The science is in the technique, all the rest is just commentary.

— Allen Newell
SCS Distinguished Lecture, Dec, 1991

Chapter 3

Differential Techniques

This chapter introduces the basic techniques required to implement the differential approach. We begin by reviewing how graphical manipulation can be viewed as an equation solving problem. To solve these equations differentially, we will solve a constrained optimization that computes rates of change of the parameters given the desired rates of change of the controls. Basic methods for solving these optimizations will be discussed.

Constrained optimization computes the rate of change of object parameters. To determine the objects' trajectories, an ordinary differential equation (ODE) must be solved from an initial boundary value. Some of the basic issues in solving such equations as well as some methods will be introduced.

Additional flexibility is provided by adding additional terms into the constrained optimization problems. This is used to provide default behaviors for objects and to permit the creation of soft controls that can be used to express preferences.

The chapter concludes with an alternate solving method that sacrifices generality for simplicity, a simple example worked through in detail, and a summary of the symbols and methods discussed. The subsequent chapters describe how these methods can be implemented in an efficient and flexible manner.

3.1 The Differential Optimization Problem

In the introduction, the basic notion of treating graphical manipulation as an equation solving problem was introduced. We control graphical objects by specifying what happens to the values of selected attributes called controls. These controls are defined by functions,

$$\mathbf{p} = \mathbf{f}(\mathbf{q}), \tag{3.1}$$

where $\mathbf{q}$ is the state vector of the objects, $\mathbf{p}$ is the vector of values of the controls, and $\mathbf{f}$ is the function that defines the controls. A full table of all mathematical symbols used in this chapter is provided on page 62.

As was discussed in Section 1.1.5, it is not practical to solve Equation 3.1 for $\mathbf{q}$ given $\mathbf{p}$. Instead, we take a differential approach to the problem, as described in Section 1.2. Rather than specifying values for the controls, we will specify how they are changing over time. At particular instants in time, we compute how the state vector must change in order to achieve the desired changes in the controls.

Given a particular instant in time, the value for the state vector at that instant ($\mathbf{q}$), and the desired values for the rate of change for the controls ($\dot{\mathbf{p}}$), we must compute the necessary rate of change of the state vector($\dot{\mathbf{q}}$). We call this problem the *differential optimization*. Since the value for the state and the control function are given, we also know the value of the controls at the instant the optimization is to be solved.

Our need to deal with the time derivatives of the controls and state variables leads us to take the derivatives of each side of Equation 3.1 to yield

$$\dot{\mathbf{p}} = \frac{d\mathbf{p}}{dt} = \frac{d\mathbf{f}(\mathbf{q})}{dt}. \tag{3.2}$$

Applying the chain rule yields

$$\dot{\mathbf{p}} = \frac{\partial \mathbf{f}}{\partial \mathbf{q}} \frac{d\mathbf{q}}{dt}. \tag{3.3}$$

For the general case of a vector of control functions, the derivative is a matrix called the *Jacobian,* which is the matrix $\partial \mathbf{f}/\partial \mathbf{q}$ and is denoted by $\mathbf{J}$. Using this notation, we get

$$\dot{\mathbf{p}} = \mathbf{J}\dot{\mathbf{q}}. \tag{3.4}$$

Like the controls themselves, the Jacobian $\mathbf{J}$ is a function of the state variables. The matrix of functions that compute the elements of the Jacobian can be determined by differentiating the control functions with respect to the variables. Since the values for the state variables are given, the value of the Jacobian is effectively given as well. Methods for computing the Jacobian efficiently will be discussed in Section 5.2, but for now, we simply assume we have some method to compute the Jacobian from the variables, and treat is as if it were a given as well.

Since $\mathbf{J}$ is a known matrix, Equation 3.4 is a *linear* equation, even though $\mathbf{p}$ is a non-linear function of $\mathbf{q}$. The differential approach has replaced the multi-dimensional non-linear root-finding problem with a linear system and an ordinary differential equation. Unlike non-linear equations, for which good solving techniques are unlikely to exist, linear systems are relatively easy to solve.

3.1.1 Underdetermined Cases

Unless enough controls are specified to uniquely determine a solution, Equation 3.4 will be underdetermined. There will be many possible ways for the state vector to change to achieve the desired changes in the controls. As discussed in Section 1.1, the system must select one of the ways for things to change. We must use some heuristic to pick a solution, because we lack any information as to what is desired. The rule chosen for the differential approach is to minimize the amount that the configuration changes, or, more precisely, to minimize the rate of change of the configuration. That is, if the user's controls don't ask for something to change, the system should avoid changing it. This leaves open a variety of ways to measure change that will be explored in Section 3.4. Since the rate of change of the configuration will be a linear function of the rate of change of the variables, its magnitude will be a quadratic function, which we denote by g.

When the linear system of Equation 3.4 does not uniquely determine $\dot{\mathbf{q}}$, it provides constraints on its possible value. To determine the particular value of $\dot{\mathbf{q}}$, we must solve the problem

$$\text{minimize } E = g(\dot{\mathbf{q}}) \text{ subject to } \dot{\mathbf{p}} = J\dot{\mathbf{q}}. \tag{3.5}$$

That is, we have cast the problem as a constrained optimization: minimize the value of a quadratic objective function of $\dot{\mathbf{q}}$, subject to the linear constraints that the controls are met. In the following sections, we discuss solution methods using standard techniques that meet the needs of differential manipulation.

3.2 Solving the Differential Optimization

The linear/quadratic constrained optimization problems are a standard class of problems for which a wide range of techniques have been developed. Good surveys can be found in texts such as [Fle87] and [GMW81]. A standard technique is the Lagrange multiplier method. A form of it is reviewed here for use in the differential approach.

To begin, we consider minimizing a specific quadratic objective function, simply minimizing one half the magnitude of $\dot{\mathbf{q}}$ squared. The value of $\dot{\mathbf{q}}$ that minimizes that is the same value that minimizes the magnitude of $\dot{\mathbf{q}}$. The specific constrained optimization problem we consider in this section is then

$$\text{minimize } E = \frac{1}{2}(\dot{\mathbf{q}} \cdot \dot{\mathbf{q}}) \text{ subject to } \dot{\mathbf{p}} = \mathbf{J}\dot{\mathbf{q}}. \tag{3.6}$$

We will consider the general case of quadratic objectives in Section 3.4.

To provide an intuition for how Lagrange multiplier methods work, consider an extremely simple case: a particle in 2 dimensions, with its state represented as its Cartesian coordinates, $\mathbf{q} \equiv \{x, y\}$. We will place a control on the particle that is its distance to the origin, $p = f(\mathbf{q}) = x^2 + y^2$. Suppose we specify $\dot{p}$ to be 1.

As shown in Figure 3.1, there are many possible values for $\dot{\mathbf{q}}$ which achieve the desired value for $\dot{p}$. In this case, it is clear to see that the one with smallest magnitude is the one which is in the same direction as the gradient of f. Any component of $\dot{\mathbf{q}}$ not along this line will not be helping to achieve the desired controls. We can therefore restrict $\dot{\mathbf{q}}$ to be some multiple of the gradient, that is, $\dot{\mathbf{q}}$ can be expressed as a scaling factor times the gradient. This scaling factor is called the *Lagrange Multiplier.*

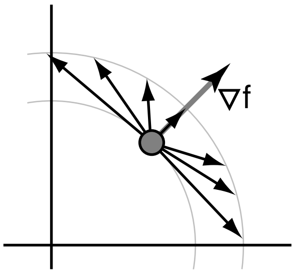

Figure 3.1: A point on the plane with a radial control. Many possible values of $\dot{\mathbf{q}}$ will yield the desired value of $\dot{p}$. The one with least magnitude has the same direction as the gradient of f.

If there were multiple controls, each would make a contribution to $\dot{\mathbf{q}}$. For each control, the contribution is some multiple of its gradient. We therefore have a vector of Lagrange multipliers, which we denote by $\boldsymbol{\lambda}$. $\dot{\mathbf{q}}$ is determined by the linear combination $\mathbf{J}^{\mathbf{T}}\boldsymbol{\lambda}$.

