*The secret to walking on water is knowing where the rocks are.*

— Herb Cohen
Vail Symposium 14 poster

# Chapter 4

# Efficient Solution Techniques

In the previous chapter, we introduced methods for implementing the differential approach. In this chapter, we now consider how to solve the differential optimization problems efficiently.

## 4.1 The Demands of Interactive Systems

Interactive systems place a different set of demands on numerical techniques than more traditional, batch computation applications might.

One unique demand of the numerical problems in interactive systems is that they are dynamic. Because the equations are created in response to the users' actions, they are not known when the system is created. More significantly, the set of equations to be solved is continually changing in response to the user. This dynamic nature of the numerical problem means that we must be able to define equations at run time, which will be addressed in Chapter 5. Solution methods that rely on extensive pre-analysis are to be avoided as the problem being solved may change before the cost of the analysis can be amortized.

A common practice in numerical computations is to adapt the solution methods on a per problem basis. This ranges from experimenting with different algorithms to see which best solves a given problem, to adjusting parameters to make solvers converge. Such per problem tweaking is unacceptable in the setting of an interactive system. Not only is the problem continually changing, but we would like to insulate the user from the mathematics. We do not want the user to have to learn about constrained optimization just to draw a picture.

Speed is an important consideration for numerical routines. For the differential approach, it is critical. If the computations are not fast enough, the system will not be able to provide the smooth motion which is demanded by direct manipulation. We also must be concerned with scalability, that is how the methods will perform as the problems grow larger.

Accuracy, typically an important concern in numerical analysis, is less critical to interactive systems. This is important since there is often a tradeoff between the time a computation takes and how accurate it is. The accuracy required is typically limited by factors such as device resolution. In the cases where users demand sub-pixel accuracy, for example when designing an object that is to be manufactured, the accuracy demands are typically known. Even in applications where high accuracy is required, fast, inaccurate methods are useful if these results can be refined.

While accuracy is not essential for interactive applications, stability is. Numerical instabilities can cause such undesirable effects as objects wobbling or flying off the screen. Stability is, therefore, an important concern for interactive systems.

For the purpose of this thesis, there is an additional goal for the numerical routines. We would prefer techniques that are simple and widely available. For several of the numerical problems we face, for example solving linear systems and ordinary differential equations, a vast array of sophisticated software packages are available, both from public sources and commercial vendors. Development of such algorithms is beyond the scope of this thesis. Similarly, relying on a particular software package would make the approach harder to reproduce and port.[1]

### 4.1.1 Basic Methods for Achieving Performance

There are a few general strategies for improving the performance of the computations. These will be applied in various ways throughout this chapter.

**Trade Accuracy for Performance —** As discussed earlier, in an interactive system, we are often willing to trade accuracy for performance. Techniques for doing this will be discussed in Section 4.5. It is generally *not* acceptable to trade stability for performance.

**Trade Convergence for Iteration Rate —** As a control moves towards a goal value, it is more important that it moves with smooth motion than that it gets to its goal in a minimum amount of time.

**Exploit Sparsity —** The matrices involved in the differential optimization problems are filled mainly with zeros. It is crucial to exploit this fact, both for speed and memory usage.

**Reduce the Problem Size —** In the next section, we will see that the computational complexity of the differential methods is linear in the number of variables and quadratic in the number of constraints. Therefore, to handle larger problems, the size of the numerical problems actually solved must be reduced, while giving the user the illusion that the system is solving the larger problem.

[1] The work of Mark Surles[Sur92a, Sur92b] has such a problem. Anyone wishing to reproduce the results must purchase an expensive sparse matrix package on which the work relies.

**Reuse Previous Results —** Many intermediate results are used by several later computations. Caching such intermediate results can avoid redundant computation. Caching will be discussed in the next chapter.

One other important method for speeding numerical computations is to exploit special cases. For example, extremely efficient algorithms exist for solving n-body dynamics problems, finite elements, diagonal matrices, and the kinematics of articulated chains. However, the goals of the differential approach demand general purpose solutions. We therefore focus on general purpose methods for enhancing performance.

## 4.2 Scalability of the Differential Approach

With the differential approach, it is important that the redraw steps happen fast enough to give the illusion of continuous motion. As the number of objects and controls grows, so does the time required to make a step. In this section, we consider how the performance of solving in the differential approach scales as problems get larger, and identify the bottlenecks in performance. We are primarily concerned with parts of the computation that scale worse than linearly.

