \Theta^I, \Delta_y^{II}, \Theta^{II}) = P_y - Q_y = (\Delta_y^I + b\sin\Theta^I) - (\Delta_y^{II} + d\sin\Theta^{II})$$

The partial derivatives of these functions can be computed easily.

$$\frac{\partial f}{\partial \Delta_x^I} = \frac{\partial g}{\partial \Delta_y^I} = 1$$

$$\frac{\partial f}{\partial \Delta_x^{II}} = \frac{\partial g}{\partial \Delta_y^{II}} = -1$$

$$\frac{\partial f}{\partial \Theta^I} = -a \sin \Theta^I, \frac{\partial f}{\partial \Theta^{II}} = c \sin \Theta^{II}$$

$$\frac{\partial g}{\partial \Theta^I} = b \cos \Theta^I, \frac{\partial g}{\partial \Theta^{II}} = -d \cos \Theta^{II}$$

Clearly, the partial derivatives of f and g with respect to any other variable other than those above is 0. Thus, we have a vector of functions. In the case of the 6-bar, there are two functions for each of the 7 joints, $\{A, B, C, D, E, F, G\}$, giving 14 functions. There is one additional driving input function of the form $h(\Theta^V) = \Theta^V - \alpha$, where α is a constant indicating the angle at which link V is placed by the mouse. The partial derivatives of this function are also easily computed. Each mobile link, I, II, III, IV, V, has three variables associated with that link, giving 15 variables. Thus, we have 15 functions which are associated with the 15 rows of the Jacobian and 15 variables which are associated with the columns.

Given some starting guess, the values of the 15 functions can be computed to give the vector $-\mathbf{f}$. The 15 by 15 Jacobian matrix, $\mathbf{J}$, can be computed using the equations for the partial derivatives above. The equation $\mathbf{J} \cdot \delta\mathbf{x} = -\mathbf{f}$ can then be solved for the correction vector to the original guess, $\delta\mathbf{x}$. This is added to the initial guess vector and the process repeated until the guess converges to a solution of the system of equations.

3.4.5 Linear Equation Solving within Newton's Inner Loop

At the heart of the Newton-Raphson method in multiple dimensions is a linear equation solver. The process of simultaneously solving a set of linear equations is performed to determine in what direction and how far the current guess should be moved to arrive at a root of the original system of nonlinear equations. Of course, because the linear equations are only a local approximation of the original system, their solution can, at best, lead only to an improved guess with the hope of ultimately converging to a solution.

Because every iteration of Newton's method requires that a new system of linear equations be solved, the performance of the linear equation solver, together with the rate at which the method converges, determines the performance of the algorithm. Because linear equation solving is an $\Omega(n^2)$ process, it is clear that the order of growth of numerical methods is limiting; however, modern workstations provide enough compute power to solve significant systems of constraints interactively. The computational complexity of numerical methods does suggest the need to use closed-form solutions whenever possible.

Beyond this computational complexity lies another problem: that of robustly handling over- and under-constrained systems as well as degenerate systems. Algebraically, over-constrained systems correspond to systems that have more equations than unknowns, and under-constrained systems correspond to systems that have fewer equations than unknowns. Degenerate systems correspond to singular systems. Non-linear systems of constraints that are over- or under-constrained or have inherent degeneracies clearly will lead to equivalent degeneracies in the linear system derived from the Jacobian. The ability to solve such linear systems in a reasonable way is necessary to facilitate a robust solver for non-linear systems.

3.4.6 Singular Value Decomposition

Singular value decomposition (SVD) provides a powerful method for solving linear systems that are singular or nearly singular. By recognizing such a singularity and selecting the shortest vector (in a least squares sense) from the infinite number of vectors that satisfy a singular linear system, SVD provides the functionality to overcome under-constraint and degeneracy in constraint systems.

SVD is based on the ability to decompose any m x n matrix, $\mathbf{A}$, where $m \geq n$ into the product of three matrices, $\mathbf{U\Sigma V^T}$. $\mathbf{U}$ is an m x m orthogonal matrix, $\mathbf{V}$ is an n x n orthogonal matrix, and $\mathbf{\Sigma}$ is an m x n matrix whose off diagonal entries are zero and whose diagonal elements $\sigma_1 ... \sigma_n$ are positive and in decending order. Thus, any such matrix $\mathbf{A}$ can be written:

$$\begin{pmatrix} \\ \\ & \mathbf{A} & \\ \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \\ & \mathbf{U} & \\ \\ \\ \end{pmatrix} \begin{pmatrix} \sigma_1 & & & & \\ & \sigma_2 & & & \\ & & \sigma_3 & & \\ & & & \ddots & \\ & & & & \sigma_n \\ \\ \end{pmatrix} \begin{pmatrix} \\ & \mathbf{V^T} & \\ \\ \end{pmatrix}$$

$$\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \cdots \geq \sigma_n \geq 0$$

The assumption that the number of rows of $\mathbf{A}$ is greater than the number of columns, $m \geq n$, implies that there are at least as many constraint equations as unknowns; however, this assumption is merely a convenience. Any system of linear equations may be expanded without altering the set of solutions simply by adding rows of zeros until $m = n$.

There are numerous interesting properties about this factorization that make it useful within the inner-loop of a multidimensional Newton-Raphson solver. First, the σ_i's are unique and are called the *singular values* of $\mathbf{A}$. The number of nonzero σ_i's gives the rank of $\mathbf{A}$, and the ratio, c, of σ_1 to the smallest nonzero σ_i measures how close the matrix $\mathbf{A}$ is to a matrix of lower rank. c is called the *condition number* of $\mathbf{A}$. Computationally, the condition number identifies if a system is ill-conditioned and should be treated as numerically singular.

Computation of $\mathbf{A^{-1}}$, and, in turn, the ability to generate a solution to $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$, is straightforward in the simple case where $\mathbf{A}$ is an n x n matrix of full rank:

$$\mathbf{A} = \mathbf{U\Sigma V^T}$$

$$\mathbf{A^{-1}} = \mathbf{V^{T^{-1}}\Sigma^{-1}U^{-1}}$$

Since $\mathbf{V}$ and $\mathbf{U}$ are orthogonal, they may be inverted by taking the transpose. Since $\mathbf{\Sigma}$ is diagonal, its inverse is the matrix whose diagonal elements $\sigma_i^{-1} = \frac{1}{\sigma_i}$. Thus, $\mathbf{A^{-1}}$ may be computed from the singular value decomposition as:

$$\mathbf{A^{-1}} = \mathbf{V\Sigma^{-1}U^T}$$

$$\mathbf{\Sigma^{-1}} = \begin{pmatrix} \frac{1}{\sigma_1} & & & \\ & \frac{1}{\sigma_2} & & \\ & & \ddots & \\ & & & \frac{1}{\sigma_n} \end{pmatrix}$$

In the case where $\mathbf{A}$ is of degenerate rank, k, that is $k < n$, $\sigma_i = 0$ for all $k < i \leq n$. Thus, computation of $\mathbf{\Sigma^{-1}}$ cannot be performed simply by taking the reciprocal of the σ_i's. For a given set of simultaneous equations where $\mathbf{A}$ is not of full rank,

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$$

there is some subspace of $\mathbf{x}$, called the *nullspace*, that maps to zero, $\mathbf{A} \cdot \mathbf{x} = \mathbf{0}$. The dimension of the nullspace is called the *nullity* of $\mathbf{A}$ and is equal to $n - k$ where k is the rank of $\mathbf{A}$.

A particular $\mathbf{b}$ may or may not be in the range of $\mathbf{A}$, that is, there may or may not be a solution to the equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. If, however, $\mathbf{b}$ does lie in the range of $\mathbf{A}$ but $\mathbf{A}$ is not of full rank, then there will be more than one solution to the equation. In fact, there will be a solution space for $\mathbf{x}$ of dimension equal to the nullity of $\mathbf{A}$. This is clear since any vector in the null space may be added to the solution vector since vectors in the null space give $\mathbf{A} \cdot \mathbf{x} = \mathbf{0}$. In the context of constraint systems, the linear systems to be solved will be derived from the Jacobian of the system of non-linear constraint equations. In this context, k represents the number of independent constraint equations, and n represents the number of independent variables. The nullity of the Jacobian, $n - k$, represents the degrees of freedom of the system of constraints.