More formally, to be a solution to the constrained minimization problem, $\dot{\mathbf{q}}$ must satisfy two criteria. First, $\dot{\mathbf{q}}$ must satisfy the linear constraints, given by Equation 3.4. Secondly, $\dot{\mathbf{q}}$ must minimize E as much as possible, subject to the constraints. In the case of an unconstrained minimization, we would require that the gradient $\partial E/\partial\dot{\mathbf{q}}$ vanish, meaning that there is no direction to change $\dot{\mathbf{q}}$ that would result in a lesser value for E. With constraints, there might be a way to change $\dot{\mathbf{q}}$ to further minimize E, but only if these changes are prohibited by the constraints. That is, if the gradient of the objective function is not zero, it must lie in the row space of the constraint gradients. This requirement is expressed by defining the objective function gradient to be a linear combination of the constraint gradients,

$$\frac{\partial E}{\partial \dot{\mathbf{q}}} = \mathbf{J}^T\boldsymbol{\lambda}, \tag{3.7}$$

for some value of $\boldsymbol{\lambda}$. The vector $\boldsymbol{\lambda}$ is an intermediate result which we call the *Lagrange multipliers.*

In the case of the simple objective function of Equation 3.6, the gradient $\partial E/\partial\dot{\mathbf{q}}$ is simply $\dot{\mathbf{q}}$, giving

$$\dot{\mathbf{q}} = \mathbf{J}^T\boldsymbol{\lambda}. \tag{3.8}$$

Substituting this into Equation 3.4, gives

$$\dot{\mathbf{p}} = \mathbf{J}\mathbf{J}^T\boldsymbol{\lambda}, \tag{3.9}$$

a linear system which can be solved for its one unknown, $\boldsymbol{\lambda}$. This intermediate result can be substituted back into Equation 3.8 to yield the desired final result, $\dot{\mathbf{q}}$.

3.2.1 Over-determined Cases

To this point, we have focussed on techniques that handle the cases where an insufficient set of controls are specified to uniquely determine a solution. We now must consider the problem of handling cases where too many controls specify the solution. Such cases may involve redundant controls, where multiple controls all specify the same solution, or conflicts, where there are no solutions to all of the controls.

Conflicting controls are obviously a problem as there is no solution which will meet the controls. However, redundant controls manifest themselves in exactly the same way. Consider a system subject to two identical controls p_1 and p_2. The net result should be that $\dot{\mathbf{q}}$ moves in the manner specified. But $\dot{\mathbf{q}}$ is created by the sum of the contributions of p_1 and p_2. How much does each contribute? Does p_1 contribute a little and p_2 a lot? Does p_1 contribute a huge positive amount and p_2 a huge negative amount?

When controls are over-specified, whether their values conflict or not, they cause the Lagrange multipliers to be under-specified. The matrix $\mathbf{JJ^T}$ will be singular. Redundant or conflicting controls are inevitable, and are notoriously hard to detect. It is important that our solution method be robust in the face of these singularities, and that it will do something reasonable with conflicts.

One approach to handling the over-constrained cases would be to employ a linear system solver which could handle Equation 3.9 even when the matrix is singular. For example, singular value decomposition (SVD) [PFTV86] could be used. The SVD has many attractive properties, for example it provides information as to which controls are redundant. Unfortunately, SVD is expensive to compute. In contrast, if we can restrict the problem so that the solver only needs to solve non-singular systems, we can exploit this property to solve them efficiently, as will be discussed in Chapter 4.

Rather than force the solver to handle singular matrices, we will instead modify the matrices so that they have a unique solution. We will not get an exact solution to the original linear system, but we are trading accuracy for improved behavior in bad cases. Because we are interested in interaction, rather than high accuracy quantitative methods, we will make such tradeoffs often, as discussed further in Section 4.5.

The technique for making the matrices non-singular is called *damping*. The basic intuition is that we generally prefer to avoid large values for the Lagrange multipliers, therefore, in cases where the Lagrange multipliers are undetermined, we should minimize their magnitude. The derivation here most closely follows that of Nakamura's derivation of a robust pseudo-inverse [Nak91].

In cases where the controls are over-determined, the solver will not be able to achieve all the desired values for them. Instead, we must settle for getting as close as possible, that is, to satisfy them in a least squares sense. To find Lagrange multipliers that achieve this minimum, we minimize $1/2(\mathbf{J}^T\boldsymbol{\lambda} - \dot{\mathbf{q}}) \cdot (\mathbf{J}^T\boldsymbol{\lambda} - \dot{\mathbf{q}})$, In addition, we would like to minimize the magnitude of $\boldsymbol{\lambda}$, although since this is not as important, we can scale this term by a small amount, which we will call μ. The function we wish to minimize is

$$E = \frac{1}{2}(\mathbf{J}^T\boldsymbol{\lambda} - \dot{\mathbf{q}}) \cdot (\mathbf{J}^T\boldsymbol{\lambda} - \dot{\mathbf{q}}) + \mu(\boldsymbol{\lambda} \cdot \boldsymbol{\lambda}). \tag{3.10}$$

The minimum of this quadratic is found by differentiating with respect to $\boldsymbol{\lambda}$, and setting that equal to 0, yielding

$$0 = \mathbf{J}\mathbf{J}^T\boldsymbol{\lambda} - \mathbf{J}\dot{\mathbf{q}} + \mu\mathbf{I}\boldsymbol{\lambda}, \tag{3.11}$$

where **I** is the identity matrix. Recalling Equation 3.4, a little rearrangement yields

$$\dot{\mathbf{p}} = (\mathbf{J}\mathbf{J}^T + \mu\mathbf{I})\boldsymbol{\lambda}, \tag{3.12}$$

a variant of Equation 3.9 which has small amounts added to the diagonal of the matrix.

Rather than having a single scaling factor for the magnitude of the vector of Lagrange multipliers, we could have an individual one for each individual multiplier. This would enable damping selectively, or to damp some controls more than others. Selective damping allows the creation a a limited constraint hierarchy: if two constraints conflict, and one is damped but the other is not, the undamped constraint will dominate the damped one. When two conflicting controls are both damped, their effects are blended. By individually adjusting their damping values, the controls can be weighted. The larger the damping value, the less weight the control receives.

The damping technique presented here has three major drawbacks. First, it penalizes large values of the multipliers whether they are underdetermined or not. This can cause a problem when the multipliers legitimately need to be large to satisfy the desired controls. Secondly, damping is applied whether there are conflicting controls or not. Finally, it introduces a new dimensionless parameter μ. Because it has no real meaning to the original problem, values for it are difficult to determine. For the differential approach, damping values must be determined empirically.

3.3 Solving the Differential Equation

The methods of the previous section permit us to compute the rates of change of the state vector. We now consider how to use the rates to find the trajectories of the configurations over time. We must solve the problem of computing the trajectory of the state given its initial value and time derivatives, a problem of solving an ordinary differential equation (ODE) from an initial boundary condition. Here, we provide a brief introduction to handling this problem in the context of the differential approach. For a more complete, but still practical, introduction to ODE solution methods, see Chapter 15 of [PFTV86].

The value of **q** is actually is a function of time, defined by a function that computes its time derivative. The form that we have this function defined in is

$$\dot{\mathbf{q}} = \mathbf{f}'(\mathbf{q}, t), \tag{3.13}$$

where $\mathbf{f}'$ is found by solving the differential optimization. Given the value for $\mathbf{q}$ (which corresponds to $\mathbf{q}(t)$ for some time t), we can compute $\dot{\mathbf{q}}$. This is a standard form for an ODE.

For the types of problems we encounter with the differential approach, we cannot solve the ODE in closed form. Instead, we must solve it numerically by discretizing time into a series of small steps. In computing a step, the following problem must be solved: given the state at the current time, $\mathbf{q}(t)$, find the state at some time in the future, $\mathbf{q}(t+\Delta t)$. The time derivative $\dot{\mathbf{q}}$ (i.e. the result of the differential optimization) does not directly provide the solution to this problem. It only specifies how the state is changing at the instant that it is computed. The problem of updating the state given the ability to find its time derivatives is solving an ordinary differential equation from an initial boundary condition.