The problem size of the differential approach can grow in two ways: the number of controls $(n)$, and the number of variables or objects $(m)$. For complexity analysis purposes we consider variables and objects equivalent because each object will have a small constant number of variables. For all interesting cases, $m > n$, otherwise there will certainly be redundant or conflicting controls.

We consider only the complexity of solving the differential optimization problems. Other parts of the system might scale badly: for example, an input technique might need to examine all pairs of objects (requiring $O(m^2)$ time), or a rendering computation might require solving for interactions among all objects. However, such issues would need to be addressed in non-differential approaches as well. ODE solving, the other part of the differential approach's computation, will require a small constant number of calls to the differential optimization solver for the kinds of ODE solvers we might consider.

With arbitrary controls, the computation costs of the differential approach are almost unbounded. For example, we might have a control on the average center of each combination of 4 objects, requiring a combinatorial explosion just to enumerate the terms in the expression. However, for analysis we make some assumptions that are rarely violated in practice:

1. Objects are independent, therefore, the addition of another object does not change the number of variables an object has, or the amount of time that it takes to compute attributes.

2. Controls are independent, therefore, the addition of another control does not change the number of variables a control depends on or adversely change the amount of time to compute a control's value. Some optimizations, such as notably common subexpression sharing, may speed evaluations.

3. Controls depend on a fixed number of variables, independently of the total number of variables or controls in the system. This restriction eliminates controls on the aggregate of all objects, for example the center of mass of all objects in the world.

From these three assumptions, it follows that $n$ and $m$ are independent. It also follows that the time to compute the values for the controls is $O(n)$, because computing each of $n$ controls cannot depend on either $n$ or $m$. By a similar argument, the Jacobian of the controls can also be computed in $O(n)$ time.

The fact that the Jacobian of the controls $(\mathbf{J})$ can be computed in $O(n)$ time is significant, and non-obvious. $\mathbf{J}$ is an $n \times m$ matrix, so it would take $O(nm)$ time just to fill the matrix with 0s. The key observation is that we do not have to store all the values in the matrix because many of them will be zero, that is, the matrix is *sparse*. Each row depends only on the variables that its control depends on, which is independent of $n$ or $m$. Therefore, the entire matrix will contain only $O(n)$ entries. By exploiting sparsity, this can be stored and accessed in $O(n)$ time. Exploiting sparsity is an important tool in implementing the differential approach.

Computing the differential optimization requires solving a linear system with the $n \times n$ matrix $\mathbf{JJ^T}$. This matrix can be built in $O(n^2)$ time because each element is computed by the dot product of two constant length vectors. Solving the linear system, with a standard method such as Gaussian Elimination, would be an $O(n^3)$ process. This unacceptable asymptotic performance can be improved by exploiting sparsity.

For certain classes of sparse matrices, linear systems can be solved in much less than $O(n^3)$ time. For example, if the matrix has constant bandwidth, solving time is $O(n)$. For certain configurations of controls, the matrices will have this structure. Surles[Sur92a] describes why important problems in molecular biology and other domains have this structure, and describes techniques for solving such constraint systems using methods very similar to the differential approach [Sur92b]. Unfortunately, if we permit constraints among arbitrary objects, as we must for the general differential approach, we do not know the structure of the matrix a priori. In fact, there is no guarantee that the matrix $\mathbf{JJ^T}$ will be sparse.

The way to exploit sparsity without making restrictions on the way objects can be connected is to avoid constructing $\mathbf{JJ^T}$. Many types of iterative linear system solvers, such as the Conjugate-Gradient techniques discussed later, access the matrices only by multiplying them by a vector. Using the associativity of matrices, the multiplication $\mathbf{JJ^Tx}$ can be achieved by doing two matrix by vector multiplies. Each of these would take $O(n)$ time because that is the number of entries are in the matrix $\mathbf{J}$. Technically,

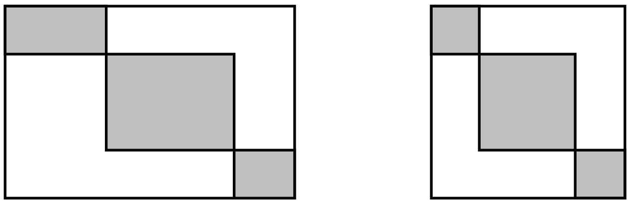

**Figure 4.1:** Sparsity patterns depicted by filling potentially non-zero elements with grey. Left: Since each object defines a few functions to contribute to the metric, and these functions depend only on the object's variables, the Jacobian will be *block-sparse* with a rectangular region for each object. Right: When the Jacobian is multiplied by its transpose, the blocks do not interact with each other, leading to independent squares along the diagonal of the matrix. This form of sparsity is called *block-diagonal.*

