Motion Editing: Mathematical Foundations

F. Sebastian Grassia

Carnegie Mellon University

Before we can begin tackling motion editing problems in earnest, we must ensure our familiarity with the basic tools we will constantly be using, and with the objects on which the tools will be applied. Our goal is the manipulation of motions performed by 3D computer characters. Most, if not all, of the high-level techniques for editing motions that will be presented later in this course depend heavily on the representations we choose for our character models, their underlying degrees of freedom (DOFs), and the motion curves that define an animation. In this section we will examine our choices for these representations, and also introduce an important algorithm for motion editing, inverse kinematics (IK).

As this is an advanced course, we assume the reader has a basic familiarity with transformation hierarchies (and stacks), parameterized cubic curves such as hermites and b-splines, and has some exposure to differential equations. Rather than present a detailed tutorial from the bottom up on these topics, we will focus on aspects of representation and inverse kinematics that are not normally discussed in research papers or texts - issues and problems that often arise when implementing motion editing tools, and observations and solutions culled from various of this course s presenters. As much as possible, however, we will provide references for further study in each topic we discuss.

1. Representing Characters

Our first fundamental choice is how we will represent animated characters in our algorithms. Ultimately, we desire our character models to be very expressive and animatable in subtle ways, from the ripple of a flexing bicep to the scrunching of a cheek when the character winks. However, in this course we are primarily concerned with the broader motions of the body, those that can be captured by today s motion capture systems. We will see that these motions can be represented fairly simply and compactly; motions of muscles can often be derived automatically from the broader body motions, but their representation and derivation, as well as that of facial animation, deserve courses of their own to study.

1.1. Structural Representation

Assuming that we are talking about animating humanoid-like characters[1], the broad motion we are considering maps almost directly to the motion of the underlying skeleton. Therefore a good representation of the skeleton will suffice to represent the motions with which we are concerning ourselves. The relevant characteristics of a human skeleton are that it is comprised of rigid links that are connected to each other by rotational joints that do not translate with respect to each other (to a very good approximation). Thus the entire skeleton assembly has an overall position and orientation, and many connected, rigid parts that pivot, twist, and swing with respect to each other.

We could choose to model a skeleton very physically, starting with a collection of bones represented as rigid 3D models each of which has a position and orientation, and attaching them together with explicit constraints at the joints that would prevent them from separating or from rotating in illegal ways (for instance, a wrist cannot twist). With this model, the DOFs of the character are the positions and orientations of each bone in the body. This is awkward because anytime we move any bone (through even the simplest of control techniques), we must reapply and solve (potentially) all of the constraints that hold the bones together.

A more convenient representation imposes a hierarchical relationship on the bones, in which the position and orientation of each bone is specified relative to a "parent" bone. Since we have stipulated that bones do not slide with respect to each other at the joints, a bone's configuration at any time can be specified by a fixed translation (thus not involving any DOFs) and a rotation relative to the parent bone (which is in turn defined in terms of *its* parent, all the way up to the root, which has full translational and rotational DOFs). In effect we have eliminated the need for constraints to hold the bones together by implicitly removing the translational DOFs from all bones in the hierarchy save the root. This behavior is exactly what we get from a *transformation hierarchy*, such as that implemented in OpenGL or Inventor.

Our primary uses for hierarchies are to compute positions and orientations of points on the body (for example, sensor positions in a motion capture setup), and to display some version of the character[2], which itself performs mainly "compute current position of point on body" operations on the hierarchy. Let us say we wish to compute the current position of the center of a character's hand in a 2D stick-figure hierarchy (all math and procedures extend straightforwardly to 3D). Define $\mathbf{v}$ as the position of the center of the hand with respect to the wrist joint, which is the center of rotation for the hand. We compute the current, world space coordinates $\mathbf{v}_w$ of the hand by computing and concatenating transformation matrices at each joint between the point's origin bodypart (here, the hand) and the root of the hierarchy, which is often set to be the hips.

$$\mathbf{v}_w = \mathbf{T}_w(x,y)\mathbf{R}_w(\theta_w)\,\mathbf{T}_s\mathbf{R}_s(\theta_s)\,\mathbf{T}_c\mathbf{R}_c(\theta_c)\,\mathbf{T}_b\mathbf{R}_b(\theta_b)\,\mathbf{T}_f\mathbf{R}_f(\theta_f)\,\mathbf{T}_h\mathbf{R}_h(\theta_h)\,\mathbf{v}$$