Solving the ODE is difficult because when we perform an evaluation to find $\dot{\mathbf{q}}$ for a particular $\mathbf{q}$, we are only finding out about a particular instant in time . We have no information about the future. $\dot{\mathbf{q}}$ might remain constant for the duration of the step, but it might also change drastically over the course of the step.

The simplest method for solving an ODE is to find $\dot{\mathbf{q}}$ at the beginning of the step and assume it remains constant over the course of the step. This is known as Euler's Method, and has the simple update rule of

$$\mathbf{q}(t + \Delta t) = \mathbf{q}(t) + \Delta t \dot{\mathbf{q}}(t). \tag{3.14}$$

Euler's method approximates $\mathbf{q}(t)$ with as a piecewise linear function. The size of each piece is the step size. If the step is too large, the approximation will not be good. Notice that each step requires computing a new $\dot{\mathbf{q}}$ by solving the differential optimization.

To understand what is meant by "good" in ODE solving within the context of the differential approach, consider a simple example. Once again, we will use a point in the plane, however, this time, we will select a control that is its angular position about the origin. Suppose we provide a desired velocity for this control of 1 unit per unit time, the starting configuration has the point a unit distance from the origin along the positive x axis, and the specified derivative always points tangent to the circle. As time progresses, we will expect the point to move around the origin in a circular path.

If an Euler's method ODE solver is applied to this example, the problems of ODE solving are quickly apparent. At the initial position on the x axis, the gradient of the control points vertically. Any step in this direction will lead the point off the circle it is expected to follow around. As more and more steps are taken, the point will continue to spiral away from the circle, speeding off the page, as shown in Figure 3.2.

Because of error in approximation, the point spirals outward over time. However, if smaller step sizes are taken, the behavior is better. That is, the point spirals outward more slowly, better approximating the expected circle. This is shown in Figure 3.3. In fact, by going more slowly, we may reach a desired destination more quickly because we are less likely to overshoot or drift away from the target. Going slowly can be

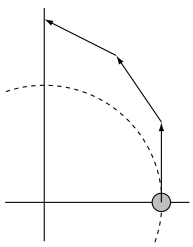

Figure 3.2: A point on the plane with a control that drives it tangent to a circle around the origin. Although it should (ideally) travel in a circular path, ODE solver error causes it to spiral off the page.

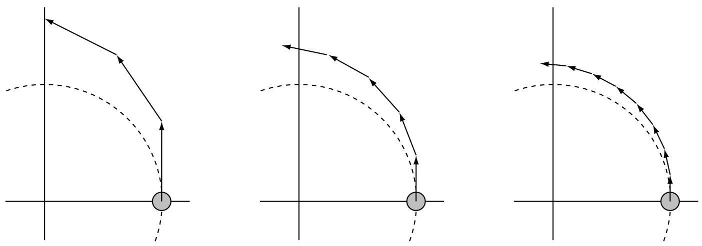

Figure 3.3: A point is pushed in a direction tangent to a circle about the origin, using an Euler ODE solver and various step sizes. In all cases, the point spirals away from the target circle, although with small step sizes, the point tracks the circle better.

accomplished two ways, either by reducing the duration of the step or by reducing the velocities.

In solving the ODE, there is effectively a speed limit. If an object attempts to change faster than this, it may speed out of control or even miss its destination entirely. For a given velocity, solving can be made more stable by reducing the step size, in the limit of infinitesimally small step sizes, ODE solving will be exactly correct. However, since each step may require significant computation, the number that can be executed is limited. Alternatively, this speed limit can be viewed with a constant step size: for a given step size, how fast can an object go without becoming unstable. If we think of steps as taking some fixed amount of time to compute, this translates directly to an apparent velocity in the image.

The speed limit given by ODE solving varies according to a number of factors. Most significantly, it depends on the path that the objects take. The more non-linear the function is, the more poorly the linear approximation will fit it. At the extreme, if an object is truly moving in a line, Euler's method achieves the exact motion, and there is no speed limit. In a sense, the speed limit can be viewed as a restriction on the types of controls used to define the motion: given that the step size is fixed, how badly non-linear a control can be used and still have the object move at a reasonable rate.

If we take smaller steps to achieve better performance, we might use multiple ODE solver steps for each redraw. For example, if we would like to maintain 10 frames per second and updating the image or solving the differential optimization takes 30ms, we might use two Euler steps between redraws. As we will see in Chapter 4, typically the differential optimization is the most time consuming part of the process, and becomes more so as the problems grow larger. With a faster computer, the time to compute each step will be decreased, so more steps can be computed per redraw, effectively raising the speed limit.

Using multiple samples per step is what is called *multi-step solving*. We might phrase the problem as follows: the starting point (t and $\mathbf{q}(t)$), compute $\mathbf{q}(t + \Delta t)$ as well as possible using n samples. Using two Euler steps has $n = 2$, and uses a 2 piece piecewise linear approximation.

Given that we have some number of samples that we can make in a step, we can consider how to best use these samples to approximate $\mathbf{q}(t)$. For example, if we can take 2 samples, we might use a 2 piece linear approximation by taking two Euler steps. Alternatively, we might use these same two samples to fit a parabola. This would be called a *2nd-order* method. The particular case of a parabola is simple to create, since a parabola has a linear function for its derivative. This is effectively done by using the initial step as a trial step, evaluating the derivative at this point, and using this for the duration of the step. This is called the *midpoint method* or the 2nd-order Runge-Kutta method. It is applied to the example problem in Figure 3.4.

According to Press et al.[PFTV86], the most popular multi-step method is the 4th-order Runge-Kutta method. As implied by the name, it uses 4 evaluations per step. According to the literature, this method is generally perceived to be superior to higher order methods. The 4th order Runge Kutta method has been the preferred solver for the prototype implementations in this thesis.

A higher order method is only better than taking a larger number of lower order steps if it provides a higher speed limit for the same number of evaluations. This will, of course, depend on the problem to be solved. However, in practice, the 4th-order Runge-Kutta method seems to be a good method for implementing the differential approach. Empirically, it usually performs at least as well, but sometimes substantially better, than taking 4 small Euler steps, or 2 Runge-Kutta 2 steps.

There are many other multi-step methods. Predictor-Corrector techniques [PFTV86] are another popular strategy. Such methods use past steps to predict future values and then correct for the error of the prediction. However, these methods are difficult to apply in dynamic settings because the dynamic nature of the problems make it difficult to maintain a history to use in prediction. Often, there will be no history so some technique like Runge-Kutta will be needed to start the process.

For the prototype implementations of this thesis, fourth order Runge-Kutta and Euler's methods solvers are used.

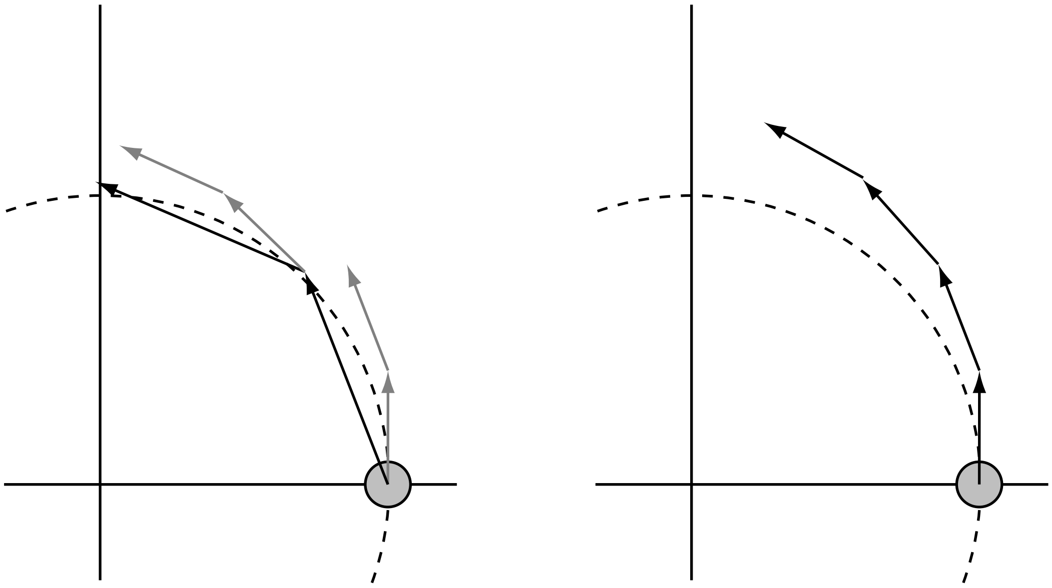

Figure 3.4: Left, a second order Runge-Kutta method, and right, an Euler's method are used in the example problem of Figure 3.2. The evaluations used by the Runge-Kutta solver are shown in grey. Notice how the Runge-Kutta solver stays closer to the circle using the same number of evaluations as the Euler solver.