Since there are an infinite number of solutions, one must be selected if we are to proceed toward a solution in the next Newton-Raphson iteration. Singular value decomposition is extremely valuable as it gives the shortest vector in the solution space which, as will be shown, leads to an approximation of the minimum change in position required to satisfy the system of constraints. Recall that the system of equations that are being solved within the inner-loop of the Newton-Raphson solver are tangent hyperplanes that approximate the hypersurfaces of the constraint equations. The solution of these equations is the intersection of all of these hyperplanes with an additional zero hyperplane (figure 27) to determine a δ offset which, hopefully, is closer to a solution of the original system. Because the solution of the system of linear equations represents a δ offset for the next guess, a short solution vector represents a small change to the guess, and a long solution vector represents a large change to the guess. Thus, choosing the shortest solution vector from an infinite number of solution vectors gives the smallest possible change, in a least squares sense, to the current guess which simultaneously zeros all of the approximating hyperplanes. If the equations represent geometric transformations, this property is extremely valuable as it selects a solution which, in some sense, minimizes the physical motion of the system. The graphical effect of this property is valuable as it produces motion very similar to that of physical bodies in the presence of friction. The importance of such predictable motion is discussed further below.

When $\mathbf{\Sigma^{-1}}$ cannot be calculated using reciprocals because the linear system is not of full rank, a modified inverse of $\mathbf{A}$ is taken which allows the computation of a solution vector, if one exists, of minimum length. The *generalized inverse*, also known as the Moore-Penrose pseudoinverse ([NOBL77]), of $\mathbf{A}$, $\mathbf{A}^+$ is defined to be:

$$\mathbf{A}^+ = \mathbf{V}\mathbf{\Sigma}^+\mathbf{U^T}$$

where $\mathbf{V}$ and $\mathbf{U}$ are defined as before, and $\mathbf{\Sigma}^+$ is the diagonal matrix derived from $\mathbf{\Sigma}$ as follows:

$$\sigma_i^+ = \begin{cases} \frac{1}{\sigma_i} & \text{if } \sigma_i \neq 0 \\ 0 & \text{if } \sigma_i = 0 \end{cases}$$

From this $\mathbf{\Sigma}^+$, the solution to $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ may be computed as

$$\mathbf{x} = \mathbf{V}\mathbf{\Sigma}^+\mathbf{U^T}\mathbf{b}$$

In simple terms, a system of possibly singular linear equations may be solved for the shortest vector in the solution space by computing the singular value decomposition of $\mathbf{A}$, taking the reciprocal of all of the nonzero singular values of $\mathbf{\Sigma}$ to produce $\mathbf{\Sigma}^+$ while ignoring zero values on the diagonal, and then performing the matrix multiplications shown in the equation above. It is a simple proof ([PRES92]) to show that this vector will be the solution vector of minimal length.

This quality that the linear equation solver within the Newton-Raphson inner-loop provides shortest vector solutions regardless of the condition number of the system of linear equations offers massive leverage toward providing robust, predictable interactions for under-constrained and degenerate mechanisms. Because these shortest vectors are part of an iterative loop, the resulting motions cannot be characterized easily and should not be mistaken for the globally minimal motion that satisfies the input constraint; however, the resulting motion in MechEdit is strikingly predictable and has a very realistic, physical nature. Users appear to be able to control extremely under-constrained motion relatively easily as the motion is 'close enough' to the real world to match their physical intuitions. An example animation sequence of an extremely under-constrained mechanism is shown in figure 28. This is a simple chain of links which are grounded by a revolute and pulled from the last link in the chain. The use of singular value decomposition in the inner-loop of the Newton-Raphson iteration provides a smooth, predictable extension of the chain of links.

4 Mixed Analytic/Numerical Solutions

Both analytic and iterative numerical methods by themselves fall short of providing an efficient and robust constraint solver. To achieve reasonable performance, analytic techniques must be used; however, analytic techniques are not sufficiently general to solve many systems of constraints. MechEdit uses a mixed solver in which contraction is used to reduce the complexity of the system, solving it if possible, and multidimensional Newton-Raphson iteration is used to solve for any remaining undetermined variables. Such a mixed solution provides extremely efficient solving where possible, yet provides the ability to animate any linkage. Every reduction made by the closed-form solver amounts to substantial savings in the numerical solver as it reduces the number of variables, n, and, in turn, the size of the Jacobian matrix that must be decomposed in $\Omega(n^2)$ time during every iteration of Newton-Raphson.

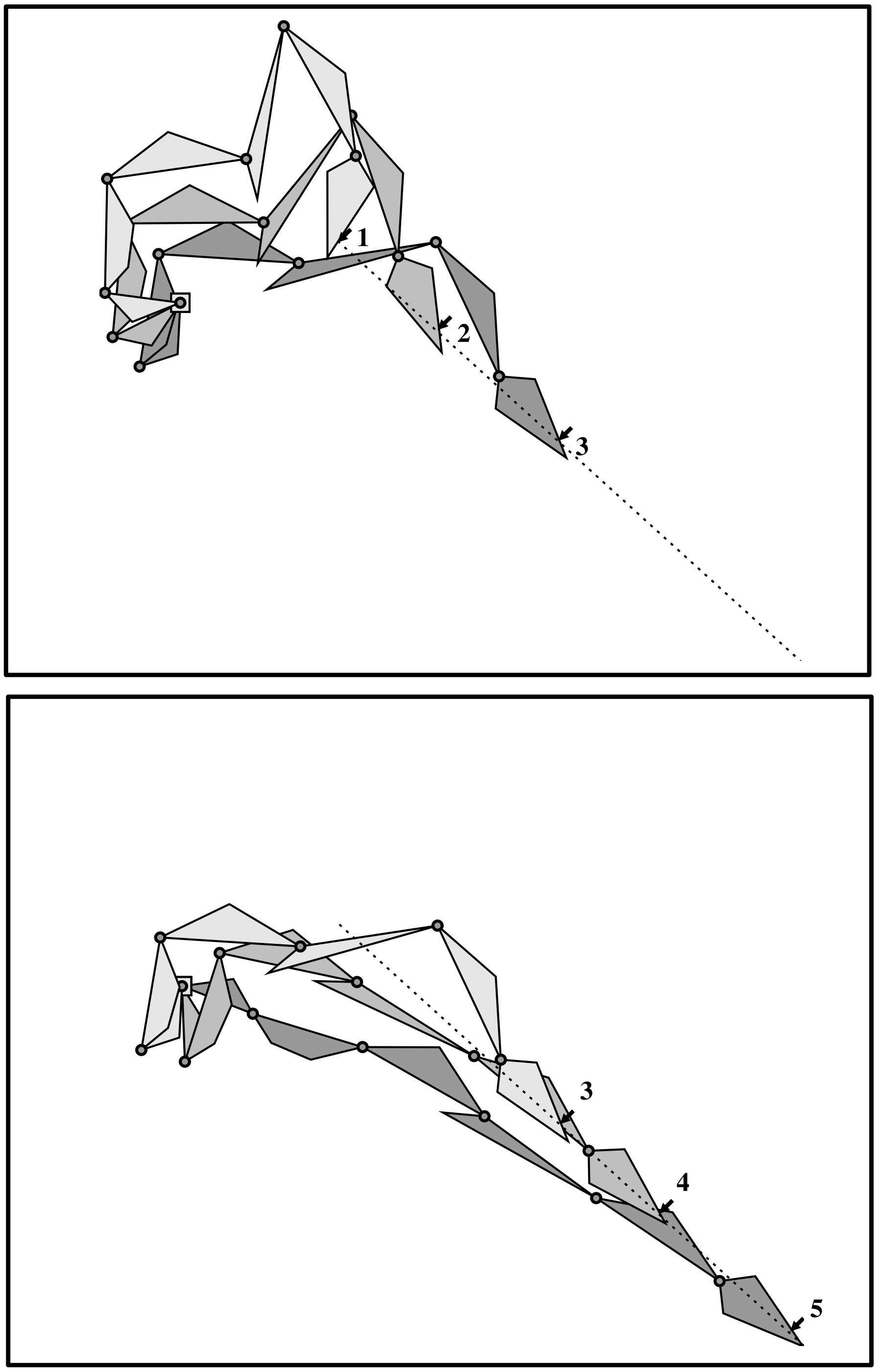


Figure 28: An extremely under-constrained system, this chain of links is shown in two sets of frames from a single animation. The last link in the chain is pulled along the dotted line by the cursor. The mechanism extends smoothly and predictably despite its many degrees of freedom.