these multiplies will take $O(m)$ time because the intermediate result $\mathbf{J}^\mathbf{T}\mathbf{x}$ is a vector of length $m$, and $m > n$. However, if $m \gg n$, the vector will be sparse, so sparse vector techniques reduce things back to $O(n)$. Using a solver that requires only $O(n)$ of these matrix vector multiplies means that the linear system can be solved in $O(n^2)$ time. Empirical results using a Conjugate-Gradient solver confirming this are discussed in Appendix B.

### 4.2.1 Complexity of the Metric

The above discussion ignored the metric. $\mathbf{M}$ is an $m \times m$ matrix, so in the general case, simply filling it or inverting it would dominate the asymptotic complexity of the differential solving. This complexity prohibits using arbitrary metrics or using the optimization objective to find the soft controls as described in Section 3.5.1. With some reasonable restrictions, metrics can be supported without adversely effecting the computational complexity. To the restrictions of the previous section, we add

4. Each object defines its metric (e.g. its contributions to $\mathbf{g}$) independently.

All of the arguments for evaluating the controls and their Jacobian apply to $\mathbf{g}$ as well. However, since each row of $\mathbf{G}$ can only depend on the variables of one object, we know that the matrix must have a block structure, depicted in Figure 4.1. When this is multiplied by its transpose to form the metric, the matrix will be block diagonal with a block for each object, and block sizes equal to the number of variables that each object has.

The block sparsity of the metric is important. It has only $O(m)$ entries, and can be inverted in $O(m)$ time because each of the blocks is independent. Using the same associativity argument as for $\mathbf{J}\mathbf{J}^\mathbf{T}\mathbf{x}$, the matrix multiplication $\mathbf{J}\mathbf{W}\mathbf{J}^\mathbf{T}\mathbf{x}$ also will take $O(n^2)$ time. The same arguments apply for the diagonal metric.

## 4.3 Solving the Linear System

To compute the Lagrange multipliers, we must solve a linear system which can be written in the standard form

$$\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}, \tag{4.1}$$

where **A** is a square matrix of size equal to the number of constraints, and is determined by the Jacobians of the control functions, and **b** is a vector computed from the control values. For example, in the simplest form of the differential optimization, **A** is $\mathbf{J}\mathbf{J}^T$, and **b** is $\dot{\mathbf{p}}$.

In choosing a numerical algorithm to solve the linear system, we first must consider the properties of **A**. First, it will always be positive semi-definite, and in the cases with damping, positive definite. It will also always be symmetric. These properties hold because it is created by multiplying a matrix by its transpose and altering the diagonal.

The most important property of **A** with regard to solving it efficiently is that it will be sparse. Or, more precisely, it will be created by multiplying a set of sparse matrices. This is significant because in this section we will show an algorithm which does not actually ever build **A**. The structure of **A** relates to the constraint problem at hand. In particular, an element of **A** will be non-zero if the two constraints (one corresponding to the row, one corresponding to the column) share a variable.

Solving a linear system is an $O(n^3)$ process in general. Exploiting the sparsity of the matrices is important to achieving better performance. Sparse matrix techniques generally fall into two categories: direct and iterative methods. Direct methods take advantage of the structure of the matrix problem to solve the linear system as quickly as possible. For matrices where the structure is unknown, the algorithms do a pre-analysis to find the structure of the matrix so it can be solved quickly. Direct methods are not well suited to the purposes of this thesis for several reasons. First, because the structure of the matrix is continually changing, the pre-analysis must be done often making its cost difficult to amortize. Secondly, direct methods' computational complexity is proportional to the bandwidth of the matrix, so it is possible that even for an extremely sparse matrix, the cost will still be $O(n^3)$. Finally, direct methods are complicated to implement.

Iterative methods solve linear systems by repeatedly performing a calculation that eventually converges on the solution. Such methods offer an opportunity to trade accuracy for performance by controlling the tolerance to which the solver is required to achieve. By setting a larger tolerance, the algorithm is permitted to stop before it achieves an exact solution. In Section 4.2, an argument was given that with an iterative solver that does only a constant number of matrix vector multiplies per iteration, $O(n^2)$ performance could be achieved for the differential approach. The particular type of algorithm suggested for use in the differential approach, Conjugate-Gradient methods, offers this performance and several other advantages.

