# Techniques for Interactive Manipulation of Graphical Protein Models

*TR92-016*

*April, 1992*

*Mark Christopher Surles*

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

# Techniques for Interactive Manipulation of Graphical Protein Models

by

Mark Christopher Surles

A Dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1992

Approved by:

Frederick P. Brooks Jr ______ Advisor

Jane S Richardson ______ Reader

Gary Bishop ______ Reader

**MARK CHRISTOPHER SURLES. Techniques for Interactive Manipulation of Graphical Protein Models (Under the direction of Frederick P. Brooks, Jr.)**

## ABSTRACT

This thesis describes a graphics modeling system, called *Sculpt,* that maintains physically-valid protein properties while a user interactively moves atoms in a protein model. *Sculpt* models strong properties such as bond lengths and angles with rigid constraints and models weak properties such as near-neighbor interactions with potential energies. *Sculpt* continually satisfies the constraints and maintains a local energy minimum throughout user interaction. On a Silicon Graphics 240-GTX, *Sculpt* maintains 1.5 updates per second on a molecular model with 355 atoms (1065 variables, 1027 constraints, and 3450 potential energies). Performance decreases *linearly* with increased molecule size. Three techniques yield interactive performance: a constrained minimization algorithm with linear complexity in problem size, coarse-grain parallelism, and variable reduction that replaces model segments with rigid bodies.

The thesis presents a Lagrange multiplier method that finds a constrained minimum and achieves linear computational complexity for articulated figures whose spine contains many more joints than any attached limb (e.g. reptiles, mammals, and proteins). The method computes the Jacobian matrix of the constraint functions, multiplies it by its transpose, and solves the resulting system of equations. A sort of the Jacobian at program initialization yields a constant, band-diagonal pattern of nonzeros. Multiplication and solution of band-diagonal matrices require computation that increases linearly with problem size. One or two iterations of this algorithm typically find a constrained minimum in this application.

The number of functions and variables can be reduced by the use of rigid bodies. A user can specify that a rigid object with few variables replace large segments of a model that should not change. For example, a user can twist a backbone into a helix and then freeze the helix by replacing its atoms and bonds with a cylinder of rigid shape but movable position and orientation.

Two improvements over existing interactive protein modeling systems have been observed in modeling sessions with *Sculpt*. First, time-consuming model correction is avoided by maintaining a physically-valid model throughout a modeling session. Second, additional cues about model properties can arise when a chemist interactively guides a folding simulation rather than viewing a cine loop from a pre-computed simulation.

## Acknowledgements

I thank my advisor, Professor Fred Brooks, for his advice and tutoring through graduate school. In addition to computer science, Professor Brooks greatly increased my critical thought and precision in technical prose. He taught strategies for conducting long-term research projects and for leading careers in scientific endeavors. I also thank Professor Brooks for generously funding this research through a molecular graphics research grant from the National Institutes of Health's National Center for Research Resources.

I thank Professors David and Jane Richardson of the Duke University Department of Biochemistry first telling me about their ideas for protein sculpting system. The applications in this thesis come from them. Both spent a significant amount of their sabbatical leaves teaching me about protein structures. I could not have attempted this thesis without their continued encouragements and ingenious insights.

I also greatly appreciate the moral encouragement given by my family and friends. My parents and brothers supported me from the beginning of school, through the gloomy days of searching for a suitable thesis topic, into the completion of this dissertation.. My newest family members, Sharon and Amanda, provided great joy and excitement during the final segment of my graduate tenure. To each of these people and many more not listed, thank you.

# Table Of Contents

# List Of Figures

# Notation

| | | |
|---|---|---|
| Vector notation: | $\mathbf{a}$ | column vector |
| | $a_i$ | $i^{th}$ element of vector $\mathbf{a}$ |
| | $\mathbf{a}^i$ | value at $i^{th}$ iteration |
| | $\mathbf{a}^T$ | transpose |
| | $\mathbf{a}^T\mathbf{z}$ | inner product; $\Sigma_i a_i z_i$ |
| | $\lVert\mathbf{a}\rVert$ | length of $\mathbf{a}$; $\sqrt{\mathbf{a}^T\mathbf{a}}$ |
| Differential operators: | $\dfrac{\partial f}{\partial a_i}$ | partial derivative of $f$ with respect to $a_i$ |
| | $\nabla f(\mathbf{a})$ | gradient vector; $\begin{bmatrix} \dfrac{\partial f}{\partial a_1} \\ \vdots \\ \dfrac{\partial f}{\partial a_n} \end{bmatrix}$ |
| | $\nabla^2 f(\mathbf{a})$ | Hessian matrix; $\begin{bmatrix} \dfrac{\partial^2 f}{\partial a_1 \partial a_1} & \cdots & \dfrac{\partial^2 f}{\partial a_1 \partial a_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial a_n \partial a_1} & \cdots & \dfrac{\partial^2 f}{\partial a_n \partial a_n} \end{bmatrix}$ |
| Reserved names: | $n$ | number of variables |
| | $m$ | number of constraints |
| | $\Re^k$ | k-dimensional space of real numbers |
| | $\mathbf{x}$ | variables in minimization problem; $\mathbf{x}\ \varepsilon\ \Re^n$ |
| | $\lambda$ | vector of Lagrange multipliers; $\lambda\ \varepsilon\ \Re^m$ |
| | $b$ | nonzero bandwidth of a square matrix |
| | $e(\mathbf{x})$ | energy function; $e:\Re^n \rightarrow \Re$ |
| | $c_i(\mathbf{x})$ | $i^{th}$ constraint function; $c_i:\Re^n \rightarrow \Re$ |
| | $\mathbf{c}(\mathbf{x})$ | column vector of constraint functions; $\mathbf{c}:\Re^n \rightarrow \Re^m$ |
| | $\mathbf{x}^*, \lambda^*$ | value at local minimizer |
| | $\nabla\mathbf{c}(\mathbf{x})$ | Jacobian of constraints; $\begin{bmatrix} \dfrac{\partial c_1}{\partial x_1} & \cdots & \dfrac{\partial c_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial c_1}{\partial x_n} & \cdots & \dfrac{\partial c_m}{\partial x_n} \end{bmatrix}$ |

# Chapter 1
# Introduction

**This dissertation examines techniques for enabling high-end graphics workstations to maintain basic physical properties of a protein model while a chemist interactively changes the model's structure. I combine constrained minimization, parallel processing, an approximation of protein properties, and a method for reduction of variables into a prototype system that maintains one update per second on a Silicon Graphics 240-GTX as a chemist moves atoms in an eight-hundred atom model. The computational model in this** research narrows the gap between interactive graphical modeling based on purely geometrical operations and batch modeling based on simulation of physical properties.

This research makes a step towards including modeling of physical properties in interactive computer graphics. Within the last ten years a trend in computer graphics has been to increase scene realism by using physically-based models. Animators use physically-based modeling to create realistic, detailed behavior. Most animations generated with physically-based modeling, to date, required minutes to hours of computation for each frame. This large computation time has kept physically-based modeling out of interactive graphics systems except with small, simple models.