The above equation is the formula we use to compute the world space position of the hand, and represents a matrix-vector multiplication, where the matrix is a product (composition) of many matrices. The sequence of **T**ranslations and **R**otations beginning on the right and moving to the left are the fixed **t**ranslation that positions the **h**and's base at the origin and **r**otates by θ_h, then positions and rotates the **f**orearm, then on to the **b**icep, **c**hest, **s**tomach, and finally, the global rotation and translation at the **w**aist. This computation can be performed rapidly for many different points on the character's body by recursively traversing the hierarchy using a transformation stack like the one provided by OpenGL. For greater detail on the mathematics of transformation hierarchies, see Foley[2].

We generally choose to represent skeletons with transformation hierarchies because of the following advantages:

1. Hierarchies are compact. The number of control parameters at each bone/joint is proportional to the number of rotational DOFs of the joint.

---

[1] For a beginning discussion on retargeting human motion to non-humanoid characters, see Gleicher[6]

[2] We need not model or draw actual bones. The hierarchy itself gives us the requisite rigid-limb behavior, and we are free to draw any geometry to represent the limbs, and even, as previously mentioned, layer more sophisticated "secondary" structures on top of the hierarchy, such as muscles whose shapes are driven by the joint angles.

2. Constraints are implicitized. Changing any control parameter in a hierarchy produces a valid configuration of the skeleton, *i.e.* bone attachment constraints are never violated[3].
3. We have efficient traversal algorithms. Most 3D graphics APIs and some hardware support fast display algorithms for hierarchies. There are also many fast (linear time) recursive algorithms for performing dynamics and computing derivatives for hierarchies (discussed further in section 3)

1.2. Mathematical – Rotations 101

Deciding on skeletons and hierarchies for the structure of our characters tells us that to generate a pose for the character, we must specify a global translation and a set of values for all of the joint angles. But we have said nothing about how we compute the rotations of the joints from their control-parameters (the joint angles). In the 2D example used in the last section, joint angles are simple because there is only one way to rotate: about the axis that sticks up out of the page, and forming linear transformation matrices that accomplish these rotations is straightforward.

In 3D, however, joints can have one, two, or three degrees of rotational freedom, *and* we may wish to place limits on the amount of allowable rotation for single DOFs or groups of DOFs. Complicating the problem further is that there are many ways of parameterizing 3D rotations, and unfortunately, no single parameterization is the clear choice for all applications. This is because rotations are inherently non-Euclidean (a property of which is that travelling infinitely far in a rotation parameter-space brings you back to the same place an infinite number of times, just as travelling in a straight line on the globe brings you back to your origin), while our mathematical tools for working with transformations, differential equations, and motions all assume Euclidean parameterizations of the quantities they manipulate.

The consequences of this situation are discussed in some detail in the included reprint[4], *Practical Parameterization of Rotations Using the Exponential Map*[7], which should appear on the CDROM as **expmap.pdf**. The paper examines the problems caused by the non-Euclidean nature of rotations, describes how commonly used parameterizations such as Euler angles and quaternions address those problems, and introduces a robust formulation of the exponential map, a parameterization not well known to the graphics community, but which performs better than the more common parameterizations for a range of graphics applications. It includes the strengths and limitations of all parameterizations considered with respect to applications such as forward and inverse kinematics, simulation, interpolation, and spacetime optimization. From this point on in these notes, we assume the reader is familiar with the material presented in the paper.

At this point, we know enough to construct hierarchies to represent characters, and to choose representations for joints that are suitable for posing and positioning the characters using various control algorithms. However, motion editing is concerned not just with altering poses, but with manipulating entire animations. Therefore we must discuss representations of motions, which will introduce new constraints on the choices we have already discussed.