### 4.3.1 Conjugate-Gradient Linear System Solving

Conjugate-Gradient is a class of iterative algorithms for solving linear systems, non-linear systems, and optimization problems. Surveys of Conjugate-Gradient methods for solving linear systems are provided in [PS82] and [She94]. [GL89] also provides a good introduction to the techniques. The actual solver I have used is adapted from the one presented in [PFTV86]. We briefly review some of the important attributes of the algorithm here.

Conjugate-Gradient algorithms operate by repeatedly refining an estimate to the solution of the system of equations. Consider the current estimate as a point in $n$-dimensional space. At each iteration, the algorithm chooses a direction in which to move the estimate, computes a distance to travel in this direction, and finally updates the estimate accordingly. This process is repeated until the estimate is sufficiently close to being a solution, which can be quickly checked by inserting the estimate into the equation and measuring the error.

The key piece of a Conjugate-Gradient algorithm is how it selects directions to move its estimate in. For each iteration, a direction is chosen that is conjugate (orthogonal) to the preceding directions. Since a set of mutually conjugate vectors in $n$-space has $n$ elements, a conjugate gradient algorithm, under ideal situations, would require at most $n$ iterations to get an exact solution. In practice, numerical inaccuracies may cause the solver to require more iterations on ill-conditioned problems. Because we are not as concerned with accuracy, we will settle for stopping the solver before it completely converges in ill-conditioned cases, limiting it to $O(n)$ iterations.

At each iteration, a conjugate gradient algorithm must compute a new direction, find a step length in this direction, revise the estimate, and compute the error residual. The only part of this which actually must access the matrix are the first and last step. What is most significant for our purposes here is that in those steps, the only accesses to the matrix are to multiply it by a vector.

The Conjugate-Gradient technique leads to a family of algorithms. Many of the more sophisticated algorithms, such as LSQR algorithm introduced in [PS82], provide greater precision and more tolerance of numerical errors. As discussed in the paper introducing LSQR, the more sophisticated algorithms offer advantages only when high accuracy is required and when the matrix is ill-conditioned. However, as discussed in Section 4.1, the standard tradeoffs of numerical analysis do not apply to our applications. Since getting an answer quickly is more important than obtaining a high-accuracy answer, the more traditional Conjugate-Gradient methods may be actually be more desirable for our purposes.

### 4.3.2 Selection of a Linear System Solving Algorithm

Solving the linear system dominates the computational complexity of the differential approach. Selecting the algorithm is, therefore, an important decision.

From my experience, a Conjugate-Gradient solver is the best candidate for use in the differential approach because:

1. It exploits sparsity irrespective of the form of the matrix to be solved, leading to $O(n^2)$ performance in typical applications that meet the assumptions of Section 4.2.

2. It is simpler to implement than other general sparse matrix solvers such as direct methods.

3. It does not need to form the actual matrix that defines the linear system, which may not be sparse. Instead, it simply uses the Jacobian matrices that make it up, avoiding multiplying the Jacobian by its transpose.

4. The only operations it requires from the Jacobian is the ability to multiply by it and its transpose by a vector, providing freedom in choosing the representation for $\mathbf{J}$.

5. The stopping criteria can be adjusted to trade accuracy for performance by accepting solutions within a larger tolerance.

Most other approaches to solving the linear system in the differential optimization problem fail to provide one of these advantages. Other solvers potentially offer other advantages, for example lower overhead, more accuracy, or better tolerance of numerical errors, however, these advantages are often outweighed by those listed for Conjugate-Gradient. For example, a Cholesky factorization, as presented in [PFTV86], is a very efficient way to solve linear systems with positive-definite symmetric matrices, just what is needed for the differential approach. Such a solver has $O(n^3)$ performance, but with a very small constant, and is numerically stable. For small problems, the small constants of the Cholesky algorithm might make it a faster method. However, even in these cases, performing the matrix multiply $\mathbf{J}\mathbf{J}^{\mathbf{T}}$ is often expensive enough to outweigh the performance gains in the linear system solver.

Other iterative solvers may compete with Conjugate-Gradient in some applications. The performance of iterative solvers is very problem dependent. My experimentation shows Conjugate-Gradient to be vastly superior than simpler Jacobi iterative solvers. Implementing Gauss-Seidel iteration or Successive Over-Relaxation (SOR) is difficult with the matrix representations used in my implementation (discussed in the next chapter), as column operations cannot be implemented efficiently.