4.1 Method

The addition of a numerical solver on top of an existing closed-form solver is straightforward. Assuming a mechanism graph has been constructed and reduced using contraction. Each of the remaining nodes (not including the ground node) in the graph corresponds to a rigid body with three degrees of freedom, thus three free variables. Each remaining edge corresponds to a joint and, in turn, two constraints. The collection of nodes and their associated free variables gives the vector $\mathbf{x}$. The collection of edges and their associated constraints gives $\mathbf{f}(\mathbf{x})$. In order to invoke Newton-Raphson, there must also exist a means of constructing the Jacobian of partial derivatives of $\mathbf{f}(\mathbf{x})$. Each constraint must be capable of providing its vector of partial derivatives. The initial guess for Newton-Raphson is taken from the current position of the links as it is expected that this will be close to the new solution, and Newton-Raphson is invoked to solve the remaining variables.

Considering this task algebraically, it can be seen that the system of equations is placed in triangular form during the planning phase. If this is not possible, a numerical solution is required. In this case, as many of the equations as can be placed in triangular form are ordered accordingly (figure 29) using contraction. The lower right hand portion of the system corresponds to the portion of the mechanism graph that can be contracted and, in turn, is the portion of the system that can be solved in closed form by backsubstitution. The upper left hand corner of the system corresponds to the uncontractable mechanism graph which must be solved numerically. At animation time, backsubstitution is performed for the lower right hand portion of the system and the remaining upper left hand portion is solved iteratively.

The mechanism graph represents rigid bodies in two dimensions as nodes with three degrees of freedoms. By decomposing these nodes into the individual freedoms of the body, as is done implicitly by Kramer, the complexity of the numerical step can be reduced. Kramer divides the solving process into two parts, position analysis and locus analysis. Position analysis is the algebraic equivalent of backsubstitution in which variables are assigned values, or geometrically, the degrees of freedom of links are fixed. Locus analysis uses the fact that while a given link may not be fixed in space by a constraint, its degrees of freedom are reduced by that constraint. For example, if a link is fixed to ground by a revolute joint, the three freedoms of that link may be reduced to a single freedom thus simplifying the underlying constraint system. If the mobility of a link is partially constrained, points on that link are confined to a locus of lower degree, in this case a circle, than the space in which the link lives.

Locus analysis is the process by which loci are intersected to con-

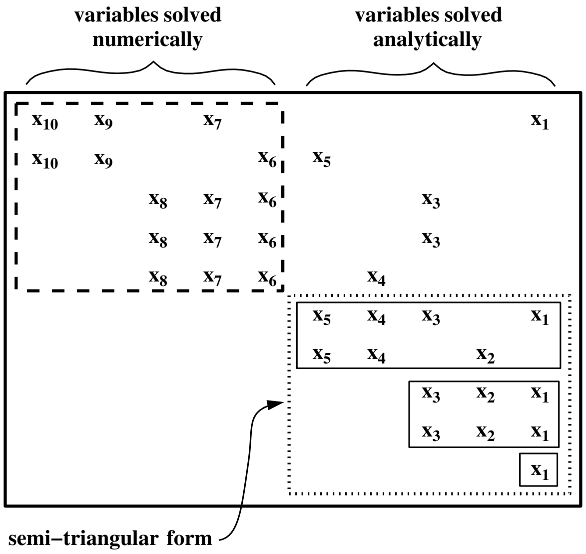


Figure 29: Partial Semi-Triangular Form. Each of the rows represents an equation dependent upon the variables listed in that row. The equations are not necessarily linear. As much of the matrix as possible is placed in semi-triangular form (lower-right box) and can be solved analytically. The remainder of the matrix (upper-left box) cannot be placed in semi-triangular form and is thus solved numerically.

fine points on links to lie in spaces of lower dimension. For example, when two circles are intersected to solve an RRR loop, two one-dimensional loci are intersected to define a zero-dimensional locus representing the collection of points that solve the system. In two dimensions, loci are either one-dimensional (circles and lines) or zero-dimensional (solutions of the system). Algebraically, locus analysis is the combination of pairs of equations into equations of equal or lower polynomial degree. In two dimensions, the reductions always reduce polynomials to zero degree spaces, that is, solutions. Thus, in two dimensions, Kramer's method is computationally equivalent to the method of dyads since every reduction by locus analysis corresponds to one of the dyad reductions. In three dimensions, reductions are more complex, however, each reduction corresponds to combining pairs of equations algebraically to yield equations of equal or lower degree which ultimately may be used to make the system of equations triangular.

Incorporating locus analysis into the numerical solver within MechEdit requires that the system remove as many degrees of freedom as possible before solving the system numerically. If, for example, a link is fixed to ground by a revolute joint, the link's position should be constrained to revolve about this joint and may be represented by one variable. The two translational degrees of freedom and the two constraints representing the revolute joint need not add to the complexity of the remaining system.

This requires that the algorithm for mixing analytic and numerical solutions be modified slightly. As before, the mechanism graph is contracted as much as possible. The remaining nodes of the graph each represent links whose exact position is not known; however, some of these nodes may be partially constrained. The degrees of freedom of each of the remaining nodes is used to determine how many variables will represent the corresponding link during Newton-Raphson iteration. Any constraint that connects a grounded or fully positioned link to an ungrounded link may be removed as it determines the locus of points on that link to be either a circle or a line. After such simplification, the remaining equations and variables are input to the Newton-Raphson algorithm to find a solution.

4.2 6-bar Example

The complexity of animating the 6-bar of figure 17 driven from link V can be substantially reduced from the strictly numerical solution presented in section 3.4.4. The strictly numerical solution examined 15 functions in 15 unknowns, solving a 15 by 15 linear system at every iteration. By recognizing propagations that can be performed and limitations that particular constraints impose on the degrees of freedom of various links, the problem can be reduced to solving a system of 6 equations in 6 unknowns.

First, propagation is used to exactly position link V based upon the input constraint and joint B which together fully constrain its three degrees of freedom. Joint D may thus be considered fixed as can joints A and C which are connected to ground. The corresponding links, IV, I, and III, thus each have only a single rotational degree of freedom, Θ. This gives three variables which combined with link II's three degrees of freedom gives a total of six variables which are solved by Newton's method. It is a simple matter to rephrase the constraint functions and their corresponding partial derivatives in terms of these reduced degrees of freedoms. The process of solving for these remaining six variables using Newton's method continues as before, however, with the complexity in the inner loop of the iteration reduced from solving linear equations in 15 variables to solving linear equations in 6 unknowns. Finally, notice that if a mixed solver were used to animate the same mechanism from ground joint A, the entire linkage would be solved analytically as described in section 3.3.2, and the numerical solver would not be used at all.