3.3.1 Adaptive Step-Sizes

So far, we have considered solving an ODE with fixed step sizes. Instead, we might consider adapting the step size to the problem. When a step is made, it could be checked to see how good it was. If it caused an unacceptably large amount of error, it could be redone with a smaller step size. Adaptive step size methods have the advantage that they can slow down to accurately handle problems when they become difficult. Also, because they check their results, they are less likely to cause bad errors.

Adaptive step size methods have some severe drawbacks when used with the differential approach. The most significant problem is that they are continually adjusting the step size which alters the amount of computation required to advance simulation time a specified amount. If the computation rates are fixed, the apparent velocities of objects will fluctuate. This can be disconcerting to the user. Also, the extra evaluations to perform checks and computing alternate steps might be better spent on making more steps since error correction is built into the controls, as controllers can adjust their values in response to what is happening as will be discussed in Section 6.3. For example, in the example of the previous section, if the user really cared about the point staying on the circle, an additional control that maintained this would be used.

The differential approach provides some interesting opportunities for employing adaptive step sizes. Standard methods for ODE solving treat the equation as a black box, that is they cannot get any information about the problem other than asking for evaluations of $\dot{\mathbf{q}}$. Standard adaptive ODE solvers employ methods such as taking the same step with a higher order method to compare with.

With the differential approach, we have more information about the problem we are solving. In particular, the functions that define the controls provide measures of error. For example, if a controller is meant to keep a control at a particular value, at the end of the step it can be checked to insure that the control has not changed too much. Each different control might have its own way of defining what an acceptable amount of error is. I call this *semantic adaptation* because it adapts based on the meaning of the problem. I have experimented with some simple semantic adaptation of step size, simply reducing the step size when a control has a value that the system finds unacceptable. The method works as follows: a certain set of controls are monitored. When a step is computed, the monitored controls are examined. If any exceed a specified error limit, the step size is shortened.

A different type of semantic adaptation is to use a different step when problems occur. One useful variant of this is the cleanup step. In an application like constraint-based drawing or mechanism simulation, there is typically some small number of controls that cause motion and a potentially larger number that represent constraints. A cleanup step is an extra step that is run only with the constraints. It is used when the pulling controls have broken the other constraints to cause them to get back to their desired configuration.

3.4 Generalized Objective Functions

The optimization objective determines which solution will be given in under-determined cases. By selecting different optimization objectives, different default behaviors can be given to objects. To this point, we have only consider one optimization objective: one half the magnitude of the state vector derivative squared. This section considers other objective functions.

The types of default behavior that we consider in the Differential Approach can be summarized by the idea that objects should not change unless a control causes them to change, and that when an object changes to achieve what is specified by a control, it should do so by changing as little as possible. By altering how we measure change, we can control the default behavior or feel of an object. For an example, consider manipulating a line segment by moving one of its points, as shown in Figure 3.5. Depending on the metric of change, the line segment will behave differently. In all cases, the line segment achieves what is specified by the controls. However, by selecting an appropriate metric, the programmer can create a desirable default behavior. An appropriate metric is not essential since if there was something that was important, it could be specified with a control. But, properly defined objectives can alleviate the need for extra

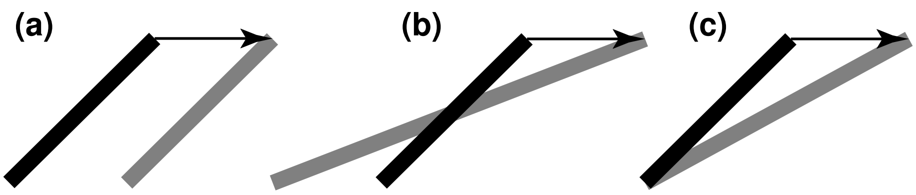

Figure 3.5: A line segment is dragged by controls that specify the position of the upper point. Different objective functions provide different behaviors: a) change in length and orientation are minimized; b) change in the position of the center is minimized; c) change in the position of the lower point is minimized.

specifications.

The simplest metric of change is the magnitude of the rate of change of state vector, $\dot{\mathbf{q}}$, as was used in Section 3.2. This simple objective has been used up to this point, and is sufficient for a wide variety of applications. It has a drawback: it causes the parameterization to affect the behavior of the object. Using the parameterization violates the goal of separating manipulation from representation. Even simple decisions, such as whether to represent an angle by degrees or radians, can affect an object's behavior [Wit89a]. This may be a serious problem, or an opportunity. It means that we can choose which interactive behavior we would like by carefully choosing the representation. However, if we are not careful about choosing the representation, we might get a less desirable behavior. The severity of this problem is limited, because the user could always provide additional controls if they really cared what happened.

3.4.1 The Metric

In order to spare the user the increased effort of more completely specifying their intent, and to give programmers more freedom to select representations that are convenient, we must use a different objective function. Rather than measuring change in values of the parameters, we could measure change in something that did not depend as closely on the representation. We have used the functions that compute objects' attributes to serve as controls that are independent of representation. Similarly, we prefer to define an object's metric in terms of its attribute functions as well. We select a subset of the attributes to define the metric. Just as we denote the subset of the attributes that serve as the controls as **f**, we will denote the function used to define the metric by **g** which is also a function of **q**. We denote the Jacobian $\partial \mathrm{g}/\partial \mathrm{q}$ by **G**.

The optimization objective will be to minimize the magnitude of the change in the attributes. This rate of change is

$$\dot{\mathbf{g}} = \mathbf{G}\dot{\mathbf{q}}. \tag{3.15}$$

Since we are searching for the minimum, we can minimize $1/2$ the sum of squares of

$\mathbf{g}$, rather than the magnitude. Since we might wish to emphasize some of the attributes more than others, we also introduce a scaling factor for each to make the objective a weighted sum of squares. Writing the scaling factors as the diagonal elements of a matrix for notational convenience, the objective function is

$$E = \frac{1}{2}\dot{\mathbf{q}}^{\mathbf{T}}\mathbf{G}^{\mathbf{T}}\mathbf{S}\mathbf{G}\dot{\mathbf{q}}, \tag{3.16}$$

We will call the matrix that defines this quadratic ($\mathbf{G}^{\mathbf{T}}\mathbf{S}\mathbf{G}$) term the *metric* because it defines a way to measure $\dot{\mathbf{q}}$.

The simple objective function of Section 3.2 used the parameters of the objects as the attributes that defined the metric. Since $\mathbf{g} = \mathbf{q}$, $\mathbf{G} = \mathbf{I}$ and $\mathbf{M}$ also is the identity matrix. Viewed this way, the advantage of using a correct metric can be seen. The identity metric defines the behavior of the object in terms of its parameters, rather than in terms of something that is potentially meaningful to the user.

A Particularly Useful Metric

The metric provides a method for an interface designer to define a default behavior for an object. In effect, it allows for hand-tuning the behavior that a user sees when the object is manipulated. However, this leaves the problem that the interface designer must hand-tune the behavior in order to hide effects of the parameterization. Often, this is not an issue, as the parameters provide a reasonable default behavior, or, if a specific behavior is required, it can serve to describe a metric. However, some applications demand an automatic method for determining a metric that provides a consistent, parameterization independent, feel for a variety of objects. Such a method is particularly useful in cases where a user may define object behavior. An example is the parametric curve manipulation of Section 8.1.1 and Section 9.4.