As a goal, I sought to build a modeling system that maintains marginal interactivity (one to three updates per second) with real chemical modeling tasks and whose computation increases linearly with model size. I constrain stiff properties such as bond lengths rather than model their potential energy. An algorithm that maintains these constraints while minimizing the energy in the rest of the model can make much larger steps per iteration than an algorithm without constraints that minimizes the energy in the entire model. However, the algorithm must satisfy the constraints (a set of nonlinear equality functions) on each iteration. In order to solve these efficiently I only allow constraints among atoms reached by traversing one, two, or three bonds (no such restriction is placed on energy functions).

This property yields a banded system of equations. Solving this requires time *linearly* proportional to the number of atoms rather than *cubically* proportional, as would be required if arbitrary atoms appear in the same constraint.

## 1. Problem description

A protein, to a first approximation, contains fixed bond lengths, bond angles, and planar segments. Figure 1.1 shows three sequential segments (peptides) in a protein with vectors representing bonds between atoms and gray areas denoting planar regions. The only degrees of freedom in the figure are rotations between the planar segments (shown with arrows about the N-$C_\alpha$ and $C_\alpha$-C bonds). A linear sequence of the segments comprise the protein *backbone*. Attached to the atom between each segment ($C_\alpha$) are *sidechains* (not shown) with additional fixed length and angle properties, and often additional allowable rotations. The sidechains are short relative to the backbone (up to eighteen atoms versus hundreds to thousands of atoms).

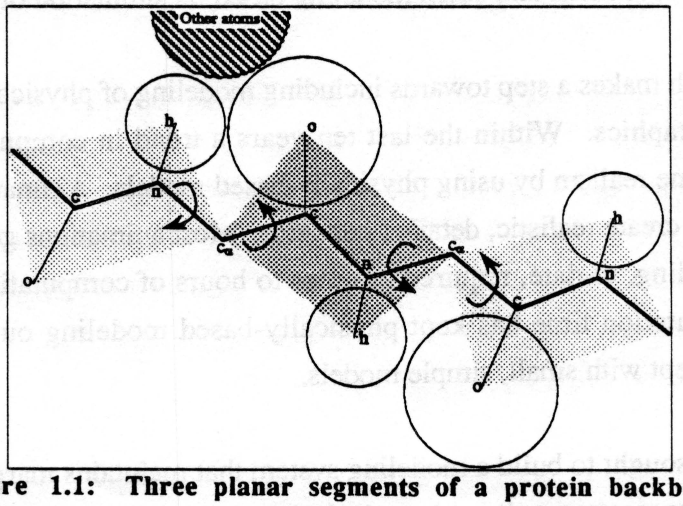


**Figure 1.1: Three planar segments of a protein backbone.**

Proteins also contain attractions and repulsions between non-bonded atoms (points without connecting vectors). Attractions hold nearby atoms together, whereas repulsions maintain a minimal separation defined by the atoms' electron shells. Figure 1.1 shows near-ideal separation between *H* and *O* atoms with adjacent circles (ideally, the circles should be tangent). If two circles overlap, their atoms repel each other; otherwise, they attract. This is called a *van der Waals* interaction.

Chemists build and change molecular models with interactive molecular modeling systems such as *Sybyl* [Tripos 1988], *Quanta* [Polygen 1991], and *Insight* [Biosym 1991]. A chemist usually changes the backbone conformation in these systems by rotating the segments about the N-$C_\alpha$ and $C_\alpha$-C bonds. This prevents changes in the bond lengths and angles in the segments. However, adjusting an interior segment with this method often requires numerous rotations, since even a small rotation moves all the atoms further along the backbone. Thus achieving a desired interior adjustment is an inverse-kinematics problem with perhaps hundreds of joints. Additionally, interactive systems do not maintain proper non-bonded atom separations during a modeling session. A chemist must return a protein to physically-valid geometry by either laborious manual adjustments using these rotations, or by batch energy minimization. Batch minimization automates model repair but often changes the model differently than the user desires.

Professor David Richardson of the Duke University Department of Biochemistry posed the driving problem in this research, called ***protein sculpting***: let a chemist manipulate a graphical representation of a protein while the model simultaneously mimics fundamental behaviors in the real protein.

## 2. Thesis and demonstration

The thesis of this dissertation is that current, high-end graphic workstations with multiple processors can interactively maintain physically-valid protein properties by combining the following:

1. a constrained minimization algorithm that converges rapidly and scales linearly with protein size;
2. parallel and concurrent execution of the constrained minimization algorithm;
3. hierarchical models that reduce the number of variables by exploiting inherent rigid substructure.

I demonstrate the validity of this thesis in two ways. First I demonstrate the proof of concept through a graphical modeling system, called *Sculpt,* based on these techniques. *Sculpt* lets a user move any atom by first attaching a symbolic spring between it and the cursor and then dragging the cursor in the desired direction. Throughout the dragging process, *Sculpt* polls the cursor position and adds the strain energy of that spring to the potential energy modeled among bonded and non-bonded atoms in the protein. *Sculpt* then finds a local minimum of the new total energy that also maintains rigid bond lengths, angles, and planar segments (Section 3 details the protein model). Lastly, *Sculpt* displays the results. Figure 1.2 shows a photograph of a *Sculpt* session with depth-cued vectors representing bonds between atoms; cyan denotes the central backbone, and gray denotes sidechains connected to the backbone. The backbone winds through four helices. Each gold coil shows a spring left by a user that pulls an atom towards the three-dimensional position marked with the gold thumbtack. Figure 1.2 shows a medium-size protein, called Felix [Hecht 1990], with 760 atoms. The model contains 2205 rigid constraints (bond length, angle, and others) and approximately 8030 energy functions (attraction, repulsion, and others). On a four processor Silicon Graphics 240-GTX, *Sculpt* maintains 0.8 updates per second with this model.