4.3 Advantages

Mixing analytic and numerical solvers provides clear advantages over purely analytic and purely numerical solvers. Purely analytic solvers are not suitably general. Under-constrained motion is generally not handled in a reasonable way, and degeneracies can cause substantial difficulty. The primary advantage of closed-form solvers is that for those problems that they can solve, they are extremely fast. Straightforward solution paths can be created which solve mechanisms in linear time in the number of links allowing efficient animation of extremely complex mechanisms.

Iterative numerical solutions are, in many ways, a perfect complement to analytic solutions. When combined with an appropriate underlying linear equation solver, they offer the ability to solve virtually any mechanism. In particular, under-constrained mechanisms that cannot be contracted or that exhibit transient degeneracies can be animated in a predictable, quasi-physical manner. Because numerical solutions are substantially slower than analytic solutions, and because their running time is at least $O(n^2)$ in the number of links, analytic methods should be used extensively to minimize the size of the input to a numerical solver.

The coding of a numerical solver is also substantially simpler than that of an analytic solver. For this project, the addition of the numerical solver on top of the existing analytic solver took one week. Because the programming complexity of analytic solvers is extremely high, programmers might choose to implement a numerical solver and then implement various closed-form reductions to improve its performance. A working system could be implemented very quickly, and pieces of the closed-form solver could be inserted incrementally beneath it for improved performance.

5 MechEdit

5.1 Implementation

MechEdit is an interactive mechanism editor which embodies many of the ideas presented in this report. The implementation is divided into two modules, the user interface and the underlying geometric-constraint system. The code for these modules was written almost completely independently by two different people†, and the interface between the two modules is relatively clean.

Both modules maintain an abstraction of *markers* and *links* where links are rigid bodies and markers are points fixed in the local coordinate

†Roger Bush and the author

system of a particular link. Markers also have the added notion of a direction vector that allows one to constrain one marker to lie along a line defined by another marker. Beyond the basic requirements of creating and destroying markers and links, communication between the two modules centers around the creation, deletion, and fulfillment of constraints.

Constraints can be specified between markers. The underlying constraint solver uses three constraints based on those described by Kramer ([KRAM92]): **coincident**, **in-line**, and **offset**. A **coincident** constraint between two markers constrains those markers to lie at the same location in space and, in turn, creates a revolute joint. An **in-line** constraint forces one marker to lie along the line determined by the direction vector of the other. Two reciprocal **in-line** constraints, for example **in-line(A,B)** and **in-line(B,A)**, can be used to specify a prismatic joint. The **offset** constraint is merely a convenience and could be constructed using the **in-line** constraint and a marker on the ground link. It provides a simple way to specify an offset between the direction vectors of two markers and, as such, gives an easy way to specify the angle of a crank or other driving input.

MechEdit's user interface hides all notions of constraints from the user and operates in a framework of links on a planar surface. Links can be created, moved, and edited with the mouse. Snap-dragging ([BIER86]) is used extensively to allow the user to move links and connect points on different links. When a user moves a point on one link into contact with a point on another link, a revolute joint is formed between those two points. The underlying constraint system is notified of the creation of a **coincident** constraint. The user may toggle the type of joint between revolute and prismatic using the keyboard and the appropriate changes are communicated to the constraint system. Because of the manner in which constraints are inferred from the user's actions, constraints are always added in a solved state, for example, a **coincident** constraint is added by making two markers coincident. Thus, the solver's task becomes one of constraint maintenance rather than constraint satisfaction. This is a powerful simplification since it provides a good initial guess when numerical techniques are required.

The solver is invoked by altering the position or orientation of a grounded marker or the angle provided to an **offset** constraint and then requesting that constraints be resatisfied. The solver constructs a plan using contraction as described in section 3.3, taking the altered marker as the input constraint and contracting the mechanism graph as much as possible given the currently imposed constraints. Contraction proceeds from a permanent ground link and is performed whenever a loop consisting of three nodes and three edges is found. Unlike Romdhane ([ROMD92]) who uses Dijkstra's algorithm for finding the shortest path in a graph, we use a naïve depth first search in which we search from every node in the graph to a depth of three to locate solvable loops. While clearly inefficient, this forms part of the planning phase and need not perform exceptionally well. Long planning times have not yet become a problem for achieving interactivity.

Each node that cannot be contracted is then represented as some set of variables that must be solved numerically. The size of this set corresponds to the number of degrees of freedom that the link maintains after contraction is complete. This is described more thoroughly in section 4.1. The set of unsolved variables, the links they correspond to, and the constraints between these variables are added to the plan as parameters for the numerical phase of the solution. The complete plan thus has two parts: a list of calculations to be performed to place some set of links analytically, and a list of variables and equations to be used to place the remaining links numerically.

The creation of a plan allows rapid calculation of link positions while a set of input parameters are varied continuously. This is important for interactive applications since click-and-drag operations with the mouse generally correspond to selecting and varying an input parameter. The plan is executed by performing hard-coded calculations corresponding to geometric reductions such as the intersection of two circles for those variables that can be solved analytically and then passing the remaining variables to a Newton-Raphson solver to perform the iterative, numerical portion of the plan.

The numerical solver uses the currently displayed positions of the links and markers as an initial guess. The partial derivatives of the constraint equations are provided by hard-coded routines. The Newton-Raphson solver uses singular value decompostion to solve the system of linear equations imposed by the Jacobian of the system of constraint equations. The singular value decomposition is computed using the algorithm supplied by Press, *et al.*, ([PRES92]), which is based upon a routine by Forsythe, *et al.*, ([FORS77]), which, in turn, is based on the original routine of Golub and Reinsch ([WILK71]). The necessary motion of each link is reported to the user interface which then updates its data structures and redisplays the model accordingly.

The user interface relies heavily on the underlying constraint solver for basic operations. To move a link, the user clicks on that link and drags it with the mouse. To perform this task, the user interface module creates two temporary markers, one on the permanent ground link and one on the selected link. Both markers are placed so as to be coincident with the position of the mouse. These two markers are then constrained to be **coincident**. As the mouse is dragged, the position of the temporary ground marker is altered to maintain coincidence with the mouse. The constraint system is used to position the link in space so as to satisfy the new **coincident** constraint. If the link that is being dragged is part of a larger mechanism, the constraint system animates the connected links under their constrained mobilities as demonstrated in figure 28. By using the underlying constraint system for simple editing operations, the user interface merges the tasks of editing and simulation into a single, quasi-physical environment.

Unsatisfiable systems of constraints or systems that do not converge present some difficulty for MechEdit. As described in section 3.4.1, it is not possible to differentiate a numerical system that does not converge rapidly from one that has no real solutions. Thus, it is not possible to determine when a breaking policy (section 3.3.5) must be invoked or when the number of iterations performed by the numerical solver should be increased. This problem becomes more difficult as the size of the system of constraints solved numerically increases. As the number of variables grows, Newton-Raphson iteration generally requires longer to converge. This subsequently slows down the animation causing the mouse sampling rate to decrease, and, in turn, the input parameter to vary more dramatically. Since the starting guess becomes worse and worse, convergence takes longer and longer, and ultimately the number of iterations required exceeds the preset limit for time to converge.

To deal with inconsistent systems, we have tried two breaking policies, neither of which is entirely acceptable. For systems in which an inconsistency is detected during an analytic step of the plan, the current constraint is relaxed to create a *stretching* policy. For example, rather than forcing two markers to be coincident, the two markers are placed as close as possible to each other. This is easily done for analytic solutions and is identical to the method used in [ENDE90]. Unfortunately, there is no simple method of extending this to the numerical portion of a solution.