2. Representing Motions

The end goal of 3D animation is the creation of a sequence of rapidly shifting 2D (at this point in time!) images to create the illusion of motion. This representation is not convenient for editing the animation in significant ways, however, for several reasons, most prominently of which is that all the 3D information has been lost, and that the representation is not very compact. We require of potential representations for character motion that they facilitate the kinds of motion editing operations in which we are interested, and that they are as compact as possible. The operations we would like to be able to perform on motions, all of which will be discussed in much more depth later in the course, include the following:

---

[3] We are ignoring the possibility of violating joint limits, for the moment. It is, however, fairly straightforward to incorporate rotational joint limits into a hierarchy so that, for example, a knee will not bend backwards.

[4] Courtesy of A K Peters, Ltd., publisher of jgt, Journal of Graphics Tools

- Displacement mapping. Also known as *Motion Warping*[17], this technique deforms an animation by adding a low frequency displacement (calculated to satisfy newly introduced pose constraints at a number of key-times) onto the existing animation.
- Various signal processing algorithms. Many different types of signal processing algorithms have been applied to motions to produce new, related motions[1][14]. All assume that motion can be decomposed into continuous, one-dimensional signals over time.
- Spacetime optimizations. Parts or all of the spacetime optimization algorithm[16] have been used to warp animations while preserving kinematic constraints[5], and additionally preserve dynamics properties of the original motion[10]. Like Motion Warping, these techniques produce a displacement to be added to the original motion, but in a far more sophisticated and powerful manner. They additionally require that we be able to compute derivatives through the animation representation.
- Blending. The simplest means of smoothly transitioning from one motion into another, blending interpolates between two motions, generally changing the interpolation parameter over time according to some sigmoid-like function[9]. Blending two complete motions (rather than just overlapping head & tail segments) can also be used to produce a new motion partway between the two. Rose *et al* have extended this to multidimensional interpolation between many motions[12].

The most compact means we have of representing a character pose is the set of joint angles that determine its configuration (including the global translation and orientation), which are often collected into a single vector called the *state vector*. This being the case, we can think of motions most generally as functions of time that return poses in the form of a state vector. For many of the operations above, it is, however, more useful to carve up an animation into a collection of functions of time that each return the value of a single DOF or joint in the character. A commonly used class of such functions is *motion curves*. In the remainder of this section we will evaluate how each of the rotational parameterizations discussed in the last section perform under the above operations, and how our motion curves may differ depending on their source.

2.1. Euler Angles

Euler angle representations of rotations fit easily into the above motion editing operations. Because Euler angles are Euclidean quantities, we can construct time varying motion curves using simple Euclidean cubic curves, such as hermites or natural splines for interpolating a dataset, or b-splines for modeling very smooth motion. However, the singularities in Euler angle parameterizations can cause mischief, especially when applied to motions acquired from motion capture. The problem is as follows: near singularities, one or more of the Euler angles parameterizing a three DOF joint can vary over a wide range without producing any corresponding change in rotation. This can easily happen when inferring Euler angle motions from a sequence of rotation matrices, which is often done in the process of applying sensor data to 3D characters. Motion editing operations such as displacement mapping, blending, or signal processing can take a motion segment that straddles a singularity in Euler angle space and move it or amplify it *away* from the singularity. If this happens, the wide variations in the previously unimportant DOFs suddenly become very visible, resulting in artifacts. Most of the time, careful pre-filtering of Euler angle motion curves can prevent this occurrence.

2.2. Quaternions

Because the unit quaternion parameterization of rotations is neither scalar-valued nor Euclidean, we must use different mathematical tools to apply the above motion editing operations. First of all, we cannot use Euclidean interpolants to generate unit quaternion motion curves. We do have at least two widely known alternatives. First are the *spherical Bezier curves* derived by Shoemake[13], which are intuitively similar to the de Castlejau construction of Euclidean Beziers. Spherical Beziers are easy to implement, but do not possess a closed form, nor do they provide C^2 continuity. A more robust (and involved) solution is the closed form quaternion curves of Kim *et al*[8], which parallel closed form Euclidean hermites, Beziers and b-splines. We can use such representations not only to construct motion curves for the animation itself, but also to construct displacement curves for displacement mapping. This is, perhaps, an advantage, since it seems to make more sense to manipulate 3D rotations in S^3 (which, recall, corresponds very closely to SO(3)) than in a collection of uncoupled scalar spaces, as one would with Euler angles. This is an area ripe for some good empirical measurement.