An analogy to physics provides an automatic, consistent metric for a broad class of objects. As first suggested by Witkin [Wit89b], we can imagine an object as a physical entity with an uniform mass distribution. In effect, we can view each pixel of the object as an atom, each with a tiny bit of mass. This mass distribution defines how the object changes as forces are applied in particular places. Inertia causes each particle to move as little as possible. The mass distribution serves as the metric does, defining the behavior of objects in response to controls. Witkin and Welch [WW90] used specification of the mass distribution to allow animators to specify the the default behaviors of simulated objects that were acted upon by point controls.

When solving the equations of motion of a physical object in generalized coordinates in order to simulate it, the mass distribution is encoded into a matrix known as the inertia tensor or mass matrix [Gol80]. This matrix is found by accounting for the effects of each particle on the objects' behavior by integrating over the mass distribution. When such a matrix is determined numerically, the integral is approximated by sampling a set of particles.

Analogously, a metric can be defined by viewing a graphical object as a collection of "particles" and minimizing the motion of these particles. In practice, the distribution is estimated by a set of points. We define the metric functions to be the positions of an evenly spaced set of points on the object. We call such a metric the *mass matrix* because of its physical analog.

The mass matrix is an important metric because it can be defined independently of the object. For any graphical object, a set of points can be evenly distributed either along its length (for a curve) or within its area (if it is solid). Section 8.1.1 will illustrate the utility of this, allowing default behavior to be automatically provided for a wide variety of objects.

3.4.2 Solving the Generalized Quadratic Objective

Using a different quadratic optimization objective requires a slightly different set of methods for solving the constrained optimization problems. In this section, we derive the method in its full generality for the case of any quadratic objective function and linear constraints.

The standard form of a vector quadratic equation is

$$E = \frac{1}{2}\mathbf{x}^{\mathbf{T}}\mathbf{M}\mathbf{x} + \mathbf{b}^{\mathbf{T}}\mathbf{x} + k, \tag{3.17}$$

where **x** is the vector parameter ($\dot{\mathbf{q}}$ for this chapter), **M** is the quadratic or matrix term, the vector **b** is the linear term, and k is a scalar constant. Since we are not interested in the value itself, but rather only the value of **x** that minimizes it, we can ignore the constant as it goes away when we take the gradient, and we can multiply the quadratic by $1/2$ as it cancels out other values later, simplifying the equations. The linear term permits measuring change from a point other than 0, and will be used in Section 3.5. The methods of Section 3.2 solve the special case of this objective with an identity matrix for **M**, and 0 for the linear component **b**.

The Lagrange multiplier derivation can again be applied, this time to Equation 3.17. We denote the linear constraints as $\mathbf{A}\mathbf{x} = \mathbf{a}$. We require the gradient of the objective function to be a linear combination of the of the constraints,

$$\frac{\partial E}{\partial \mathbf{x}} = \mathbf{M}\mathbf{x} + \mathbf{b}\mathbf{x} = \mathbf{A}^{\mathbf{T}}\boldsymbol{\lambda}. \tag{3.18}$$

Solving this for **x** and denoting the inverse of the metric M^{-1} by **W**, gives

$$\mathbf{x} = \mathbf{W}\mathbf{A}^{\mathbf{T}}\boldsymbol{\lambda} - \mathbf{W}\mathbf{b}. \tag{3.19}$$

Inserting this into the constraint equation $\mathbf{A}\mathbf{x} = \mathbf{a}$ gives

$$\mathbf{A}\mathbf{W}\mathbf{A}^{\mathbf{T}}\boldsymbol{\lambda} = \mathbf{a} + \mathbf{A}\mathbf{W}\mathbf{b}, \tag{3.20}$$

a linear system that can be solved for $\boldsymbol{\lambda}$. Once $\boldsymbol{\lambda}$ is computed, it can be inserted into Equation 3.19 to compute $\mathbf{x}$.

The damping techniques of Section 3.2.1 are not taken into account by the generalized quadratic objective. Using a derivation similar to that of Equation 3.12, yields

$$(\mathbf{AWA^T} + \mu\mathbf{I})\boldsymbol{\lambda} = \mathbf{a} + \mathbf{AWb}. \tag{3.21}$$

Using the notation of the rest of the chapter, $\mathbf{M}$ is the metric as defined in Section 3.4.1 and the linear constraints are given by Equation 3.4 so $\mathbf{A} = \mathbf{J}$ and $\mathbf{a} = \dot{\mathbf{p}}$.

3.4.3 An Approximation to the Metric

Using the metric to define the default behavior of an object has several advantages. It permits separation of manipulation and representation, and provides an abstraction for defining the feel of an object. However, it has a significant cost: we must find the metric and invert it. This is problematic because the metric is large. Inverting a matrix this large would be prohibitive. One advantage to using the identity matrix as the metric is that it is trivial to invert.

What we aim for in this section is an approximation to the metric that is inexpensive to invert, yet provides some of the features of the full metric. The approximation we consider is simply using the diagonal elements of the metric. This diagonal matrix is trivial to invert — we merely take the reciprocal of all its elements — and cheap to use in solving Equation 3.20. It still addresses some of the important issues that the full metric addresses, particularly the selection of units.

Consider again the example of dragging a line segment in Figure 3.5. Suppose its configuration is represented by the position of its center, its length and its orientation, and that the simple identity metric objective function is used. If the upper left corner of the segment is move a quarter of an inch to the left, the line segment might have its center move, scale and rotate, or some combination of the two. Suppose that the position of the center of the line segment was represented in micrometers from the corner of the page, the orientation represented as radians from horizontal, and the length in inches. To achieve the movement of the upper left point by simply moving the center would require the parameters to change very quickly as there are many micrometers to be covered, while achieving the movement by scaling and rotating would require considerably smaller changes in the parameters. Because the simple optimization objective minimizes change in the parameters, the latter would be chosen. If the position of the center were measured in miles instead, a very tiny change in the position of the center would create the needed motion, so this would be selected by the optimization criteria.

In the example, the simple selection of units with which to represent the position of the center of the line segment determined the dragging behavior. The problem is parameters having different units. One way around this is to define an objective function which minimized the amount of change in the parameters after converting them to some standardized units. Suppose we knew the conversion factors between the units of the parameters and the standard units. We would have a scaling factor for each parameter. For notational convenience, we can write the scaling factors as the diagonal elements of a diagonal matrix **S**, so the component-wise scaling of parameters would simply be the multiplication **Sq**.

Rather than simply minimizing the magnitude of $\dot{\mathbf{q}}$, we would instead minimize the time derivative of the scaled parameters, $\mathbf{S}\dot{\mathbf{q}}$, giving

$$E = \frac{1}{2}(\mathbf{S}\dot{\mathbf{q}} \cdot \mathbf{S}\dot{\mathbf{q}}), \tag{3.22}$$

or, to use the generalized notation of Equation 3.17

$$\mathbf{M} = \mathbf{S}^{\mathbf{T}}\mathbf{S}. \tag{3.23}$$

We see that we have a diagonal metric.

The problem is to determine **S** to convert the parameters to the standard units. One way to define standard units would be to require that equal changes in each variable should affect the attributes the same amount. As for the metric, we pick a subset of the attributes to define the objective function, and denote the function that computes these attributes by **g**. However, we are only interested in the derivatives with respect to a single variable at a time. That is we want to measure the change in all of the attributes of **g** with respect to each variable independently. For a particular variable, the scaling factor is the magnitude of the derivatives of each element of **g** with respect to the variable, that is,

$$\mathbf{S}_{ii} = \sqrt{\frac{\mathbf{g}}{\mathbf{q}_i} \cdot \frac{\mathbf{g}}{\mathbf{q}_i}}. \tag{3.24}$$

The diagonal terms in Equation 3.23 are the scaling factors squared,

$$\mathbf{M}_{ii} = \frac{\mathbf{g}}{\mathbf{q}_i} \cdot \frac{\mathbf{g}}{\mathbf{q}_i}. \tag{3.25}$$

These are exactly the diagonal terms of the metric in Equation 3.16.

The diagonal metric cannot take into account interactions between variables. While it can remove differences in units between similar terms, it cannot make two different representations seem alike. In the line segment example, it eliminates the effects of the choice of units, however it does not remove the effect of a completely different parameterization. It could not, for example, express an objective function that minimized the motion of the endpoints. Therefore, no matter what diagonal metric are chosen, a line segment parameterized by position of center, length and orientation will feel different than a line segment parameterized by the positions of its endpoints.