Systems that do not converge during the numerical step can be handled by some form of *searching* policy. If a system of constraints does not converge for a given input, an input closer to the current state of the mechanism is attempted. Thus, if the user turns a crank 20 degrees, and the system does not converge, the solver attempts an input of 10 degrees instead. The step size is reduced until the system can converge in a fixed number of iterations. This solves both the case in which the input step size is too large and the case in which the system has been driven into an inconsistent state. In the former case, large input steps are reduced to a sequence of smaller steps. In the latter case, the mechanism is driven to the limit of its motion. This method can also be used for purely analytic solutions. The problem with this method is that it leads to a "mushy" interface in which the mechanism does not exactly follow the mouse, but slides slowly into position. Also, the computational cost of searching is extremely high, particularly when a numerical solution is necessary since identifying if the system converges requires

Mechanism	Mobile Links	Active Joints	Newton-Raphson Solver Unk	Eqs	Fr/Sec	Mixed Solver Unk	Eqs	Fr/Sec
4-bar	2	3	6	6	370	0	0	2875
6-bar (a)	4	6	12	12	81	0	0	1711
Stress-10	8	12	24	24	14	0	0	968
Stress-20	18	27	54	54	0.3	0	0	445
Stress-30	28	42	84	84	0.07	0	0	282
Stress-40	38	57	114	114	≈ 0.02	0	0	222
Stress-50	48	72	144	114	–	0	0	176
6-bar (b)	4	6	12	12	85	6	6	339
8-bar	6	9	18	18	32	12	12	86
10-bar	8	12	24	24	5.1	14	14	42
Duplex	10	15	30	30	2.9	20	20	17
Chain-4	3	4	9	8	140	5	4	448
Chain-6	5	6	15	12	55	11	8	122
Chain-8	7	8	21	16	23	17	12	45
Chain-10	9	10	27	20	10	23	16	20
Chain-15	15	14	42	30	0.8	38	26	1.0

Figure 30: Performance Results: There is substantial benefit to using analytic solutions when possible; however, the performance of the numerical solver is reasonable for producing interactive animations of smaller systems.

that the largest acceptable number of iterations be attempted. Currently, we have a combination of the two breaking policies which gives an inconsistent interface but provides smoother animations when entirely analytic solutions are possible.

5.2 Performance

Figure 30 presents data that reflects the performance of MechEdit and a comparison of the performance of numerical and mixed numerical/analytic solution methods. The tests were run on an unloaded Silicon Graphics Indigo. The single processor was a 100 MHz MIPS R4000 (revision 3), and the floating point unit was a MIPS R4010. The machine had 32 megabytes of memory and 8 kilobyte instruction and data caches.

The table of figure 30 indicates a collection of mechanisms some of which may be solved entirely analytically and others that require some form of iterative numerical solution. The number of mobile links is the number of links in the mechanism excluding the ground link and the driving input. The number of active joints is the number of joints excluding the joint connecting the driving link to ground. Each mechanism was simulated with two dif-

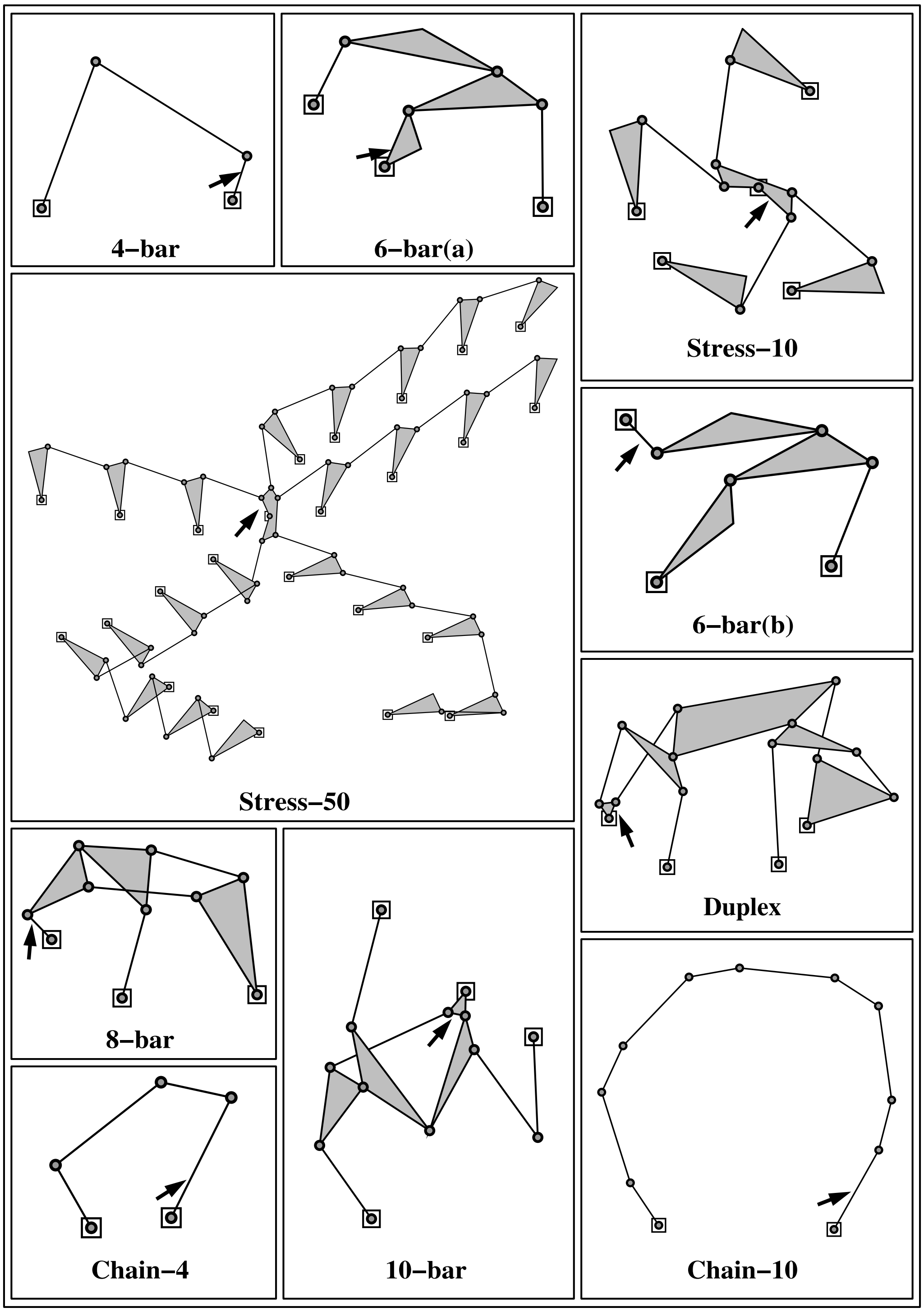


Figure 31: Test mechanisms whose performance is reported in figure 30. The arrows indicate the driven links.

ferent solvers: a Newton-Raphson solver which does no analysis and simply assigns three variables to each mobile link and solves iteratively, and a mixed solver which does as much contraction as possible before solving numerically. The number of unknowns and equations passed to the numerical solver in each case are shown. Each mechanism was driven by a link connected to ground whose location was calculated analytically and does not contribute to the number of equations or unknowns. This input link was rotated continuously about the ground joint and the number of frames per second which the constraint solver produced was recorded.

To isolate the results from the cost of rendering, the animation frames were not displayed during these tests. The locations of all links was calculated and the models maintained by the user interface and the constraint solver were updated, but the screen was not refreshed.

The mechanisms that were animated are shown in figure 31. The 6-bar was driven from two different joints corresponding to the examples demonstrated in sections 3.3.2 and 4.2. The Stress-n mechanisms are collections of n links connected as a chain of 4-bar mechanisms and demonstrate the maximum performance of purely analytic solutions. Because of the poor performance of the numerical solver for large systems, it was impossible to accurately measure the performance of the Stress-50 mechanism. The Chain-n mechanisms are simple chains of n links connected to ground at their ends by revolute joints and demonstrate the performance of the numerical solver on under-constrained systems.

6 Related Work

6.1 Overview

Despite their limited success in the marketplace, constraint systems have roused considerable interest in the research community. The palette of topics studied in the general field of constraint systems is extremely broad as is the degree of success enjoyed by various projects. Freeman-Benson, *et al.* ([FREE90]) have categorized work in constraint systems into five areas which are repeated here to place our work in context: geometric layout; simulations; design, analysis, and reasoning support; user interface support; and general-purpose programming languages. [FREE90] also provides an excellent bibliography of fundamental work in each of these areas.