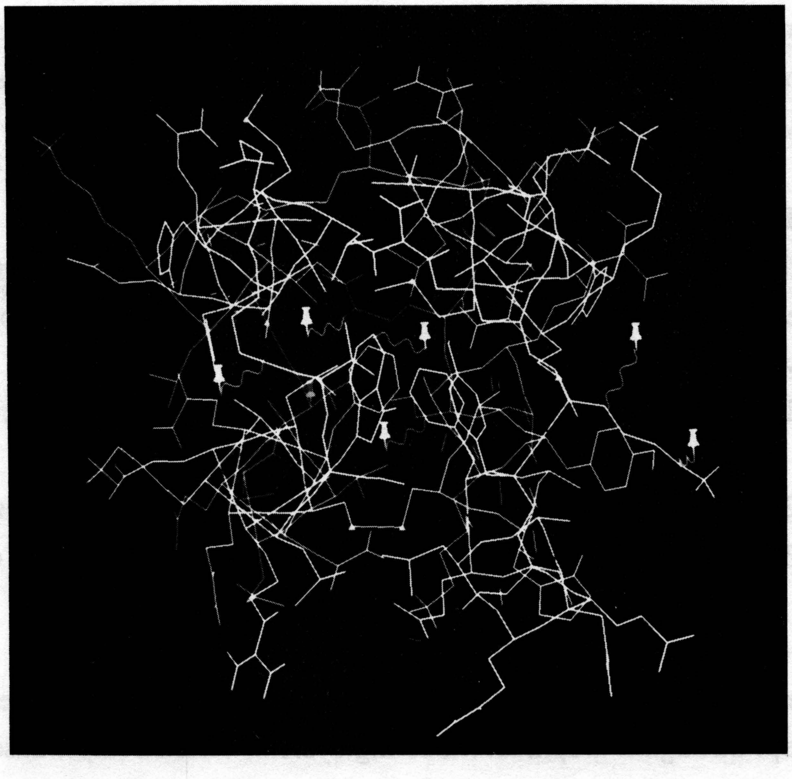


**Figure 1.2: Photograph of a *Sculpt* session.**

Second I prove the computation in the constrained minimization algorithm increases linearly with model size. In this application I assume atom positions in a protein model at local energy minima are more important than the dynamics of the atoms for reaching the minima. This assumption lets *Sculpt* minimize strain energy in the model rather than simulate model dynamics. The time steps in the algorithm are large enough for interactive performance on medium-size protein models. The algorithm also has linear complexity when used with other applications (e.g. articulated figures) that have more joints in the spine than in any limb. An additional benefit of the algorithm is that most of its steps can execute in parallel.

The remainder of this chapter surveys the protein model, algorithms, analysis, and performance that are described in detail in the subsequent chapters. The following list summarizes the contents of Chapters 2 through 10.

Chapter 2 describes protein properties and how I model them.
Chapter 3 describes algorithms for finding unconstrained and constrained minima.
Chapter 4 surveys related molecular modeling and computer graphics systems.
Chapter 5 proves the number of operations in the constrained minimization algorithm increases linearly as the number of atoms in the model increases.
Chapter 6 analyzes the steps in the algorithm that can execute in parallel.
Chapter 7 describes a method that reduces the number of variables by combining atoms into rigid bodies.
Chapter 8 discusses important implementation details.
Chapter 9 describes a user session and performance results from the session.
Chapter 10 concludes with a discussion of future research.

## 3. Protein model

The protein properties addressed in this research are listed in the center column of Figure 1.3 and described in Chapter 2. The left column separates the properties into bonded and non-bonded categories. Each bonded property is defined throughout a modeling session; *Sculpt* does not model breaking and forming bonds. The non-bonded properties include the attractive and repulsive van der Waals interaction discussed in Section 1, electrostatic interactions, and solvent interaction. Non-bonded properties are applied to atoms within a given radius of each other. As atoms move, the pair-wise interactions change. Electrostatic interactions and solvent interaction are not currently implemented in *Sculpt* because they seriously degrade performance; the linearity of the minimization algorithm, however, still holds. For simplicity of implementation I specify the hydrogen bonds at program initialization. Hydrogen bonds are actually weak attractions between certain nearby, non-bonded atoms. A future version will model hydrogen bonds between atoms within a given radius of each other.

| Property type | Protein Property | Mathematical Model |
|---|---|---|
| Bonded | Bond length | Constrained to ideal value |
| | Bond angle | |
| | Fixed dihedral angle | |
| | Variable dihedral angle | Spring energy from nearest ideal value |
| | Hydrogen bond | |
| Non-bonded | Van der Waals potential | 4–8 Lennard-Jones potential |
| | Electrostatic charge | Coulomb potential (not implemented) |
| | Solvent interaction | Not treated |

**Figure 1.3: Protein properties addressed in this research.**

The right column in Figure 1.3 divides the computational model of each property into constraints and energies. A *constraint* requires the value of a function equal some ideal value. An *energy* (or *re*straint) imposes a penalty as a function value varies from its ideal value. Consider a distance function, *d*, that represents the separation of two bonded atoms. Modeling this as a constraint requires that $d = \bar{d}$ or equivalently, $d - \bar{d} = 0$, where $\bar{d}$ is the ideal separation. Modeling this as a spring energy function evaluates the variance of the separation from the ideal value using Hooke's Law: $Energy = k(d-\bar{d})^2$, where $k$ denotes the stiffness of the spring. The van der Waals interactions and the electrostatic interactions use empirical potential energy models.

The non-bonded interactions are modeled with energy functions. Additionally, I model weak and multi-value bonded properties with energy functions and stiff bonded properties with constraint functions. For example, the energy required to change certain dihedral angles is three orders of magnitude weaker than that required to change bond lengths. I model these dihedral angles with springs with an ideal value equal to the ideal angle and model bond lengths with constraints fixed to their ideal values. A hydrogen bond is modeled with two energy functions—one models the length and the other models the angle of the bond. The strength of a hydrogen bond is approximately two orders of magnitude weaker than that of the other bonds in a protein.

**The energy and constraint functions are grouped into the following constrained minimization problem. Assume each point has three variables that denote its three-dimensional position. Let the vector *x* (bold face denotes vectors) with *n* elements represent the independent variables (atom positions). Let *e(x)* equal the sum of all the energy functions. Let *c(x)* represent the vector of the *m* constraint functions. Specifically, row *i* in *c(x)* contains a constraint function, $c_i(x)$, minus its ideal value, $\overline{c_i}$, as follows:**

$$\mathbf{c(x)} = \begin{bmatrix} c_1(\mathbf{x}) - \bar{c}_1 \\ c_2(\mathbf{x}) - \bar{c}_2 \\ \vdots \\ c_m(\mathbf{x}) - \bar{c}_m \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

**On each update as a user tugs an atom, *Sculpt* finds a local minimum of the total energy, *e(x)*, while maintaining the set of constraint functions, *c(x) = 0*.**

## 4. Constrained minimization algorithm

**Minimizing an arbitrary function subject to arbitrary constraints is an open research problem. The class of constrained minimization problems in this research has the following properties:**

1. **nonlinear energy and constraint functions;**
2. **equality constraints;**
3. **continuous energy and constraint functions through the second derivative;**
4. **initial variables that satisfy or nearly satisfy constraints;**
5. **starting point is usually very close to a local minimum.**

The last property allows rapid convergence to a local solution. A new constrained minimization runs each time a user tugs an atom. The only differences in the total energy from the previous invocation are the changes in energies from a user spring and near-neighbor interactions resulting from prior atom movements. Neither case changes the total system energy very much. In most cases, a small change in the total energy only slightly shifts the minimum.