Secondly, the addition operator, by which we add signals and motions together (as in displacement mapping, blending, and other forms of signal processing), changes for quaternions. Addition becomes quaternion multiplication, which, we should note, is not commutative. Also, whereas multiple Euclidean quantities can be easily interpolated and averaged using standard statistical tools, we must use more sophisticated mathematics to combine multiple quaternions intelligently, such as the radial basis functions described by Rose *et al*[12].

Spacetime optimization methods present the greatest challenge to all potential representations. This is because, in addition to requiring the ability to calculate derivatives of positions and orientations on characters with respect to state at a particular instant in time (which even inverse kinematics and dynamics require), spacetime methods require that we be able to calculate these derivatives with respect to the animation itself. That is, the DOFs for spacetime are the motion curves themselves! This causes real problems for quaternions, because neither of our representations for quaternion motion curves permit differentiation with respect to the quaternions that define the curves. At this point in time, about the best we can do is fall back to constructing quaternion motion curves by independently interpolating each of the four quaternion components independently using Euclidean interpolants, and a) treat the values of the quaternion motion curves as non-unit quaternions, *i.e.* include normalization into the formulas for computing rotations from the motion curves; b) impose constraints or penalties at a reasonable number of points in time to keep the quaternions unit or near-unit length. This is essentially the scheme utilized by Popović[11] with good results.

2.3. The Exponential Map

The included reprint has already described the potential problems in using motion curves based on the exponential map to spacetime. As mentioned there, since most motion editing problems would not attempt to compute a rotational displacement of more than a complete revolution, the singularity in the exponential map may not be a problem. The larger issue is the performance of the exponential map as an interpolant when the key values differ significantly from each other in either S^3 or SO(3). More work needs to be done characterizing the error bounds (deviation from a comparable geodesic in S^3 over a wide range of endpoints, for example). Since displacement mapping itself is limited in its usefulness to reasonably small displacements, the exponential map may, in fact, prove perfectly functional.

Use of the exponential map brings all of the benefits of a Euclidean parameterization: addition is really addition (subject to the caveat raised by Grassia[7]); it is compact; we can use Euclidean interpolants. In fact, we have found it to be excellently suited to filtering and resampling rotational data acquired from motion capture, as the data is typically closely spaced.

2.4. Discrete vs. Continuous Motion Curves

For the purposes of constructing and utilizing motion editing software we can abstract a motion curve as a continuously differentiable function of time that returns rotations or angles of some form. The actual structure of a motion curve, as well as the means by which we compute derivatives, *etc.*, depends on the source of the motion. Hand-generated 3D animation typically consists of interpolated keyframed poses; thus the motion curves are cubic curves of some flavor, with potentially widely-spaced keys. Such functions generally have closed form solutions to computing the value, parametric derivative, and derivatives with respect to the DOFs (*i.e.* the keys). Motion captured animation, on the other hand, is, in its original form, a discrete set of closely spaced (in time) samples. Used in this form, we generally perform some simple interpolation such as linear or parabolic to sample the "curve" between existing samples, and use finite differences to calculate derivatives. It is always possible to fit a cubic or other interpolating curve through the original data, however. Data reduction is possible by using fewer keys than datapoints and optimizing the positions of the keys to a "least mean squared error" or other metric; in doing so we run the risk of losing some of the high frequency content of the original data. In practice, a reduction by a factor of two or three can generally be achieved without noticeable degradation.

3. Controlling Characters with Inverse Kinematics

Controlling characters through motion editing is the end-goal of this course. As we will see, however, our motion editing techniques rely heavily on lower-level control techniques, most notably forward and inverse kinematics. We assume that the reader is well versed in forward kinematics, which moves a character by directly specifying values for all of its joint angles. Inverse kinematics (IK), which allows an animator to move position or orientation "handles"

attached to points on a character's body and lets the computer figure out how to set the joint angles to achieve the goal, may be familiar in use, but possibly not in function. Therefore, after outlining the uses of IK and related techniques in motion editing, we will provide an introduction to the various types of IK algorithms. Time and space constraints prevent us from providing the reader enough detail to code an IK algorithm directly from these notes; rather our goal is to enable enough understanding to choose intelligently from different approaches when faced with a particular motion problem. We refer the reader interested in a more detailed presentation to the work by Welman[15], which is both broad and fairly deep in its coverage.