Geometric layout involves the placement of geometry based on standard geometric relationships. Included within the broad category are Ivan Sutherland's "Sketchpad" ([SUTH63]), Alan Borning's "ThingLab" ([BORN79]), Greg Nelson's "Juno" ([NELS85]), James Gosling's "Magritte"

([GOSL83]), and the work of Glenn Kramer ([KRAM92]) and Eric Enderton ([ENDE90]). MechEdit also falls into this category. Most of these tools, through their ability to simulate mechanisms, may also be viewed as lying in the category of simulation.

Simulation is an extremely broad category for applications of constraint-based systems. [FREE90] includes within this category, the simulation of linkages, circuits, bridges, planetary motion, and the like. While any system that permits basic geometric constraints such as coincidence of points can then be used to simulate the kinematics of mechanisms, we would prefer to narrow this category to include only physical simulations that require some notion of the laws of physics such as mass, momentum, energy, and laws of conservation. Constraint systems of this sort include ThingLab ([BORN79]), Alan Barr and Ronen Barzel's work in constructing physical animations ([BARZ88]), and the work of Andrew Witkin and Michael Gleicher ([GLEI93]) in their physically based constraint-satisfaction methods.

Design, analysis, and reasoning support is a general category consisting of constraints that are used to analyze and improve designs. While it is a somewhat inexact grouping, it appears that what is meant by this category are constraint systems that are used primarily to help locate solutions to design problems rather than simulating solutions that a user might design. Thus, systems that analyze and characterize general traits and expert systems would find their way into this category as would systems that focus on design optimization. Such systems are not directly related to our work.

User interface support is an obvious application of constraints since constraints provide a direct method of specifying the complex relationships between geometry on the screen and underlying data. The comlex task of propagating consistency information between different views of that data or specifying the requirements that geometry on the screen must satisfy in response to a user's actions may be offloaded to a constraint system. [FREE90] reports a number of systems that play various roles within user interface systems including [CART84], [MYER87], [SZEK88], [VAND88], and [EGE87].

Finally, some research has been done in the construction of *general-purpose languages* that use constraints. Nelson's "Juno" ([NELS85]) formulates a figure editor in terms of a general constraint language. William Leler's "Bertrand" ([LELE88]) is a constraint language based on augmented term rewriting. Other systems based on logic programming reported by [FREE90] are described in [JAFF87], [COLM87], [HEIN87], and [DINC88].

6.2 Geometric Constraint Systems

Despite the large number of projects that fall under the overly inclusive heading of constraint systems, only a few have direct bearing on the work performed here and will be described in more detail. Attention will be focused on more recent work despite its grounding in earlier efforts.

6.2.1 Analytic Solvers for Mechanisms

MechEdit is an extension of work performed by Eric Enderton ([ENDE90]). [ENDE90] describes a two-dimensional mechanism editor which allows the direct manipulation and immediate animation of certain classes of mechanisms, in particular, those that can be solved entirely by the method of contraction described in section 3.3. The goal of rapid synthesis and simulation of mechanisms and the requirement of interactivity imposed by Enderton were built upon in MechEdit.

As described in sections 1.1 and 3.3, Enderton's mechanism editor suffered shortcomings in the classes of mechanisms it could animate. In particular, it could only animate linkages that had a single degree of freedom, were generic, and were reduceable to a collection of class II groups. Enderton's solution method is entirely analytic, requiring no iterative numerical methods.

A more recent explication of an almost identical method was introduced to the mechanical engineering community in [ROMD92] and termed *dyad search*. Romdhane describes a system called KAMEL which uses Dijkstra's algorithm for finding the shortest path in a graph to dissect the mechanism graph into dyads which are then solved in a manner equivalent to that of Enderton.

The structure of the code in MechEdit is based on the work of Glenn Kramer described in [KRAM92]. Kramer defines a method called *degrees of freedom analysis* in which rigid bodies are iteratively placed in space according to the constraints imposed upon them. If a body cannot be placed exactly using a single constraint, its degrees of freedom can be reduced, and its remaining freedoms used to determine the locus of positions in which points on this body may lie. Thus, for example, if a body is connected to a ground link by a spherical joint, any point on that body must lie somewhere in a sphere about that joint. By intersecting such loci, it may be possible to exactly position a body in space. Kramer ([KRAM92]) uses complex geometric reductions to describe the freedoms of a body and their intersections, and table look-up to identify when such intersections yield useful geometric reductions.

In two dimensions, loci are one- or zero-dimensional. Zero-dimensional loci represent points and thus represent fully constrained bodies. One-dimensional loci are constrained to be lines and circles. The code fragments that compute the intersections of these various loci are equivalent to the dyad solvers in Enderton ([ENDE90]) and Romdhane ([ROMD92]) above. Thus, in two dimensions, these methods are essentially equivalent. In three dimensions, the number of geometric reductions possible and the complexity of these reductions is substantially increased. [KRAM92] presents a complete method for solving the systems of constraints required for animating three-dimensional mechasims made up of lower-pairs when such systems can be solved in closed form. Examination of his work clarifies the complexity of the task of enumerating analytic reductions and indicates the need for more general techniques.

6.2.2 Iterative Numerical Solvers

Numerous systems have used iterative numerical techniques for solving systems of constraints either through relaxation techniques or some form of Newton iteration. Sutherland's "Sketchpad" and Borning's "ThingLab" both use variations of propagation combined with global relaxation. Both recognize the need to avoid numerical techniques where possible and yield planning phases that identify where solutions can be propagated directly.

Nelson's "Juno" ([NELS85]) is a language for defining figures in which constraints are used in a substantial way. The underlying solver to Juno is based on Newton-Raphson iteration, and the language has facilities for providing a starting guess to Newton's method. This leads to interesting properties of the language, in particular, the fact that the language is non-deterministic. Juno provides an interesting dual-view environment for designing figures in which a user may view both the code and the figure it represents simultaneously. Altering the figure will change values in the code, and changes to the code effect changes in the figure.

L. J. Gutkowski ([GUTK92]) describes the particular use of singular value decomposition for mechanism analysis in combination with Newton's method. His method is entirely numerical but recognizes the particular advantages of SVD in kinematic analyses including the analysis of under- and over-constrained mechanisms and identification of idle degrees of freedom, as well as handling such degeneracy. He also describes that this method may be carried over into general force analyses. Gutkowski's work is framed entirely in the analysis of mechanisms, but its value can easily be carried over into general constraint systems.

6.2.3 Physically Based Constraints

One final approach that appears to have significant merit uses constraints based on physical modeling. By defining constraints in terms of physical notions of force and velocity, constraint maintenance becomes a task of simply driving a system forward using the laws of physics. Thus, each constraint imposes a force, and this collection of forces drives the model iteratively toward a goal state.

Ronen Barzel and Alan Barr ([BARZ88]) introduced the idea of *dynamic constraints* in which objects are specified by geometric constraints, and then the models assemble themselves in accordance with the rules of physics. Constraints are implemented as forces between points, like rubber bands of rest length zero that constrain the coincidence of points. Constraint satisfaction is simply the process of permitting the forces imposed by these constraints to be summed and acted out on the model. Over time, the model will converge toward, although not necessarily reach, a position in which all constraints are satisfied.

Michael Gleicher and Andrew Witkin's "Snap Together Mathematics" ([GLEI90]) provides an underlying mathematical system on top of which a differential constraint system has been constructed ([GLEI93a]). This sytem is used to identify how values should be changing at any given point in time in order to ultimately satisfy a given set of constraints. Using this underlying ordinary-differential-equation solver, Gleicher and Witkin have constructed a variety of applications that demonstrate the general applicability of such techniques including a planar-linkage editor, a three-dimensional editor for links connected by spherical joints, and a variety of tools that demonstrate the use of constraints to simplify user interface design and the associated maintenance of consistency between multiple views ([GLEI93], [GLEI93b], [GLEI92]). The use of physical laws for determining motion has obvious benefit for providing intuitive response in under-constrained systems.