3.5 Soft Controls

The generalized objective functions of the last section allow the creation of objective functions that can be used to control the object. To this point, we have discussed default behaviors for objects that causes objects to minimizes their motion if controls are causing them to change, they do not change. In this section we consider an alternative: having objects change by default unless a control specifies otherwise. The non-zero defaults lead to linear terms in the general quadratic optimization objective function of Equation 3.17. Using this term will enable soft controls: controls which are overridden by the regular controls. These are important because they allow us to create behaviors such as dragging subject to constraints, where an object is manipulated by the user but constraints will not be violated. While these techniques do not provide general constraint hierarchies as described by Borning et al.[BFBW92], the dragging subject to constraints that they can provide is useful in many graphical applications. The use of optimization objective terms to provide user control was pioneered in the vision research of Kass et al.[KWT88].

Suppose that we had some desired default value for $\dot{\mathbf{q}}$, denoted $\dot{\mathbf{q}}_\mathbf{0}$. Rather than simply minimize the magnitude of $\dot{\mathbf{q}}$, we would instead minimize its difference from the default value, so

$$E = \frac{1}{2}(\dot{\mathbf{q}} - \dot{\mathbf{q}}_\mathbf{0}) \cdot (\dot{\mathbf{q}} - \dot{\mathbf{q}}_\mathbf{0}). \tag{3.26}$$

In terms of the generalized objective function of Equation 3.17, the coefficients of the linear term $\mathbf{b}$ in this case is $\dot{\mathbf{q}}_\mathbf{0}$. Similar metrics can be worked out to include a metric as well. While the generalized solution of Section 3.4.2 can be applied, we review the derivation for this important special case here as it provides insight.

To provide intuition for how this works, consider again the simple point example from Section 4.3. Notice that the control specifies the behavior only in the direction of its gradient, so the optimization objective is free to do whatever it wants in an orthogonal direction. Suppose that we have specified $\dot{p}$ to be 0. This restricts $\dot{\mathbf{q}}$ to lie along the line perpendicular to the gradient, as shown in Figure 3.6. To find $\dot{\mathbf{q}}$ closest to $\dot{\mathbf{q}}_\mathbf{0}$, we must project $\dot{\mathbf{q}}_\mathbf{0}$ onto this line. We do this by adding in a component of $\dot{\mathbf{q}}$ which cancels out the disallowed portion. This *constraint component* must be a multiple of the gradient. This multiple is the Lagrange multiplier.

We compute $\dot{\mathbf{q}}$ as the sum of two components, its default value $\dot{\mathbf{q}}_\mathbf{0}$ and the contributions of the controls, $\dot{\mathbf{q}}_\mathbf{c}$, so

$$\dot{\mathbf{q}} = \dot{\mathbf{q}}_\mathbf{0} + \dot{\mathbf{q}}_\mathbf{c}. \tag{3.27}$$

Since $\dot{\mathbf{q}}$ must satisfy the controls, we substitute this into Equation 3.4, to yield

$$\dot{\mathbf{p}} = \mathbf{J}(\dot{\mathbf{q}}_\mathbf{0} + \dot{\mathbf{q}}_\mathbf{c}). \tag{3.28}$$

Since the contribution of the constraints is a linear combination of the control gradients,

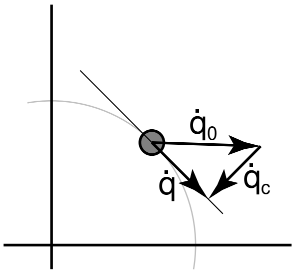

Figure 3.6: A hard controls constrains the distance from the point to the origin. Any movement of the point must be orthogonal to the gradient of this control. When a default velocity ($\dot{\mathbf{q}}_0$) is given for $\dot{\mathbf{q}}$, it must be projected into the space that meets this restriction. To achieve this, a component is added to $\dot{\mathbf{q}}_0$ that projects it onto the space where $\dot{\mathbf{p}} = 0$.

we define the Lagrange multipliers as

$$\dot{\mathbf{q}}_{\mathbf{c}} = \mathbf{J}^T \boldsymbol{\lambda}. \tag{3.29}$$

Which, with a little rearrangement yields

$$\dot{\mathbf{p}} - \mathbf{J}\dot{\mathbf{q}}_{\mathbf{0}} = \mathbf{J}\mathbf{J}^T \boldsymbol{\lambda}, \tag{3.30}$$

a linear system, which like Equation 3.9 can be solved for $\boldsymbol{\lambda}$, which in turn determines $\dot{\mathbf{q}}_{\mathbf{c}}$ by Equation 3.29, from which $\dot{\mathbf{q}}$ can be computed by Equation 3.27. Notice that when $\dot{\mathbf{q}}_{\mathbf{0}} = 0$, the method of this section is exactly the same as that of Section 4.3. The damping techniques of Section 3.2.1 also apply.

3.5.1 Determining the Values for Soft Controls

We now must figure out how to obtain $\dot{\mathbf{q}}_{\mathbf{0}}$. Our goal is to provide soft controls that work as the hard controls do, except that the hard controls are given precedence over them. Soft controls are defined as $\mathbf{p_s} = \mathbf{f_s}(\mathbf{q})$, but like the hard controls, would be specified by their derivatives, $\dot{\mathbf{p}}_{\mathbf{s}}$.

If the soft controls do not conflict with the hard controls, they can simply be treated as hard controls. The more interesting cases, however, are when the hard controls limit the soft controls. The ultimate goal is to have have soft controls work exactly as hard controls do, except in the cases where there are hard controls that take precedence over the soft controls.

We would like to satisfy the soft controls as closely as possible subject to the restriction that the hard controls are specified exactly. We can define the objective function to minimize the squared error of the soft controls meeting their desired values

$$\text{minimize } E = \frac{1}{2}(\mathbf{J_s}\dot{\mathbf{q}} - \dot{\mathbf{p}}_{\mathbf{s}}) \cdot (\mathbf{J_s}\dot{\mathbf{q}} - \dot{\mathbf{p}}_{\mathbf{s}}) \text{ subject to } \dot{\mathbf{p}} = \mathbf{J}\dot{\mathbf{q}}. \tag{3.31}$$

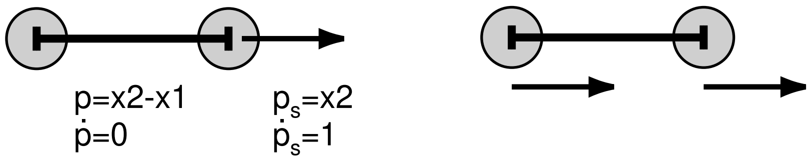

Figure 3.7: Two points are connected by a hard control constraining their distance. The right point is also pulled by a soft control. If the soft controls are computed independently, a non-optimal solution may be found, as shown on the left. The soft control may specify that the right point should move, but part of this motion might be removed by the constraints. As shown on the right, there is a solution that both satisfied the constraints and meets the desired values for the soft controls.

Such an objective function is the similar to the definition of the metric in Section 3.4.1, except that rather than minimizing the magnitude squared of the change of the derivatives of the functions value, we minimize the magnitude squared of the difference between the derivative's value and a default value. Generalized sets of soft controls (e.g. $\mathbf{J_s}$) will lead to $\mathbf{M}$ terms in Equation 3.17. The problems in using general metrics also apply to soft controls: A sufficient number of soft controls must be specified to uniquely determine $\dot{\mathbf{q}}$ in all cases, even when there are no hard controls. If an insufficient number of soft controls are specified, $\mathbf{M}$ will be singular. Problems with ill-conditioning and efficiency in inverting $\mathbf{M}$ make this soft control scheme impractical. In this section, we concentrate on methods for simpler achieving soft controls by computing values for $\dot{\mathbf{q}}_0$.

Two-Pass Solver

One way to find $\dot{\mathbf{q}}_0$ is to ignore the hard controls, and simply use the method used for hard controls for the soft controls. Using this approach, two linear systems are solved: first, a linear system is solved to compute the Lagrange multipliers that will determine $\dot{\mathbf{q}}_0$, then a linear system is solved to project $\dot{\mathbf{q}}_0$ into the subspace allowed by the hard constraints.