3.1. IK Application and Problem Definition

Inverse kinematics has become a widely used tool for character positioning in 3D environments and animation packages. Simple implementations consider each limb of a character a separate, "IK-enabled" object whose end-point, or *end-effector*, can be moved about through direct manipulation. More sophisticated implementations allow multiple, coupled, handles placed at arbitrary points on the body, and provide solutions that position the entire character body so as to best satisfy all of the constraints generated by the handles. For instance, if the character were in a squatting position with its toes fixed to the ground by position handles placed there, and the animator continuously pulled upwards on the character's hand, the character would rise to a standing position, and eventually stand on tip-toes if the hand were pulled high enough.

IK has other uses than primary, keyframe animation. IK is the prevalent technique for mapping motion capture sensor data onto 3D characters. Techniques related in structure to IK include inverse dynamics (solving for muscle accelerations given forces), and much of the core engine of spacetime algorithms. Except for extremely simple systems, the IK problem is generally underspecified. That is, we may be specifying a small number of constraints, such as "move the hand to position X while keeping the foot at position Y," which gives a total of six constraints (three position constraints for each handle), while we need to solve for many DOFs (upwards of fifty for a moderately detailed humanoid). Thus there are many possible settings of joint angles that will satisfy the constraints, and it is up to the IK algorithm to pick the "best" one. Therefore IK algorithms typically have some metric that allows them to rate potential solutions.

IK algorithms designed to control robotic manipulators are fairly well-defined, because such manipulators typically have few DOFs (so they are less underspecified), and metrics such as "low power consumption" or "low torque generation" make sense for evaluating solutions to their IK problems. Algorithms designed to control humanoid or other characters are typically much more complex, because characters have many more DOFs than a robotic arm, and the physically-based metrics such as those used in robotic algorithms tend to produce (not surprisingly) robotic looking motion. Metrics that effectively encapsulate "how a human moves" require great skill and art to design and implement. Part of the promise of motion editing is the potential of using examples of existing motions to define better metrics for IK algorithms.

There are two basic approaches to designing IK algorithms, but of course many variations on each.

3.1.1. Geometric IK Algorithms

Geometric algorithms have in common that they all solve directly for joint angles, given a handle and desired location/orientation for the handle. They do not include any information about physics into their calculations – they use only the geometry of the problem: relative position of the base of an IK chain to the handle goal-position, trigonometric relations, *etc.* Such algorithms generally use heuristics designed to generate perfectly reproducible results, and have the added benefit of being very fast. However, there is no published methodology for designing such systems; they tend to be proprietary.

A bigger drawback is that these algorithms do not generally scale to sets of multiple, interacting constraints. Furthermore, since they are geometric rather than differential in nature, they cannot be incorporated into spacetime or dynamics simulation systems.

3.1.2. Optimization/Differential IK Algorithms

The most general way of posing the IK problem is like so: our character is currently in some configuration $\mathbf{q}$, and we have imposed a set of constraints $\mathbf{C}$ on the character, which are functions of the IK position and orientation handles whose values are zero when the handles have achieved their goal values; consequently, generate a new pose $\mathbf{q}_n$ that

satisfies the constraints (if possible), and subject to that, tries to optimize some scalar valued function $E(\mathbf{q})$, which serves as the metric that guides the solution choice. Unfortunately, both the constraints and the function we are trying to optimize are non-linear, because the functions that generate rotations from joint angles are (extremely) non-linear. Such functions are the most difficult to optimize, and algorithms that perform direct, non-linear optimizations are intricate and difficult to tune[3].

We are not without alternatives, however. One common technique for solving non-linear optimization problems is to linearize them by moving from the initial state to a solution state *differentially*. Given the same starting state $\mathbf{q}$ and constraints $\mathbf{C}$, we no longer directly specify values for the $\mathbf{C}$ functions to take on, but rather a *direction* $\dot{\mathbf{C}}$ in which we would like them to move. To get the constraints to move, we now solve not for $\mathbf{q}$, but for a "state-space velocity" $\dot{\mathbf{q}}$, that is, an instantaneous velocity for each joint that will cause the character to move in a particular way. We solve for $\dot{\mathbf{q}}$ using optimization, and the function we use as our metric can be any quadratic function $G(\dot{\mathbf{q}})$, a family of functions that includes minimizing power consumption, among many others.