A solution $(\mathbf{x}^*, \lambda^*)$ of the following minimax problem, for sufficiently large *p*, is also a solution to the constrained minimization problem [Hestenes 1975]:

$$\min_{\mathbf{x}} \max_{\lambda} \mathrm{L}(\mathbf{x}, \lambda, \mathrm{p}) = \min_{\mathbf{x}} \max_{\lambda} \mathrm{e}(\mathbf{x}) - \lambda^{\mathrm{T}} \mathrm{c}(\mathbf{x}) + \frac{\mathrm{p}}{2} \mathrm{c}(\mathbf{x})^{\mathrm{T}} \mathrm{c}(\mathbf{x})$$

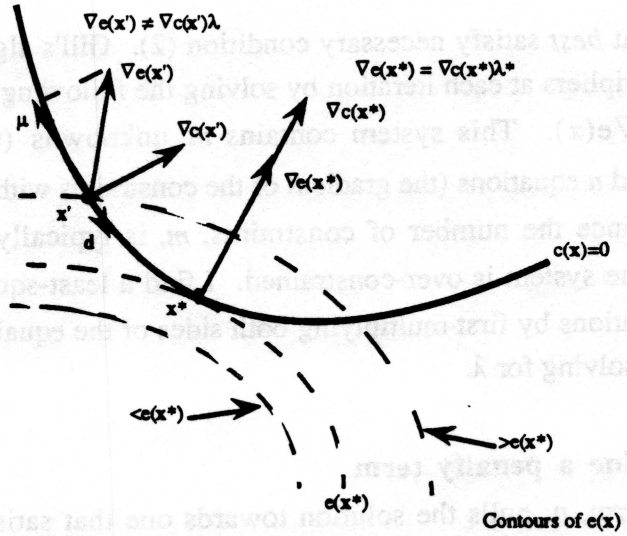


**Figure 1.4: Example of first-order necessary conditions at a constrained minimum.**

For penalties greater than some finite value, a point $x$ minimizes $L(x,\lambda^*,p)$ if and only if it is a constrained minimum of the original problem [Gill 1981, p. 226]. Additionally, a sequence $x^i$ that minimizes $L(x,\lambda^i p)$, for a sequence of multipliers that converge to $\lambda^*$, converges to $x^*$ [Hestenes 1975, p. 308]. I use these results in the algorithm listed in Figure 1.5 to find the saddlepoint. This algorithm is first presented in Gill [Gill 1981, p. 227]; Witkin presents a similar algorithm for constrained dynamics [Witkin 1990]. Step 3 minimizes the augmented Lagrange function by following the negative gradient. Steps 1 and 2 are more complex and are discussed in the following subsections. Chapter 3 details the steps in this algorithm.

| | |
|---|---|
| 0. Set initial x | Initializes variables to current position |
| 1. Find $\lambda$: $[\nabla c(x)^T \nabla c(x)]\ \lambda = \nabla c(x)^T \nabla e(x)$ | Estimate Lagrange multipliers |
| 2. $p \leftarrow \|\nabla e(x) - \nabla c(x)\lambda\|$ | Determine a penalty weight |
| 3. $x^{change} \leftarrow -\nabla e(x) + \nabla c(x)\lambda - p\nabla c(x)c(x)$ | Minimize augmented Lagrange function over $x$ given $\lambda$ and $p$ from Steps 1 and 2 |
| 4. $x \leftarrow x^{old} + x^{change}$; Goto step 1 | Update; repeat if change is greater than some threshold |

**Figure 1.5: Algorithm that finds a local constrained minimum.**

### 4.1. Estimate Lagrange multipliers

I use a modification of the first-order multiplier method [Gill 1981, p. 248] to estimate the Lagrange multipliers. Given the current variables, $x$, this method finds the Lagrange multipliers that *best* satisfy necessary condition (2). Gill's algorithm estimates the ideal Lagrange multipliers at each iteration by solving the following system of equations for $\lambda$: $[\nabla \mathbf{c}(\mathbf{x})]\ \lambda = \nabla \mathbf{e}(\mathbf{x})$. This system contains $m$ unknowns (the number of Lagrange multipliers) and $n$ equations (the gradient of the constraints with repect to each of the atom positions). Since the number of constraints, $m$, is typically less than the number of positions, $n$, the system is over-constrained. I find a least-squares approximation to the system of equations by first multiplying both sides of the equality by the transpose of the matrix before solving for $\lambda$.

### 4.2. Determine a penalty term

The penalty term, $p$, pulls the solution towards one that satisfies the constraints. The previous step estimates the Lagrange multipliers using a first-order approximation of the constraints. Since the constraints are nonlinear, this approximation lets the solution drift from the constraints. The penalty term keeps this from moving beyond some limit. The penalty term is set to the error in the least-squares approximation.

### 4.3. Parallel execution of algorithm

Many of the steps in this algorithm can execute in parallel. *Sculpt* evaluates a set of the constraint and energy functions and their derivatives on separate processors. *Sculpt* also evaluates in parallel the elements of vectors calculated in Steps 1-3 from vector-matrix and matrix-matrix operations.

## 5. Algorithm analysis

This section surveys the main properties in the protein model that yield linear computational complexity, O($n$), with this algorithm. The full proof is given in Chapter 5. The two important properties assumed are that the number of constraint and energy functions increases linearly with protein size and that constraint functions are only defined on topologically-near atoms. The former gives a linear increase in the evaluation time for the constraint and energy functions and their derivatives. The latter, coupled with a pre-processing sort of the constraint gradient matrix, yields a band-diagonal matrix (all nonzeros lie within a fixed distance from the diagonal).

### 5.1. Linear complexity of energy and constraint evaluation

The number of bonded and non-bonded functions increases linearly with the number of atoms. Each atom bonds to at most four other atoms which bond to at most three others, etc. This bounds the number of distance, angle, and dihedral angle functions for each atom. Therefore, the total number of bonded functions increases linearly with the number of atoms. The maximum number of constraints per variable, $k_{cv}$, is used in the matrix-multiplication analysis.

For each atom, *Sculpt* determines the other atoms within a given distance and evaluates the van der Waals potential energy between them. Only a finite number of atoms fit within this neighborhood because each atom's electron shell occupies a nonzero volume [Levinthal 1966]. *Sculpt* uses an O($n$) algorithm described in [Bentley 1979] that determines the neighbor lists. The algorithm uniformly subdivides space into cubes with the given radius on each side, deposits each atom into the cube containing its three-dimensional position, and sets the neighbor list for each atom to the atoms in its cube and adjacent cubes. Atoms in adjacent cubes may be outside the neighborhood. A final step prunes the lists of such atoms.

### 5.2. Constraint topology

Matrix-matrix multiplication and linear equations solution require O($nm^2$) and O($m^3$) operations, respectively, for an arbitrary number of constraints defined among arbitrary points. However, constraints in the protein model are defined only on atoms within a fixed number of bonds. Figure 1.6 shows a graph of the bond topology in a protein (lengths and angles are not drawn to scale). Nodes represent model variables (a three-dimensional coordinate for each atom). Nodes are numbered by a depth-first traversal beginning at the left. Constraints are defined on nodes connected by a fixed number of arcs. For example, a distance constraint is defined for each pair of connected nodes. Angle and dihedral angle constraints are defined on nodes connected by two and three arcs, respectively.