If the matrices to be solved have a known sparsity pattern for which an efficient, special purpose solver exists, such a solver would probably be preferable to using conjugate-gradient. For example, if the matrix is known to be banded with a narrow bandwidth, linear time algorithms can be used. However, selective use of special purpose solvers has not been explored in this thesis as I have tried to emphasize general purpose techniques.

### 4.3.3 Partitioning the matrix

One very important type of sparsity that will often be useful to exploit in the differential approach is partitioning. In some cases, the rows of the linear system may not all depend on one another, that is, they may be partitioned into separate, smaller pieces, similar to those shown in Figure 4.1. With the differential approach, a partionable matrix occurs whenever there are groups of objects that do not share any controls. Partitioning breaks the large matrix into smaller pieces when they are independent.

The reasons for partitioning the matrix include:

1. Solving several smaller problems will be faster than a single large one if the complexity is greater than linear.

2. If one of the partitions is ill-conditioned, it can have bad effects on the other partitions.

3. Some of the partitions may be trivial to solve. This is especially true in cases like constraint-based drawing where one partition will be receiving user input, and the other partitions will be sitting idle.

Reason 1 is not as important with the conjugate-gradient method, described in Section 4.3.1. In a sense, the Conjugate-Gradient algorithm solves the disconnected partitions in parallel. However, the solver must take the number of iterations required to solve the largest block. The savings is, therefore, not as great as when a straight $O(n^2)$ or $O(n^3)$ solver is used. However, the savings can be considerable when one block requires many iterations, for example if it is large or ill-conditioned, and many other blocks can be solved quickly, either by trivial checks or because they are small.

Reason 2 is particularly important with the conjugate gradient methods described in Section 4.3.1. If any partition of the matrix is ill-conditioned, the steps the solver will take will involve very small direction vectors and very large scaling factors. The parts of the direction vector that correspond to the well-determined partitions of the matrix will contain extremely small numbers, so the large steps should not have any effect. However, because of floating point inaccuracy in representing the very large and very small numbers, much error is introduced. The net effect of this in differential manipulation is that if there are any controls which are ill-conditioned they will cause completely disconnected graphical objects to jiggle.

#### An algorithm for partitioning

One of the features of partitioning is that it is simple and fast to implement. To partition $\mathbf{J}\mathbf{J}^\mathbf{T}$, it is sufficient to order $\mathbf{J}$.

We begin with each variable in a disjoint set. For each constraint, we union the disjoint sets that correspond to each variable that the constraint affects. When completed, we can then gather each set together into the state vector.

The key piece to performing the partitioning is that we can do the disjoint set union and find operations very quickly. In fact, using an extremely simple algorithm, the unions and finds can be performed in nearly linear time[2]. The disjoint set union and find algorithms, along with a complexity analysis, are provided in [Cor89].

The partitioning algorithm runs in time linear with the number of variables and the number of non-zero elements of the matrix. Since the algorithm must actually have the matrix to partition, and filling the matrix takes time proportional to the number of non-zero elements in it, computing $\mathbf{J}$ is often the most expensive step in partitioning.

## 4.4 Reducing Problem Size

As described in Section 4.2, solving the linear system in the differential optimization is the dominant factor in the computational complexity of the differential approach. Without placing restrictions on the problems, it is unlikely that we can achieve better than $O(n^2)$ complexity for general constraint problems. We must find ways to keep $n$ small, without restricting the size of the problems that the user actually works on. That is, to find methods which give the user the illusion that the system is working on a larger problem, while in fact, the problem's size has been reduced.

Partitioning, described in the previous section, is an example of a method for transparently reducing the problem size by solving a set of smaller problems rather than the larger problem.

If we know how some variables are going to change by some other means, the expense of solving the differential problem is not required. For example, If we know that an object is frozen in place, we know that it will not move, and that the time derivatives of its variables are simply zero. We can implement the constraints by removing the object's variables from the set of variables solved for, rather than by adding more equations.

For complexity purposes, $m$ and $n$ are really the number of *active* variables and constraints, that is the number that might actually have an affect on the current step. We can discount objects if there are no controls that may cause them to move or if there is something else which requires that they do not change. We can forget a control if it does not effect any changeable objects. We call the set of variables and constraints that are actually participating the *working set.*[3]

In general, adding a new control or constraint adds to $n$ and therefore makes the differential optimization more expensive to compute. However, constraints realized by removing variables from the working set instead reduce $m$ rather than increasing $n$, speeding computations. Such constraints are represented implicitly in the structure of

[2] Actually, it is inverse Ackerman worse than linear, but since the inverse Ackerman is a small constant ($< 5$) for any quantity we are likely to encounter, we can consider it to be linear.