The good news is that $\dot{\mathbf{C}}$ and $\dot{\mathbf{q}}$ are related linearly, regardless of how non-linear the original $\mathbf{C}$ are. The actual relationship is $\dot{\mathbf{C}} = \partial\mathbf{C}/\partial\mathbf{q}\ \dot{\mathbf{q}}$; the derivative $\partial\mathbf{C}/\partial\mathbf{q}$ (computation of which is discussed by Welman[15]) is a matrix that maps velocities in state space into velocities of the constraints. The "bad" news is that we have not yet satisfied the constraints, only computed a direction for the character to move in that will cause the constraints to move in a certain direction. If, however, we choose $\dot{\mathbf{C}}$ so that the constraints move towards their goal states (this is trivial to do), then by moving the character in the direction of $\dot{\mathbf{q}}$ will, up to a point, bring the constraints closer to their goals. This suggests that we can eventually achieve the constraint goals by iterating the process of "taking steps," each of which brings us closer to the goal. To produce a new state using our computed $\dot{\mathbf{q}}$, we must solve the ordinary differential equation $\mathbf{q}_n = \mathbf{q} + \dot{\mathbf{q}}dt$, or more precisely, since we are integrating the differential equation through an iteration parameter "t", $\mathbf{q}(t+dt) = \mathbf{q}(t) + \dot{\mathbf{q}}(t)dt$. We continue this iterative procedure until either the constraints are satisfied (in optimization lingo, until the constraint *residual* is zero), or until it is clear the constraints cannot be satisfied (until the constraint residual stops decreasing). For a much more in-depth introduction to this process, see Gleicher's treatment[4]

There are several good reasons for going through all this trouble. First of all, this formulation is extremely general. Many different types of relations can be expressed as numerical constraints, including geometrical constraints such as keeping a point (*i.e.* position handle) in a plane or on a line, keeping a point a certain distance from another point or plane, making orientations or directions line up, keeping a point on one side of a plane, and making sure joint limits are not violated. Furthermore, there is no limit on the number or complexity of constraints that can be specified and solved for simultaneously. Secondly, this formulation applies its metrics (whatever we specify them to be) not to poses, but to differential *movements*, *i.e.* the computation of $\dot{\mathbf{q}}$. This means that our metrics talk about how the character *moves* in response to stimuli, *i.e.* IK goals. Finally, this formulation can be straightforwardly extended to the dynamics realm (also used by spacetime), where we solve for acceleration $\ddot{\mathbf{q}}$, and integrate through ODE's for both velocity and position.

Unlike geometric algorithms, which generate a goal state in a single step, differential IK methods must iterate a number of times to achieve their IK goals. Their performance, therefore, depends heavily on the convergence properties of the series of integrations they make. There are two main classes of optimization algorithms, distinguished by the amount of work they put into generating $\dot{\mathbf{q}}$, which has direct consequences on their convergence properties.

Constrained Optimization

Constrained optimization algorithms set out to solve the problem described above exactly. That is, they construct and solve a linear system that attempts to find a $\dot{\mathbf{q}}$ that exactly satisfies the relation $\dot{\mathbf{C}} = \partial\mathbf{C}/\partial\mathbf{q}\ \dot{\mathbf{q}}$ while simultaneously minimizing the metric function $G(\dot{\mathbf{q}})$. There are many different formulations (*eg.* Lagrange Multipliers, the Lagrangian), and many different numerical algorithms for solving the resulting linear systems (*eg.* conjugate gradients, LSQR, singular value decomposition); many formulations and solvers are discussed in our references for this section [3][4][15]. We will now list the features common to algorithms of this class.

- These algorithms converge to a solution that satisfies the constraints (or comes as close as possible) with quadratic convergence. In practice, this means that within a few iterations, we are very close to the solution, and when, in an interactive setting, constraints change suddenly, we can track them rapidly.