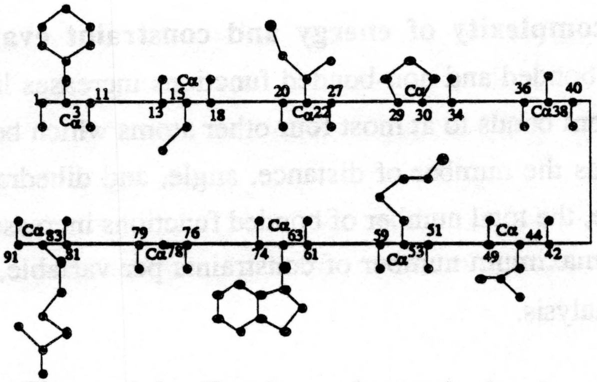


**Figure 1.6: Topology of protein bonds.**

The maximum separation between the lowest and highest variable indices in a constraint is bound by a constant, $k_{span}$. This constant is a function of the maximum number of variables referenced in a constraint and the maximum number of atoms in any sidechain. The worst case in a protein occurs when a dihedral angle constraint is defined on backbone atoms around a sidechain with the most atoms (tryptophan).

Some proteins violate this assumption by containing a few bonds, called disulfides, between atoms in separate sidechains. Chapter 5 shows that the computational complexity analysis remains valid when there is a fixed number of constraints among arbitrary atoms.

## 5.3. Linear complexity of matrix algorithms

The linear complexity of the matrix algorithms arises from the pattern of nonzeros in the matrices. *Sculpt* stores two matrices, the transpose of the constraint gradient (denoted $J^T$ for *Jacobian* transpose) and its product with the Jacobian ($J^TJ$). Both matrices are very sparse (typically more than ninety-five percent of the entries are zero).

### 5.3.1. Contents of matrices

**Jacobian transpose, $J^T$.** The dimension of this matrix is $m \times n$. Element *(i,j)* of the Jacobian transpose holds the first partial derivative of constraint *i* with respect to variable *j*.

**Matrix product, $A \equiv J^TJ$.** The dimension of this matrix is $m \times m$. Nonzero elements in this matrix are related to nonzero elements in the Jacobian transpose. Matrix multiplication defines an element $a_{r,c}$ in *A* as the inner product of row *r* in the left matrix and column *c* in the right matrix. Column *c* in a matrix is also row *c* in its matrix transpose. Using this information, $a_{r,c}$ equals $\sum_k J^T_{r,k} J_{k,c} = \sum_k J^T_{r,k} J^T_{c,k}$. The element is nonzero only when rows $r$ and $c$ in the Jacobian transpose contain a nonzero entry in the same column (i.e. when constraints $r$ and $c$ reference the same variable).

*Sculpt* determines the nonzero elements of $A$ at program initialization by comparing each row in the Jacobian transpose with the other rows for common, nonzero columns (requires O($m^2$) operations). Since bonds are not broken or formed during a modeling session, rows (constraints) are not added or removed in the Jacobian transpose. Therefore, the sparsity pattern of $J^T$ and $A$ remains constant.

### 5.3.2. Sparsity pattern with a sorted Jacobian transpose

Matrix $A$ is *band-diagonal* if the rows of the Jacobian transpose are sorted relative to the smallest nonzero column index. The nonzero elements in a band-diagonal matrix lie within a constant distance (bandwidth) from the diagonal. What is the bandwidth of an arbitrary row, $i$, in $A$? Column $j$ in row $i$ is nonzero if and only if constraints $i$ and $j$ reference the same variable.

Constraint $i$ references a fixed span of variable indices. Each variable in that span is in at most $k_{cv}$ constraints. Therefore, the number of constraints that can reference a variable in constraint $i$ is bounded (see Figure 1.7). The bandwidth, $b$, of $A$ is a function of the number of constraints defined on a variable times the span of variables in a constraint, $k_{cv}k_{span}$.

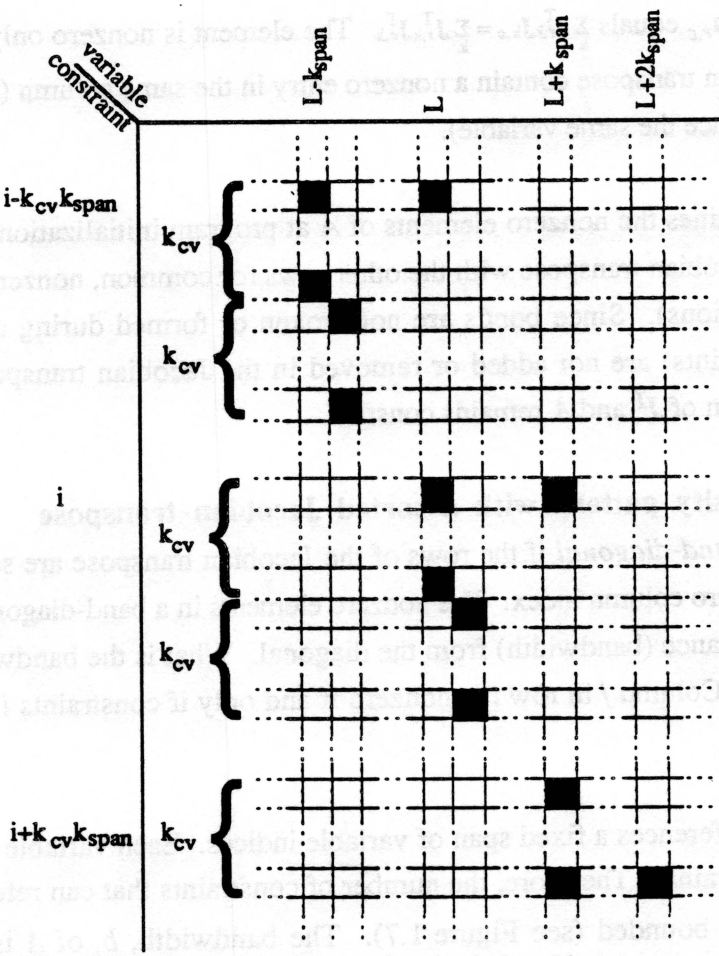


**Figure 1.7: Worst-case structure of sorted Jacobian transpose around constraint *i*.**

### 5.3.3. Matrix algorithms

**Matrix-matrix multiplication, A←J$^T$J.** The limited bandwidth limits the number of nonzero elements in *A* to twice the bandwidth times the number of rows. Computing the value of a given element, *(r,c),* requires multiplying rows *r* and *c* of the Jacobian transpose. Each of these contains a fixed number of elements, so the entire multiplication requires O(*bn*) operations.