The method of computing the soft controls independently has a serious drawback: it does not achieve the desired solution. Consider a case where two points are connected with a hard control constraining their distance, and a soft control pulling one of the points to the right, depicted in Figure 3.7. The soft control alone would move one of the points, violating the hard control. When this is projected into the legal subspace, part of the motion world be cancelled out. However, if all the controls were treated equally, the other point could be moved to satisfy the hard control.

An alternative is to account for the hard controls in computing $\dot{\mathbf{q}}_0$. We compute $\dot{\mathbf{q}}_0$ using the methods we would use for the hard controls, except that we include both the hard controls and the soft controls in the computation. Damping must be used in case the controls conflict. We then solve an optimization problem again using the result of the first solution as $\dot{\mathbf{q}}_0$, and just using the hard controls as constraints. I first used this technique in the *Briar* drawing program described in Section 9.1, and therefore call it the *Briar-style solver.*

In cases where all the controls, both soft and hard, are consistent, the *Briar-style* solver has the nice property that both hard and soft controls behave the same. It is also the case that the effort in solving the linear system twice can be avoided by first checking to see if $\mathbf{J}\dot{\mathbf{q}}_0 = \dot{\mathbf{p}}$, in which case $\dot{\mathbf{q}}_c$ will trivially be 0. When using an iterative linear system solver, as described in Section 4.3.1, this check happens automatically.

However, if we knew that the hard and soft controls did not conflict, then there would be no need to have soft controls. In the cases where there are conflicts, the *Briar-style* solver has a few drawbacks. Most obvious is that it requires solving the linear system twice, which can be expensive. Also, since solving the larger system will have conflicting constraints, damping must be used. The solver does not actually solve Equation 3.31, but instead minimizes the difference between the damped result, which already partially accounts for the hard constraints.

Spring Controls

An alternate method to compute $\dot{\mathbf{q}}_0$ is to use gradient descent to drive the soft controls to their desired values. We compute $\dot{\mathbf{q}}_0$ to have the direction of the gradient of the soft control functions, and a magnitude proportional to how far from the target it is,

$$\dot{\mathbf{q}}_0 = k\mathbf{J_s}(\mathbf{p_s} - \mathbf{f_s}(\mathbf{q})), \tag{3.32}$$

where k is a scaling constant. This causes the controls to be pulled towards their desired values with a decaying attraction: as the control nears its desired value, the rate at which it is being pulled is decreased. In the physical analogy of Section 1.2.2, this attraction is a spring. Equation 3.32 is the generalized force version of Hooke's law.

I will call these spring-like controls *spring controls* or simply *springs*. Their method can be viewed as a cheap way to estimate the Lagrange Multiplier, an attempt to use gradient descent to achieve desired values for the soft controls, or as generalized springs, if we view the optimization as a physical simulation. Despite the fact that they are a little harder to justify, they do work very well.

3.6 An Alternate Technique

The methods presented in the last sections described Lagrange multiplier techniques for solving constrained optimization problems with linear constraints and a quadratic objective function. The methods build a linear system and solve for an intermediate result, the Lagrange multipliers. The methods have the advantage that they permit the use of any quadratic objective function.

Often, we do not exploit generality of the Lagrange multiplier formulation. For example, the simple objective functions of Equation 3.6 or Equation 3.26 may be sufficient. In such cases, alternate, special purpose solution methods are sufficient.

The objective function that simply minimizes the magnitude of $\dot{\mathbf{q}}$ gives an important special case called a linear least squares problem. This is a very standard problem in numerical analysis. Example solving methods include singular value decomposition (SVD), QR factorization, and pseudo-inverses. These methods, and many others, are discussed in [GL89]. Unfortunately, these methods are almost all extremely expensive to compute, as they are unable to exploit properties of the problems such as sparsity that we will use in the next chapter to speed performance.

Iterative solvers may also be used to solve the linear least squares problem. In particular, conjugate-gradient solvers, discussed in Section 4.3.1, are relevant to implementing the differential approach. Variants of conjugate-gradient find a solution to the linear system, but have the property that the solution they provide to a linear system is solution closest to the starting point. Therefore, if the solver is begun with a zero starting point, the solution with least magnitude is found. The conjugate-gradient linear system solver in the Numerical Recipes text [PFTV86] is such a solver. To implement soft controls using the conjugate-gradient, Equation 3.28 is solved for $\dot{\mathbf{q}}_{\mathbf{c}}$ using the conjugate-gradient solver.

Using a least squares solver to implement the differential approach effectively solves Equation 3.4 directly, without first computing the Lagrange multipliers. For instance, implementing the differential approach by using a conjugate gradient algorithm works very well. The solver given in the text of Press et al.[PFTV86] permits the two most often used objective functions, and handles over-determined cases by providing a minimum norm residual solution to the linear system. Using the algorithm is a very practical way to implement the differential approach. It performs extremely well in practice. This method served as the backbone of my early implementations, and is available by a run-time switch even in my most current versions.

The obvious question is why bother developing a more complex technique when the simpler approach works so well. The three main reasons for using the Lagrange multiplier techniques in this chapter over the simpler "direct" approaches such as using conjugate-gradient are: the intermediate result of the Lagrange multiplier techniques (the Lagrange multipliers) will be useful in certain interaction techniques such as the active set methods of Section 6.4; they place few restrictions on the linear system solver that is used, so that fast algorithms can be found; and, they extend to other quadratic objective functions. However, in cases where these advantages are not required, the simpler approach is worth considering. The approach handles the two most often used objective functions, those of Equation 3.6 and Equation 3.26, so it is often sufficient.

3.7 A Concrete Example

To review the basic techniques of this chapter, we now consider a complete example in detail. Our object will be a fixed length line segment with a unit radius, represented by the position of its center and its orientation. The control we will create is the position of its endpoint. The state variables are $\mathbf{q} = \{\mathbf{q}_{cx}, \mathbf{q}_{cy}, \mathbf{q}_{theta}\}$, and the controls are $\mathbf{p} = \{\mathbf{p}_x, \mathbf{p}_y\}$.

The control functions are:

$$\begin{aligned} \mathbf{p}_x &= f_x(\mathbf{q}) = \mathbf{q}_{cx} + \cos \mathbf{q}_\theta \\ \mathbf{p}_y &= f_y(\mathbf{q}) = \mathbf{q}_{cy} + \sin \mathbf{q}_\theta . \end{aligned} \tag{3.33}$$

The core of the implementation will be the differential optimization that will compute a value for $\dot{\mathbf{q}}$ given a value for $\mathbf{q}$ and $\dot{\mathbf{p}}$. This routine must first compute the Jacobian of the controls, $\mathbf{J}$, as a function of $\mathbf{q}$:

$$\mathbf{J} = \begin{bmatrix} 1 & 0 & -\sin \mathbf{q}_\theta \\ 0 & 1 & +\cos \mathbf{q}_\theta \end{bmatrix}. \tag{3.34}$$

It then can compute values for the Lagrange multipliers by solving the linear system

$$\dot{\mathbf{p}} = \mathbf{J}\mathbf{J}^{\mathbf{T}}\boldsymbol{\lambda} \tag{3.35}$$

for $\boldsymbol{\lambda}$, and then computing $\dot{\mathbf{q}}$ as

$$\dot{\mathbf{q}} = \mathbf{J}^{\mathbf{T}}\boldsymbol{\lambda}. \tag{3.36}$$

Suppose at the current time, the line was at a 45 degree angle with its center at the origin ($\mathbf{q} = [0, 0, \pi/4]$), and that the control specifies the endpoint to move right with unit velocity ($\dot{\mathbf{p}} = [1, 0]$). The Jacobian of the controls is then

$$\mathbf{J} = \begin{bmatrix} 1 & 0 & -.707 \\ 0 & 1 & .707 \end{bmatrix}. \tag{3.37}$$

To compute the $\dot{\mathbf{q}}$, we must first solve the linear system

$$\begin{bmatrix} 1.5 & -0.5 \\ -0.5 & 1.5 \end{bmatrix} \boldsymbol{\lambda} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \tag{3.38}$$

then use Equation 3.36 to determine $\dot{\mathbf{q}}$. At this particular instant, $\boldsymbol{\lambda} = [.75\ .25]$, so $\dot{\mathbf{q}} = [.75\ .25\ -.35]$. If we were to use an Euler step with step size .1, the configuration at the end of the step would be $\mathbf{q} = [.075\ .025\ .75]$.