6.2.4 Comparison

As might be expected, the systems described reap the benefits and suffer the disadvantages of their underlying constraint systems. Kramer's degrees-of-freedom analysis is extremely efficient for solving a limited domain of problems but is hindered by substantial programming complexity and limited extensibility. Systems such as Juno and that of Gutkowski are more general but have far worse performance and are more limited by the size of the system that can be animated interactively. Gleicher's system suffers similar performance problems but has a more interactive feeling because the system of equations used to iteratively converge are physically based. Thus, the system can display each of the iterations as the solver converges toward a solution. Each of these systems focusses on one technique for solving constraints and their strengths and weaknesses vary based on this choice. It is our belief that proper combination of these techniques is the key to a powerful, general-purpose constraint system.

7 Discussion and Future Work

7.1 The Merits of Constraints

The paradigm of a physical workspace in which a mechanical or architectural designer invents and explores in a smooth interactive way is extremely powerful. In an interactive setting, the lengthy process of hypothesizing, designing and constructing experiments, and drawing conclusions from them is reduced to a few motions of the mouse. Intuition about a problem and the merits of different possibilities may be drawn far more quickly in an interactive environment than in an environment that permits only a small number of designs to be examined. "What-if" exploration is encouraged, and rapid prototyping takes on a new meaning. In such an interactive paradigm, the designer may focus on exploring the design space, rather than maintaining constraints, visualizing results, and attempting to convey his efforts to a machine shop.

Constraints are the language of design and as such, can be an extremely valuable tool when placed in the hands of a designer in the proper form. Constraints can provide a concise, declarative language for specifying geometry and relationships between geometry. Consider most of the geometry in the furnishings of a room, the mechanisms in a kitchen, or the architecture of a building and you will find a varied collection of objects, but relatively simple, easily described geometry; however, conveying this geometry to a CAD system is seldom simple. This disparity appears to arise because of the low-level language with which designers are expected to communicate with their CAD systems. Constraints represent a higher level language through which commonly used relationships such as parallelism, incidence, and coincidence can be conveyed with simple commands or can be introduced as defaults ([BIER86]).

Beyond the value of constraints as a description language, geometric constraint systems that can maintain constraints as various parameters of a design are altered are a boon to the engineer or architect who must simultaneously consider the many requirements of a design. One of the most difficult tasks for a designer is to avoid breaking one part of a complex design while trying to fix or improve another. The task is clearly one of searching a constrained space. If a constraint solver can enforce the existing constraints, the designer is free to explore the remaining space rather than trying to avoid its boundaries. Avoiding part-to-part interferences, maintaining connectivity between parts, and adjusting parts for manufacturability all represent feasible goals for constraint-solving systems which would provide substantial benefit to the user.

Constraint maintenance provides a designer with a *live* design into which his intentions have been built. Every creation is acccompanied by a multitude of arbitrary decisions made by the designer. Efforts to modify a design require that the arbitrary decisions be culled from those decisions mandated by the constraints of the problem at hand. A model that includes and maintains these constraints takes a large step toward incorporating design intent within a design. The resulting model is *live* in the sense that a change to one part of the design may force automatic modifications elsewhere in the model. In the ideal, a designer need only consider satisfying new constraints locally and the constraint solver insures constraint maintenance globally.

Finally, constraints can be made intuitive if they are combined with a good user interface and are used with restraint. Constraints should be thought of as an additional tool in the tool box of the designer. Like a chain saw, constraints are extremely powerful for certain tasks, but they cannot replace simpler, more direct approaches where such exist. In a complicated design, a user may wish for the constraint solver to assist him in placing geometry. In a simple design, the requirement that a user specify the position of geometry using constraints will probably be far more tedious than simiply placing the geometry directly. It is also important that constraints be incorporated seamlessly into a user interface. Ideally, the user should never need to think in terms of the algebraic constraints he is imposing. A user who snaps two links together with a revolute joint expects those links to be held together as they would be held by a hinge. The concept of coincidence constraints and rotational degrees of freedom are completely hidden from him.

7.2 The Failure of Constraint Systems

Constraint systems have been a research topic for 30 years yet have made minimal penetration into the commercial CAD market. There is no single reason for this failure, yet there are a few that appear to dominate.

Constraints are frequently promoted as a panacea by researchers in the field. Research in this area that embraces constraints to the exclusion of the advances that have been made in intuitive user interfaces for editing and design fails to recognize the positive role that constraints can fulfill. Whether a user wishes to simply place two faces of an object parallel to one another or constrain those objects to be parallel is a choice best left in the hands of the user. The tendency to ignore the limitations of constraints and force the user to conduct every aspect of his work through constraint specification has produced numerous systems in which constraints are viewed as a tedious hindrance rather than as a powerful addition to functionality.

This tendency to force the complexity of constraints where it is not necessary or desirable is far more likely in a two-dimensional environment than in a three-dimensional setting. Two-dimensional designs are generally far less complex and frequently present a manageable challenge for a designer. Users are extremely capable of visualization in two dimensions, and thus are less likely to need assistance in managing geometry. Constraint systems for two-dimensional figure editors and the like provide more power than is necessary; unfortunately most work in constraint systems has been limited to two dimensions. Three-dimensional design and visualization is significantly more complex. Very little progress has been made in the user interfaces of three-dimensional editors and interactive CAD systems, perhaps because manipulating three-dimensional objects in the two-dimensional space of a video screen is inherently non-intuitive. In this three-dimensional setting, substantial complexity, in terms of visualization and manipulation, is introduced. Such complexity demands a high-level language for interaction and presents a golden opportunity for constraint-based systems. Because of this ability to drastically reduce complexity, three-dimensional constraint systems will likely find a captive user base.

The presentation of constraints as a cure-all has also led to an unforgiving exposure of their limitations. Designers of constraint systems must focus on the generality that they can provide. The ability to solve many problems is not as valuable as the ability to solve a particular class of problems. Constraints provide a form of declarative programming, and it is the requirement of the constraint system designer to limit the expressiveness of his declarative programming language to the set of problems for which he can provide a solution. If a user can specify numerous problems for which the constraint system cannot produce a satisfactory solution, the constraint system will annoy the user far more than it assists him. If, however, for a certain easily recognizable set of problems, a constraint system always succeeds in providing useful solutions, its functionality will be exploited frequently.

Finally, one of the key limiting factors of constraint systems, and perhaps the one that is most important in determining the efficacy of constraint systems in general, is performance. Constraint systems that do not operate at interactive speeds yield an unnatural interface. The concepts of *design exploration* and *live designs* are meaningless if the constraint system cannot keep pace with the commands of the designer. The computational complexity of constraint solving has typically made systems too slow to be useful. The advances that have been made in hardware performance combined with efforts to reduce the complexity of the algorithms used for constraint solving appear to be capable of overcoming this limitation.

7.3 Future Work

There is substantial work to be done to raise constraint systems to a level that fulfills the requirements of interactivity. The primary hindrance to high performance for large systems in our current effort is the performance of singular value decomposition and the convergence of the multidimensional Newton-Raphson solver. We have expended little effort in identifying methods of lessening the bottleneck imposed by singular value decomposition. The incorporation of parallelism may offer some improvement. The convergence of Newton's method also has not been examined sufficiently. Convergence properties are difficult to characterize; however, attempts to alter the form of the equations to improve convergence may yield faster solutions.

Because of the relatively poor performance of iterative numerical solutions, a finer grained integration between analytic and numerical solutions could improve performance. A one-degree-of-freedom mechanism, in the limit, may be expressed as a single variable. The ability to rephrase Newton-Raphson in terms of this single variable and replace the linear equation solving in the inner-loop with an analytic solution would be a boon to computational complexity and scalability.