**Matrix-vector multiplication, b←J$^T$∇e.** This step multiplies each of the *m* rows in the Jacobian transpose by a vector. Each row contains a fixed number of nonzero elements, so the entire multiplication requires O(*m*) operations. The linear complexity of the matrix-vector multiplication in Step 3 follows a similar argument.

**Linear equation solution, A$\lambda$=b.** Gaussian elimination requires $O(b^2n)$ operations on a band-diagonal matrix.

## 6. Hierarchical model

I also investigate a method for improving user manipulation and system performance by replacing segments of a protein with rigid or deformable objects. For example, consider replacing the variables, energies, and constraints in a backbone segment with a rigid body that only changes position and orientation. A change in the orientation or position of the rigid body simultaneously changes the backbone atoms in the segment; sidechain atoms still move freely. This lets a chemist change the relative orientation of two segments without changing their internal configuration, which is a much more natural manipulation than tugging each atom separately. Independently of this advantage, the performance, in general, improves because a small set of variables that define the rigid body and the functions that connect it to the rest of the model replaces the larger set of variables and functions defined within the body.

A deformable object such as a coil could represent a segment of the backbone that forms a helix. In addition to the orientation and position of a rigid object, deformable objects have additional parameters that define its shape. The coil, for example, could allow bending and twisting. Rigid and deformable objects are required to move their atoms in a physically-valid manner—one that maintains an energy minimum and does not violate constraints.

Chapter 7 examines using inherent protein structure to aid specification of groups. The chapter shows the following hierarchical model of protein structure: a protein consists of a sequence of residues; each residue contains a backbone and sidechain; each of these contains atoms (see Chapter 2 for a discussion of protein structures). A chemist could then specify atoms for a group by selecting a node in the hierarchy. Superimposed on this hierarchy are inherent protein structures such as a helix defined on a sequence of residues or a sheet defined on a set of residue sequences.

The rigid and deformable groups require few changes to the constrained minimization algorithm and do not change the linear computational complexity. The constrained minimization algorithm contains a vector of variables, $x$. The variables are now augmented to include group variables. Since an atom's position may be defined by variables in a group, the gradient operator must take partial derivatives with respect to group variables as well as cartesian coordinates. When a group is inserted (removed), constraints are added (removed) from the model. This changes rows in the two matrices of the algorithm. Chapter 7 shows a method for efficiently making such changes.

The matrix bandwidth can increase using groups because the number of constraints per variable, $k_{cv}$, can increase. If only the backbone atoms in a long segment are placed in a group, then many sidechains connect to the group. Each connection requires distance (and other) constraints defined on the group's variables which can increase $k_{cv}$. On the contrary, if the backbone *and* sidechains of a long segment are placed in a group, the only connections to the group are at its ends. The number of constraints per variable in this case does not increase. The complexity of the linear equation solution increases with $b^2n$. Performance can actually degrade if the square of the bandwidth increases faster than the decrease in variables. Chapter 9 presents performance results that demonstrate this.

The current implementation of *Sculpt* contains only rigid, not deformable, objects. The objects are defined at the beginning of a modeling session and cannot be removed or added. *Sculpt* is designed for future enhancements to provide deformable groups and on-the-fly creation and deletion of groups. Figure 1.8 shows a photograph of the Felix model with two rigid cylinders representing the backbone atoms in two of the four helices.

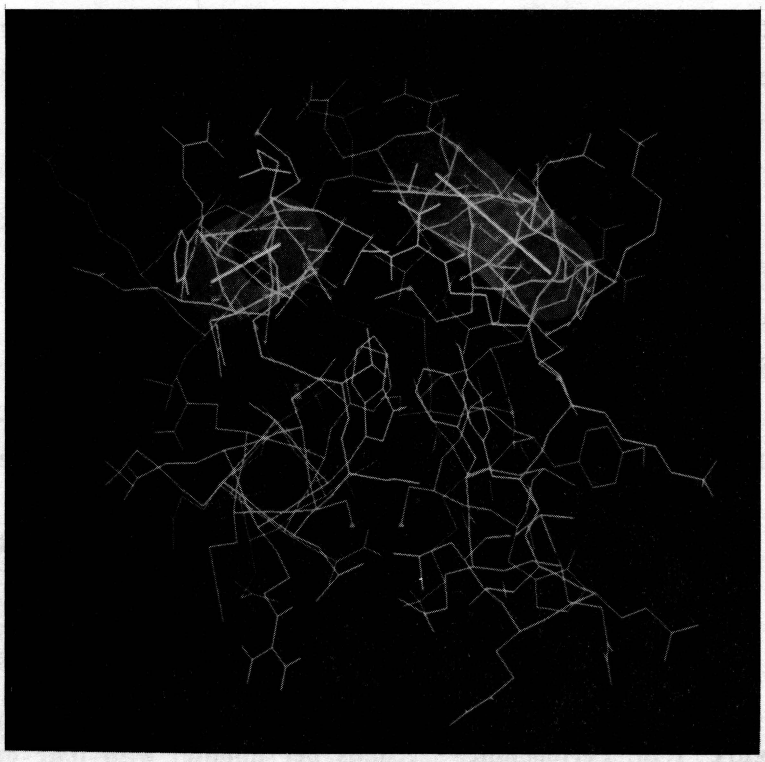


Figure 1.8: Photograph of Felix with two rigid helices.

## 7. System performance

The following performance analysis describes system behavior with four protein models. The models include (1) the Felix protein pictured in Figures 1.2 and 1.8, (2) two of the four helices from Felix, (3) a segment with ten residues, and (4) a small segment with four residues. Figure 1.9 summarizes the number of energy functions, with and without the near-neighbor interactions, and constraint functions.

| Model | Atoms | Variables | Constraints | Bonded energies | Total energies |
|---|---|---|---|---|---|
| 1 | 760 | 2280 | 2205 | 428 | 8029 |
| 2 | 355 | 1065 | 1027 | 198 | 3465 |
| 3 | 99 | 297 | 282 | 43 | 788 |
| 4 | 36 | 108 | 96 | 18 | 216 |

Figure 1.9: Statistics for the four models used in performance analysis.

The performance results listed below come from running *Sculpt* on a four processor Silicon Graphics 240-GTXB [Akeley 1988] using double-precision floating-point arithmetic.