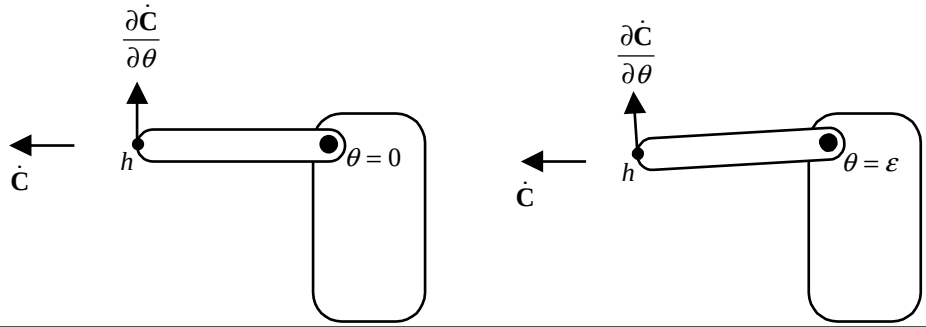

Figure 1. Here we see an arm attached to an unmovable body, free to swing about joint θ, and currently being pulled to the left by handle "h," resulting in a $\dot{\mathbf{C}}$ that points to the left. The configuration on the left is called a *singular configuration* because the derivative of the handle with respect to its joint has no component in the desired direction. The configuration on the right is a near-singular configuration, characterized by a vanishingly small, but non-zero derivative in the desired direction.

- The quadratic metric $G(\dot{\mathbf{q}})$ allows us to model physical behaviors, such as minimizing energy consumption, applying arbitrary forces on the character (for example, gravity), and even trying to remain as close as possible to some canonical pose subject to satisfying the constraints – this last is very useful for motion editing.
- Near singular configurations pose difficult problems for these methods. As demonstrated in Figure 1, when a character is in a near singular configuration, the derivative of the control handle with respect to all of the DOFs has only a very small component in the direction in which the constraint is pulling it. In this situation, we would like the solver to realize that it cannot really satisfy the constraint. Most solvers *do* recognize this for the perfectly singular configuration (where the derivative is zero). In the near singular case, however, most solvers believe that by assigning joint angle θ a very large velocity they *can* satisfy the constraint – instantaneously, this is true, but of course, it will not really cause the hand to move to the left. Instead, the whole arm will thrash about violently as the ODE integrator deals with the extremely high joint velocity generated by the optimizer. It is extremely difficult to design an IK constrained optimization algorithm that handles singular configurations gracefully and intuitively, and yet is still fast and rapidly convergent.

The complexity of solving linear systems and dealing with near-singular configurations have prompted many practitioners to opt for a simpler optimization algorithm.

Unconstrained Optimization

Rather than think of the IK constraints as a set of equations that *must* be satisfied, we can associate a specific cost to not satisfying a constraint – the greater the violation, the greater the cost. If we do so, it is possible to add the constraints directly into $G(\dot{\mathbf{q}})$, so that the violation of each constraint becomes an individual metric. $G(\dot{\mathbf{q}})$ becomes the sum of all the constraint metrics (plus whatever else might have been there before – power minimization, *etc.*), and we can now run an algorithm that simply optimizes $G(\dot{\mathbf{q}})$ by itself, since there are no longer any true constraints. This means we no longer need to solve any linear systems, so each iteration is less expensive and less complicated, and near-singular configurations are no longer much of a problem. This method of reformulating the problem as an unconstrained optimization goes by many names: it is well known as "the penalty method", Gleicher terms it "soft constraints"[9], Welman calls it the "Jacobian Transpose method"[9]. It has a physical analogy as well: its performance mimics what we would get if we attached a spring between each IK handle and its desired position. Accordingly, the characteristics of this method are the following:

- The penalty method has linear convergence, the constant on which depends on the stiffness of the springs. Although each iteration is cheaper, we typically need many more iterations to achieve our IK goals. Using stiffer springs (increasing the penalty weightings) can improve performance, but also taxes the ODE integrator.
- We no longer achieve the "optimize $G(\dot{\mathbf{q}})$ *subject* to the constraints" behavior because the original metrics in $G(\dot{\mathbf{q}})$ now compete with the constraint penalties for satisfaction. A consequence of this is that if we are trying to stay near some pose subject to some IK constraints, we will not actually be able to satisfy the constraints,

even if it is physically feasible. To be fair, we could iterate for awhile with all metrics in tact, and then run several iterations with only the constraint penalties influencing the optimizer.