Three-dimensional applications provide the true test of the viability of constraints, yet very few complex three-dimensional constraint systems exist. Numerical algorithms extend easily to three dimensions but because of their poor scaling characteristics, performance problems are accentuated. The number of degrees of freedom for each link doubles (from three to six) forcing singular value decomposition to deal with matrices twice as large. Analtyic algorithms scale well but the additional programming complexity is substantial. The potential to rephrase the geometric problems presented algebraically and to convert complex geometric analyses into general algebraic reductions may hold promise for reducing this programming complexity. We intend to expand MechEdit to three dimensions and examine the power of constraints in the context of three-dimensional editing and modeling.

Ultimately, we wish to explore the viability of implementing a black-box geometric constraint library that could be inserted beneath any application that might benefit from the added functionality of constraints, including figure editors, window managers, solid modelers, virtual reality systems, CAD systems, and the like. Geometry is a standardized language for discourse about objects in space, and maintenance of geometric constraints is a requirement, at some level, of almost every application that manipulates geometry. The rapidly increasing role of graphical applications and the increasing requirements of such systems to provide interactive three-dimensional manipulations strongly suggests the need for continued work in interactive geometric constraint systems. The computational complexity of the task will continue to make it an interesting academic pursuit for many years.

Acknowledgements

This work was performed under the guidance of Carlo Séquin whose enthusiasm and intellectual support underscore the positive environment of his research group. This work is based upon research begun by Eric Enderton whose original ideas played a key role in determining the direction of this project. Thanks to Roger Bush and Steve Burgett upon whose class project the underlying constraint solver was built, and in particular to Roger, whose willingness to serve as the "user-interface guy" shielded me from a great deal of dirty work.

Thanks to NEC Corporation whose generosity partially supported this research. Many thanks to Richard Fateman whose reading of this report and comments upon it have provided numerous ideas for continuing and improving upon this research. Discussions with Glenn Kramer and Michael Gleicher were extremely helpful and provided useful perspectives on other research being done in this field. Thanks also to my colleagues at Berkeley, Glenn Adams, Ajay Sreekanth, Dan Rice, Maryann Simmons, Laura Downs, and Rick Bukowski, who have provided no end of assistance, discussion, and inspiration and, when absolutely necessary, have referred me to cutting-edge research performed by the ancient Greeks.

Finally, thanks to Ramiro Valadez and the rest of the kids on the Mavericks under-12 boys soccer team for making sure that I kept a little perspective and inevitably making life more fun and meaningful, and ultimately, thanks to my parents, Mary and Gerald Brunkhart, who have supported every endeavor I have undertaken, even those that take me 3000 miles away.

References

[BARZ88] Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constraints. In *Computer Graphics*, volume 22, pages 179–188. ACM SIGGRAPH, ACM Press, August 1988.

[BIER86] Eric Allan Bier and Maureen C. Stone. Snap-dragging. In *Computer Graphics*, volume 20, pages 233–240. ACM SIGGRAPH, ACM Press, August 1986.

[BORN79] A.H. Borning. *ThingLab: A Constraint-Oriented Simulation Laboratory*. PhD thesis, Computer Science Department, Stanford, March 1979.

[CART84] C.A. Carter and W.R. LaLonde. The design of a program editor based on constraints. Tech. Rep. CS TR 50, Carleton University, May 1984.

[COLM87] A. Colmerauer. An introduction to Prolog III. Draft, Groupe Intelligence Artificielle, Universite Aix-Marseille II, November 1987.

[DINC88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Bertheir. The constraint logic programming language CHIP. In *Proceedings of International Conference on Fifth Generation Computer Systems, FGCS-88*, 1988.

[EGE87] R.K. Ege. *Automatic Generation of Interactive Displays Using Constraints*. PhD thesis, Department of Computer Science and Engineering, Oregon Graduate Center, August 1987.

[ENDE90] Eric Enderton. Interactive type synthesis of mechanisms. Master's thesis, University of California at Berkeley, April 1990. Report No. UCB/CSD90/570.

[FETZ90] James H. Fetzer. *Artificial Intelligence: its scope and limits*, volume 4 of *Studies in cognitive systems*. Kluwer, Norwell, MA, 1990.

[FORS77] G.E. Forsythe, M.A. Malcolm, and C.B. Moler. *Computer Methods for Mathematical Computations*, chapter 9. Prentice-Hall, Englewood Cliffs, NJ, 1977.

[FREE90] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[GLEI92] Michael Gleicher. Integrating constraints and direct manipulation. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, pages 171–174, March 1992.

[GLEI93] Michael Gleicher. A graphics toolkit based on differential constraints. Preliminary version submitted to UIST 93, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213-3891, May 1993.

[GLEI93b] Michael Gleicher. Practical issues in graphical constraints. In *Proceedings PPCP-93: Workshop on the Principles and Practice of Constraint Programming*, April 1993.

[GLEI90] Michael Gleicher and Andrew Witkin. Snap together mathematics. Technical report, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213-3891, August 1990.

[GLEI93a] Michael Gleicher and Andrew Witkin. Supporting numerical computations in interactive contexts. Technical report, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213-3891, 1993.

[GOSL83] James Gosling. *Algebraic Constraints*. PhD thesis, Carnegie-Mellon University, May 1983. Published as CMU Computer Science Department Tech. Rep. CMU-CS-83-132.

[GUTK92] L.J. Gutkowski. Using singular value decomposition in the analysis of over and underconstrained mechanisms. In *Mechanism Design and Synthesis*, volume 46 of *Design Engineering*, pages 661–669, New York, NY, 1992. ASME Design Technical Conferences, American Society of Mechanical Engineers.

[HEIN87] N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) programmer's manual. Tech. rep., Computer Science Department, Monash University, 1987.

[JAFF87] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Principles of Programming Languages Conference*, pages 196–218, January 1987.

[KAHA89] David Kahaner, Cleve Moler, and Stephen Nash. *Numerical Methods and Software*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[KRAM92] Glenn A. Kramer. *Solving Geometric Constraint Systems: a case study in kinematics*. Artificial Intelligence. MIT Press, 1992.

[LELE88] Wm. Leler. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley, Reading, Mass., 1988.

[MYER87] B. Myers. *Creating user interfaces by demonstration.* PhD thesis, Computer Science Department, University of Toronto, 1987.

[NELS85] G Nelson. Juno, a constraint-based graphics system. In *SIGGRAPH '85 Conference Proceedings*, pages 235–243, July 1985.

[NOBL77] Ben Noble and James W. Daniel. *Applied Linear Algebra.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 2nd edition, 1977.

[PHIL90] Jack Phillips. *Freedom in Machinery.* Cambridge University Press, New York, NY, 1990.

[PRES92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, New York, NY, 2nd edition, 1992.

[REUL76] Franz Reuleaux. *The Kinematics of Machinery. Outlines of a Theory of Machines.* Macmillan, London, 1876.

[ROMD92] Lofti Romdhane. A dyad search algorithm for solving planar linkages using the dyad method. In *Mechanism Design and Synthesis*, volume 46 of *Design Engineering*, pages 49–54, New York, NY, 1992. ASME Design Technical Conferences, American Society of Mechanical Engineers.

[SUTH63] Ivan Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pages 329–345, 1963.

[SZEK88] P. Szekely and B. Myers. A user-interface toolkit based on graphical objects and constraints. In *Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 36–45, September 1988.

[VAND88] B.T. vander Zanden. *An Incremental Planning Algorithm for Ordering Equations in a Multilinear System of Constraints.* PhD thesis, Department of Computer Science, Cornell University, April 1988.

[WILK71] J.H. Wilkinson and C. Reinsch. *Linear Algebra*, volume II of *Handbook for Automatic Computation*, chapter I.10. Springer-Verlag, New York, 1971. Chapter by G.H. Golub and C. Reinsch.