Figures 1.10 and 1.11 show performance results with the four protein models. The performance (seconds per update) includes the time to receive a user tug, run a constrained minimization, and re-display the screen. I list the performance for molecular simulations that model and do not model near-neighbor interactions. The performance results for simulations without near-neighbor interactions are given for one and four processors. The performance results with near-neighbor interactions use four processors. The list of neighbors for each atom is implemented with a linked list rather than an array; this requires more computation but greatly easied implementaation. Due to this inefficient implementation, I split the performance results for the near-neighbor interactions into two categories. The first uses the same neighbor list throughout a session, while the second computes a new list on each iteration. Only one processor creates neighbor lists and evaluates non-bonded interaction energy; these tasks can be parallelized in a future version.

| | Without near-neighbor interactions | | With near-neighbor interactions (4 CPUs) | |
|---|---|---|---|---|
| Model | 1 processor | 4 processors | Same list | New list |
| 1 | 1.405 | 0.954 | 1.228 | 1.603 |
| 2 | 0.586 | 0.396 | 0.514 | 0.689 |
| 3 | 0.147 | 0.105 | 0.126 | 0.169 |
| 4 | 0.045 | 0.048 | 0.047 | 0.054 |

**Figure 1.10: Performance (seconds per update) with four models using an SGI 240-GTX.**

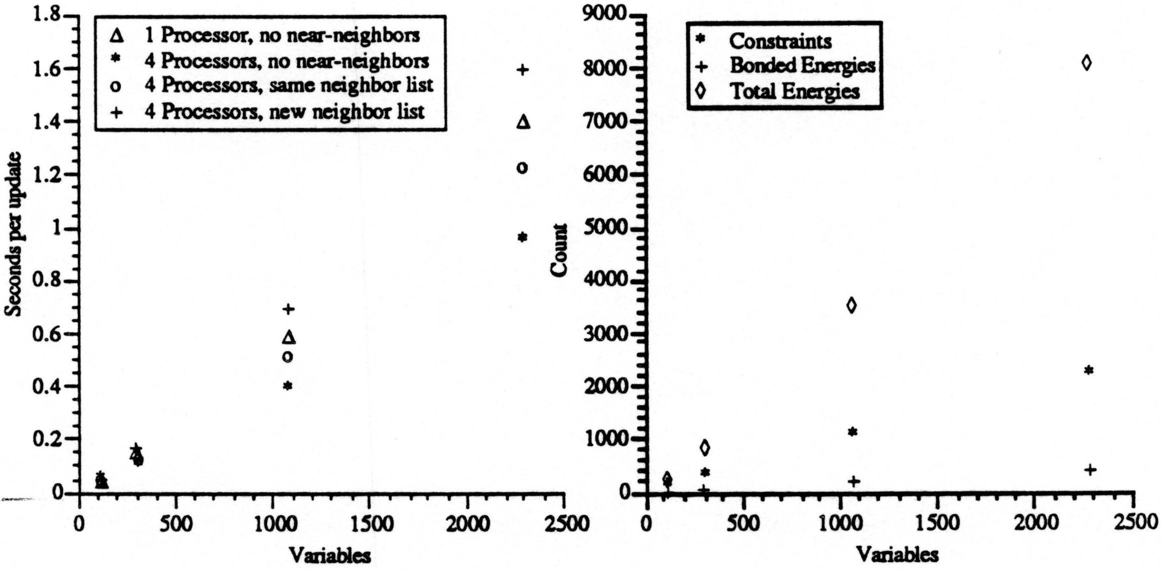


**Figure 1.11: Left plot shows linear increase in compute-time with model size; right plot shows linear increase in the number of constraints and energies with model size.**

The update rates, though far from those needed for smooth interaction, allow productive new research in biochemistry. Professors David and Jane Richardson, collaborators from the Duke University Department of Biochemistry, use a preliminary version of *Sculpt* for their research in protein design (see Chapter 9). They believe *Sculpt* significantly improves productivity and understanding over previous molecular modeling systems. Maintaining a physically-valid model throughout a modeling session relieves the user from the time-consuming task of returning the model to a physically-valid state. Immediately viewing the effect of an atom movement also provides better awareness of intricate interactions among atoms.

# Chapter 2
# Proteins

This chapter discusses the protein properties relevant to this research and the *Sculpt* model of them. Part I of the chapter introduces the properties to readers without a background in biochemistry; those familiar with proteins can skip this. Part II discusses the *Sculpt* model of proteins. This contains the specific model of each property and a discussion of its validity. Part II also lists the properties ignored and compares the model to other protein models.

A protein is an arbitrarily long, linear sequence of bonded amino acids (defined in part I). A protein contains from a few hundred to greater than fifty thousand atoms. Most proteins contain regions with regular geometry (e.g. a helix). A *subunit* is a single chain in close proximity that contains these regions; often several subunits stick together to form a large protein. The proteins considered in this research are called *globular* proteins. These contain subunits with roughly four hundred to three thousand non-hydrogen atoms. The largest proteins used in *Sculpt* so far contain approximately eight hundred atoms, including about 150 of the hydrogens.

The majority of the material and figures in this chapter comes from two books: *The Structure and Action of Proteins,* by Dickerson and Geiss [Dickerson 1969]; and *Principles of Protein Structures,* by Schultz and Schirmer [Schulz 1979]. Both books assume no prior knowledge of proteins and describe protein properties through examples and illustrations. A third book, *Prediction of Protein Structure and the Principles of Protein Conformation,* edited by Fasman [Fasman 1989], contains a detailed description of proteins and their properties.

## Part I – Protein Properties

### 1. Bonded interactions

Amino acids are the building blocks of proteins. Part A in Figure 2.1 shows a schematic of an uncharged amino acid. Lines represent bonds between atoms, and letters represent atom names (*C*arbon, *O*xygen, *N*itrogen, *H*ydrogen). Subscripts differentiate atoms of the same type, for example C and $C_\alpha$. The twenty common amino acids differ only in *sidechain* composition (denoted with ®). The sidechain in the smallest amino acid, glycine, contains only one hydrogen. The sidechains in large amino acids, such as arginine and tyrosine, contain fifteen to eighteen atoms, (refer to Figure 2.1, Part B). A sidechain can only contain carbon, nitrogen, hydrogen, oxygen, and sulfur (*S*) atoms. The arrangement of the four atoms bonded to a $C_\alpha$ is tetrahedral and always left-handed as depicted in Figure 2.2. The $C_\alpha$-C, $C_\alpha$-®, and $C_\alpha$-N bonds appear in clockwise order when viewed along the H-$C_\alpha$ bond.