The basic control for interaction will be to repeatedly take ODE solver steps, interleaving redraw between the steps to give the illusion of motion. This solver will call the differential optimization routine, possibly several times per iteration. The ODE solver will also have to provide values for $\dot{\mathbf{p}}$ to the optimization routine. These values are what accounts for the user's motions; for example, they might be tied to the input device. Methods for determining desired velocities will be discussed in Section 6.3, however, one simple way of getting $\dot{\mathbf{p}}$ from the input device is to use decaying attraction: we compute the vector from the position of the control to the pointer, and use a multiple of this for $\dot{\mathbf{p}}$.

In this example, we are already using multiple controls, one for each axis. Even more controls could be added. For example, suppose we wanted to add two more controls that position the other end of the line segment. These controls are computed by

$$\begin{aligned} \mathbf{p}_{x2} &= f_x(\mathbf{q}) = \mathbf{q}_{cx} - \cos \mathbf{q}_\theta \\ \mathbf{p}_{y2} &= f_y(\mathbf{q}) = \mathbf{q}_{cy} - \sin \mathbf{q}_\theta. \end{aligned} \tag{3.39}$$

The Jacobian would now be a 4 by 3 matrix.

Since we most likely will not have two mice, rather than permitting the user to control the position of the second point, we might want to constrain it to remain at the origin. This would require specifying $\mathbf{p}_{x2}$ and $\mathbf{p}_{y2}$ to have values that caused the point to move towards the origin, e.g. to be a negatively scaled multiple of the position of the point.

Clearly, these controls will conflict: the mouse might attempt to pull the other endpoint away from the origin. This will cause the matrix $\mathbf{JJ^T}$ to be singular. In order to solve the linear system, we might add damping by adding a small amount to the diagonal elements of the 4 by 4 matrix.

We might instead wish to drag the endpoint subject to the constraint that the other endpoint remains at the origin, that is, the mouse should not be able to rip the other endpoint from its resting point, but other than that, should be able to drag its endpoint as well as possible. To do this, we use soft controls.

To compute the optimization with the soft controls, we first must compute a value for $\dot{\mathbf{q}}_\mathbf{0}$ by computing the Jacobian of the soft controls with Equation 3.40, and multiply this by the desired value of the soft controls. The regular controls are now just the opposite endpoint, so the Jacobian is simply the 2x3 matrix of their derivatives. To compute $\dot{\mathbf{q}}$ we first compute the Lagrange multipliers by solving the linear system

$$\dot{\mathbf{p}} - \mathbf{J}\dot{\mathbf{q}}_\mathbf{0} = \mathbf{J}\mathbf{J}^T\boldsymbol{\lambda}. \tag{3.40}$$

We use that to compute $\dot{\mathbf{q}}$ by

$$\dot{\mathbf{q}} = \dot{\mathbf{q}}_0 + \mathbf{J^T}\boldsymbol{\lambda}. \tag{3.41}$$

3.8 Summary

In this section we review the techniques presented in this chapter for solving the differential optimization problem, summarize the procedure for implementing it, and de-

n	number of controls
m	number of state variables
$\mathbf{q}$	state vector (m-vector)
$\mathbf{p}$	values of the controls (n-vector)
$\mathbf{f}$	function to compute the controls, $\mathbf{p} = \mathbf{f}(\mathbf{q})$
$\dot{\mathbf{q}}$	time derivative of $\mathbf{q}$ (m-vector)
$\dot{\mathbf{p}}$	time derivative of $\mathbf{p}$ (n-vector)
$\mathbf{J}$	Jacobian of $\mathbf{f}$ ($n \times m$ matrix), $\mathbf{J} = \partial \mathbf{f} / \partial \mathbf{q}$
E	optimization objective
$\boldsymbol{\lambda}$	Lagrange multipliers (n-vector)
μ	damping factor (scalar or n-vector)
$\mathbf{q_0}$	default value for $\mathbf{q}$
$\mathbf{p_s}$	soft controls
$\mathbf{f_s}$	function computing soft controls
$\mathbf{J_s}$	Jacobian $\partial \mathbf{f_s} / \partial \mathbf{q}$
$\mathbf{M}$	metric ($m \times m$ matrix)
$\mathbf{W}$	inverse metric $\mathbf{M^{-1}}$ ($m \times m$ matrix)
$\mathbf{g}$	functions whose change is minimized to define $\mathbf{M}$
$\mathbf{G}$	Jacobian $\partial \mathbf{g} / \partial \mathbf{q}$

Table 3.1: Symbols defined in this chapter, and used throughout the thesis.

scribe the caveats as to solving the more general, numerical optimization problem. The symbols used throughout this chapter, and for the rest of the thesis are reviewed in Table 3.1.

The differential optimization takes the current value of the state $\mathbf{q}$, and the functions that define the controls and objectives as givens. From these givens, the current values of the controls and the Jacobians of the controls and objective metric functions can be computed, so they too are considered givens. The procedure is as follows:

1. Compute the $\dot{\mathbf{q}}_0$ value of the force controls, if any, using the damped spring formula of Equation 3.32, or some other methods.

2. Find the metric, $\mathbf{M}$, and its inverse. Often, the identity matrix is used instead. Computing a metric involves computing the Jacobian of $\mathbf{g}$.

3. Compute the Jacobian of $\mathbf{f}$, $\mathbf{J}$.

4. Compute the Lagrange multipliers, using some variant of Equation 3.20. Most likely, some damping will be used on some of the controls.

5. Compute $\dot{\mathbf{q}}$, for example by Equation 3.8.

6. If this computation was to compute both the hard and soft controls in a Briar-style solver (as in Section 3.5.1), remove the soft controls, set $\dot{\mathbf{q}}_\mathbf{0} = \dot{\mathbf{q}}$, and return to step 3. For the second time through, less damping might be used.

The differential optimization solves for $\dot{\mathbf{q}}$, given $\dot{\mathbf{p}}$ and $\mathbf{q}$. It makes use of the control function (**f**) and its Jacobian (which is a function of $\mathbf{q}$). If a metric is to be defined, a set of functions (**g**) and its Jacobian will be needed as well. The process has a single tunable parameter, which is the amount of damping (μ). μ might be a vector if damping values are to be provided for each control.

To emphasize, being able to solve the differential optimization problem

$$\dot{\mathbf{q}} = \mathcal{O}_{(t,\mathbf{f})}(\mathbf{q}, \dot{\mathbf{p}}). \tag{3.42}$$

is not the same as being able to solve the general control equation

$$\mathbf{q} = \mathbf{f}^{-1}(\mathbf{p}). \tag{3.43}$$

In fact, being able to solve Equation 3.42 does not necessarily allow solving Equation 3.43. There are two main reasons for this: we need to know what $\dot{\mathbf{p}}$ will achieve the changes necessary to get the desired values of $\mathbf{p}$; and, once we have $\dot{\mathbf{q}}$, we don't necessarily know what $\mathbf{q}$ is at some future time.

Given a desired value for $\mathbf{p}$, the most obvious way to proceed differentially is to use a $\dot{\mathbf{p}}$ that changes the value as needed, e.g. if the value of a control is too high, make it decrease. However, heading straight for the goal is only a heuristic that can often fail, for example, if there is a local minimum. Also, there is no certainty that there is velocity $\dot{\mathbf{p}}$ that will achieve the desired controls, either if there is no way to achieve the desired controls at all, or if there is simply no continuous path through state space.

Even if a direction for $\dot{\mathbf{p}}$ is known, and the corresponding $\dot{\mathbf{q}}$ that achieves it can be found, there is no guarantee that the correct $\mathbf{q}$ can be found. Finding values of $\mathbf{q}$ requires solving the ordinary differential equation of Equation 3.42 for $\mathbf{q}$.