- Near singular configurations are no longer a problem.
- Sudden changes in constraints can produce a "spongy" response since constraints that are satisfied produce a zero penalty (*i.e.* no spring force), so they must first become violated in response to a change in constraints before the solver kicks in and resatisfies them. By contrast, constrained optimization methods can maintain satisfied constraints exactly while constraints change.

To the extent of the presenters' knowledge, there is a dearth of freely available inverse kinematics implementations. Therefore, the choice of IK algorithm for a particular motion editing task will depend on several factors, probably chief of which is how much time can be invested in implementation. Penalty methods are much easier to get up and running. Constrained optimization is more general and can deliver substantially better performance, but is more involved and much trickier to make robust.

Bibliography

1. Armin Bruderlin and Lance Williams. *Motion Signal Processing*, Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 97-104 (August 1995). Addison Wesley. Edited by Robert Cook.
2. James D. Foley and Andries van Dam and Steven K. Feiner and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*, (1990). Addison-Wesley. pp 304-317.
3. Philip E. Gill and Walter Murray and Margaret H. Wright. *Practical Optimization*. 1981. Academic Press.
4. Michael Gleicher. *A Differential Approach to Graphical Interaction.* PhD Thesis. 1994. Carnegie Mellon University. pp. 39-65. Appears as Technical Report CMU-CS-94-217. ftp://reports.adm.cs.cmu.edu/usr/anon/1994/CMU-CS-94-217.ps.Z
5. Michael Gleicher and Peter Litwinowicz. *Constraint-based Motion Adaptation*, The Journal of Visualization and Computer Animation, 9(2), pp. 65-94 (1998).
6. Michael Gleicher. *Retargeting Motion to New Characters*, Proceedings of SIGGRAPH 98, Computer Graphics Proceedings, Annual Conference Series, pp. 33-42 (July 1998). Addison Wesley. Edited by Michael Cohen.
7. F. Sebastian Grassia. *Practical Parameterization of Rotations Using the Exponential Map.* to appear in jgt, Journal of Graphics Tools. Volume 3.3, 1998. A K Peters, Ltd.
8. Myoung-Jun Kim and Sung Yong Shin and Myung-Soo Kim. *A General Construction Scheme for Unit Quaternion Curves With Simple High Order Derivatives*, Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 369-376 (August 1995). Addison Wesley. Edited by Robert Cook.
9. Ken Perlin. *Real time responsive animation with personality*, IEEE Transactions on Visualization and Computer Graphics, 1 (1), pp. 5-15 (March 1995).
10. Zoran Popović. *Physically Based Animation*, to appear in Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, (August 1999). Addison Wesley. Edited by Alyn Rockwood.
11. Zoran Popović. *Motion Transformation Through Spacetime Optimization.* PhD Thesis. 1999. Carnegie Mellon University. Appears as Technical Report CMU-CS-99-106. Should appear in ftp://reports.adm.cs.cmu.edu/usr/anon/1999/CMU-CS-99-106.ps.Z
12. Charles Rose and Michael F. Cohen and Bobby Bodenheimer. Verbs and Adverbs: Multidimensional Motion Interpolation, IEEE Computer Graphics & Applications, 18(5), pp. 32-40 (September - October 1998).
13. Ken Shoemake. *Animating Rotations with Quaternion Curves*. Computer Graphics (SIGGRAPH '85 Proceedings), volume 19, pages 245-254 (July 1985). Addison Wesley. Edited by Brian A. Barsky.
14. Munetoshi Unuma and Ken Anjyo and Ryozo Takeuchi. Fourier Principles for Emotion-based Human Figure Animation, Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 91-96 (August 1995). Addison Wesley. Edited by Robert Cook.
15. Chris Welman. *Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulation*. Master's Thesis, Simon Frasier University, 1993. ftp://fas.sfu.ca/pub/cs/theses/1993/ChrisWelmanMSc.ps.gz
16. Andrew Witkin and Michael Kass. *Spacetime Constraints*, Computer Graphics (SIGGRAPH '88 Proceedings), 22 (4), pp. 159-168 (August 1988). Edited by John Dill.
17. Andrew Witkin and Zoran Popović. *Motion Warping*, Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 105-108 (August 1995). Addison Wesley. Edited by Robert Cook.