[3] The other obvious term, active set, already means something else.

the problem, rather than explicitly by an equation.

Freezing an object is a simple example of a constraint that can be implemented implicitly. Specifying that an object is to be frozen, e.g. that it must not change, could be represented by explicitly placing a control on each variable. However, the effects of these controls are known: they will cause the variables not to change. Since the variables will not change, they can be removed from the working set. Without variables in the working set, the object is constrained not to move, however, this constraint is represented implicitly in the structure of the problem.

Implicit constraints generalize to sets of individual variables. For example, a line segment has four degrees of freedom. Freezing its length can be implemented by explicitly placing a controller on a length connector. However, if the line segment's representation included a separate independent variable for length, this variable could be removed from the working set to implicitly represent the length constraint. This was how the fixed length line segment of Section 3.7 was created.

Implicit constraints are representation dependent. In the example above, if the programmer had chosen a different representation for the line segment, for example to represent it by the positions of its endpoints, the length constraint could not be implemented as an implicit constraint.

Often, it is worthwhile to choose representations to maximize the number of implicit constraints. For example, in a planar mechanisms simulator like the one described in Section 9.2 most line segments represent rigid linkage rods. Therefore, a representation is used that has length as a variable. This way the commonly needed constraint that the line is a rigid length can be realized as an implicit constraint.

Finding new representations of objects is a difficult problem, especially when we cannot anticipate the types of constraints and combinations that will be desired. In fact, the whole differential approach is a response to the problem that representations cannot simply be derived on demand. In terms of physical simulation methods, finding new representations is equivalent to deriving new equations of motion with Lagrangian dynamics techniques.

Finding new parameterizations is equivalent to symbolically solving the non-linear systems of equations. This task is not automatable for any general class of problems. Although it is not possible to create new representations either dynamically or in a general, automatic way, it is possible to create multiple representations for important cases of controls on objects.

It is conceivable to build a system that changes the representation of objects to maximize the number of implicit constraints. This requires solving a combinatorial optimization problem. Globally optimizing for the maximum number of constraints is most likely difficult.[4] Incremental methods might provide different results depending on the order the controls are added, which may be a problem since the behavior of implicit constraints and standard controls differ.

[4] I believe it to be NP-hard, although I do not have a proof.

Another form of implicit constraint is merging, that is having multiple parameters access the same variable, as seen in ThingLab [Bor81]. Merging is an implicit constraint for equating parameters. Because merged parameters share a single variable, they have exactly the same value. A more general variant of merging views a variable as an input. It can either be connected to a slot in the state vector, or connected to some output. This effectively mixes local propagation into the numerical methods.

Implicit constraints are exact. If an object is frozen, it stays exactly fixed. Two merged quantities are exactly equal. While this exact equality can be an asset, it can also be a problem as it means the constraint behaves differently than its explicit counterpart. This can be particularly troublesome in cases where the solver might break the equality constraint slightly, for example to achieve a least squares solution to an overconstrained problem. This distinction becomes significant when the system switches between the two types of constraints.

## 4.5 Trading Accuracy for Performance

Trading accuracy for performance is an important method for improving the performance of the differential approach.

Using simple ODE solvers with fixed step sizes is one way to trade accuracy for performance. The simpler ODE solvers compute rough solutions quickly, and then permit feedback terms to clean up the results in subsequent steps. This is useful because it gives a rough answer quickly, but provides a more accurate answer over time. For example, in dragging an object, the accuracy needed might be low. When the user stops to examine a situation more closely, the solution has a moment to become more accurate, and by the time the user has decided that a solution is acceptable enough to print or render at high resolution, the constraints are fully converged.

Varying the step size of the ODE solver is another way to trade accuracy for performance. Larger step sizes cause larger apparent velocities on the screen, assuming that control velocities are constant. As discussed in Section 3.3, larger step sizes may be less accurate.

The use of a Conjugate-Gradient linear system solver provides another way to trade accuracy for performance. By using a larger tolerance for the stopping criterion, the solver can accept an answer more quickly if it finds an approximate one.

Many aspects of the methods used to handle inequalities effectively trade accuracy for performance as well. For example, the simple scheme for selecting active sets of Section 6.4.4 trades accurate solutions for faster solving. Not backing up the ODE solver when an inequality boundary is crossed, as will be discussed in Section 6.4.5, may also be viewed as another method to achieve performance by giving up accuracy.