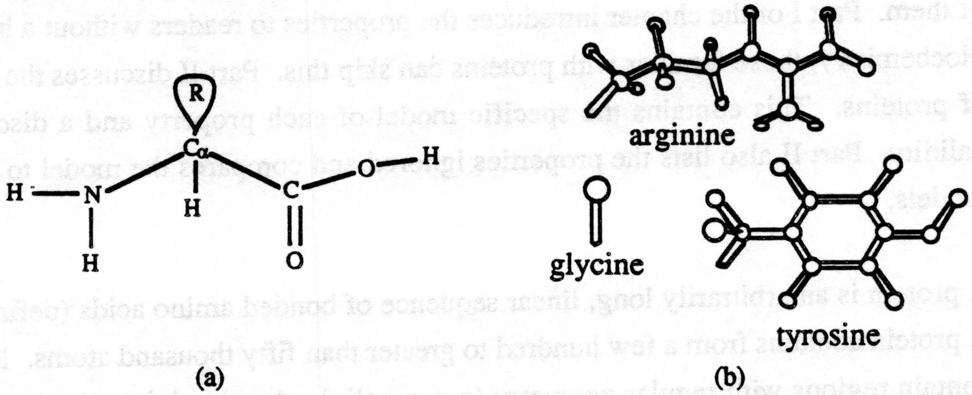


**Figure 2.1: (a) Schematic of an amino acid's bonds; (b) Three of twenty possible sidechains for the ® in part (a).**

A protein is a linear sequence of amino acid residues resulting from a chemical reaction that bonds the acids. Two amino acids can form a bond, called a *peptide bond,* between the nitrogen (N) in one and the carbon (C) in the other. The chemical reaction that forms the bond releases a water molecule created with the O-H from the carbon and one H from the nitrogen. A similar reaction occurs between positively charged (extra hydrogen on nitrogen) and negatively charged (missing hydrogen from oxygen) amino acids. The structure that remains after the reaction is called an amino acid residue, or *residue* for short.

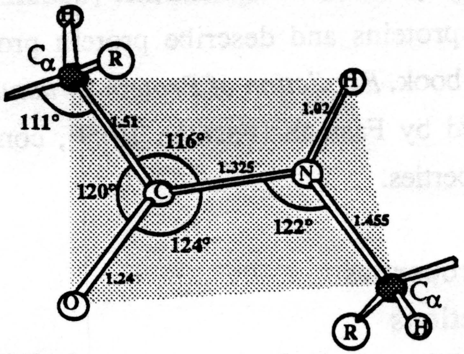
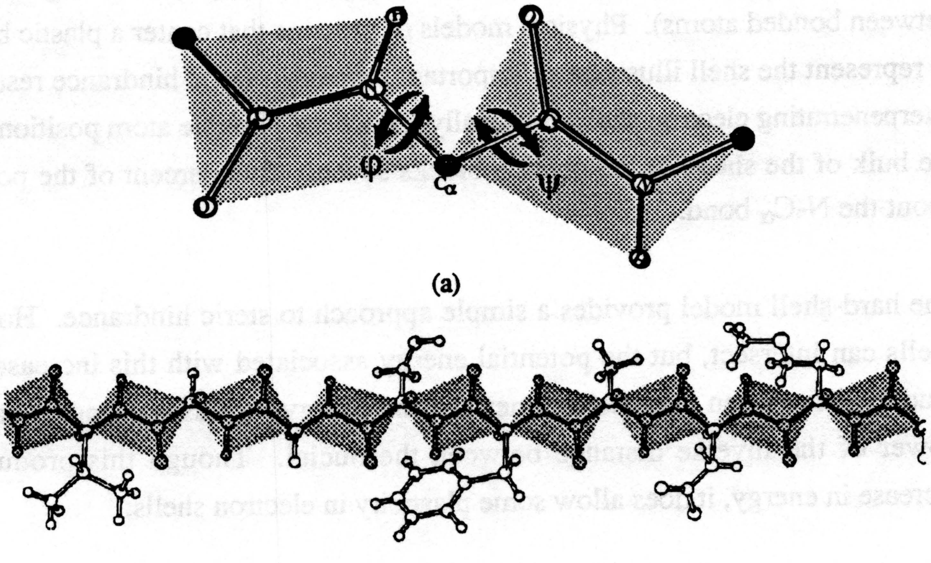


**Figure 2.2: Geometrical arrangement of atoms and bonds in peptides.**

The group of six atoms bonded to and including the carbon and nitrogen of the peptide bond ($C_\alpha$-C, C-O, C-N, N-H, N-$C_\alpha$) is called the *peptide*. Peptides are nearly rigid, planar structures. Figure 2.2 shows one peptide and its properties; bond lengths are in Ångstroms (1 Ångstrom = $10^{-10}$ meters), and bond angles are in degrees. The lengths and angles rarely change more than a few percent because the force required is not commonly developed in protein structures. The atoms bounding and within each grey area are co-planar. Atoms rarely move out of the plane more than ten degrees. The peptide atoms are called the *backbone* or mainchain of a protein; the ® atoms are the *sidechains*.

Significant conformational variation in proteins occurs only between peptides and along sidechains. Peptides rotate about the N–$C_\alpha$ bond (denoted $\varphi$) and the $C_\alpha$–C bond (denoted $\psi$). Figure 2.3 illustrates the $\varphi$ and $\psi$ angles and shows a segment of a protein containing nine peptides with different sidechains.



(a)

(b) [Dickerson and Geis 1969]

**Figure 2.3: (a) Illustration of $\varphi$ and $\psi$ angles between two peptides, and (b) a series of peptides illustrating $\varphi$ and $\psi$ rotations and sidechains.**

Atoms in separate sidechains infrequently form covalent bonds (a bond formed by shared electrons) between one another. The only such bond seen at all commonly is the disulfide bond formed between the sulfur atoms in two nearby cystine sidechains. This bond helps hold together residues that are arbitrarily separated along the backbone. Many proteins do not contain any disulfide bonds; those that do have one to ten in common sizes of subunits.

## 2. Non-bonded interaction

Atoms that are not bonded to each other nevertheless influence the three-dimensional structure of a protein by attracting and repelling each other. The attractive energy between two non-bonded atoms is approximately three orders-of-magnitude smaller than in a peptide bond, but the cumulative attractive energy among all atoms rivals the magnitude of bonded energy. This section presents the potential energy associated with four non-bonded interactions: attraction and repulsion induced by orbiting electrons, attraction and repulsion between partially charged atoms, hydrogen bonds, and hydrophobic and hydrophilic interactions.

### 2.1. Electron shell repulsion and attraction

One or more electrons orbit the nucleus of each atom within its electron shells. The electron shells can be modeled as non-interpenetrating spheres (ignoring the overlap between bonded atoms). Physical models of proteins that center a plastic ball at each atom to represent the shell illustrate an important property: steric hindrance resulting from non-interpenetrating electron shells drastically reduce the possible atom positions. For example, the bulk of the shells in the peptide blocks seventy-five percent of the possible positions about the N-$C_\alpha$ bond.

The hard-shell model provides a simple approach to steric hindrance. However, electron shells can intersect, but the potential energy associated with this increases substantially. Studies show when two shells penetrate, the energy increases proportional to the twelfth power of the inverse distance between the nuclei. Though this produces a dramatic increase in energy, it does allow some plasticity in electron shells.

The repulsive potential is countered with an attractive potential. All atoms, even neutral, attract one another. Orbiting electrons around a nucleus induce an oscillating dipole. For two atoms each dipole polarizes the other. Together the oscillators are coupled and form an attractive potential. The energy in this attraction is proportional to the sixth power of the inverse distance between the nuclei.

Van der Waals discovered this attraction and repulsion between atoms in 1873. The Lennard-Jones model combines both attraction and repulsion into one potential energy function as shown in Figure 2.4.

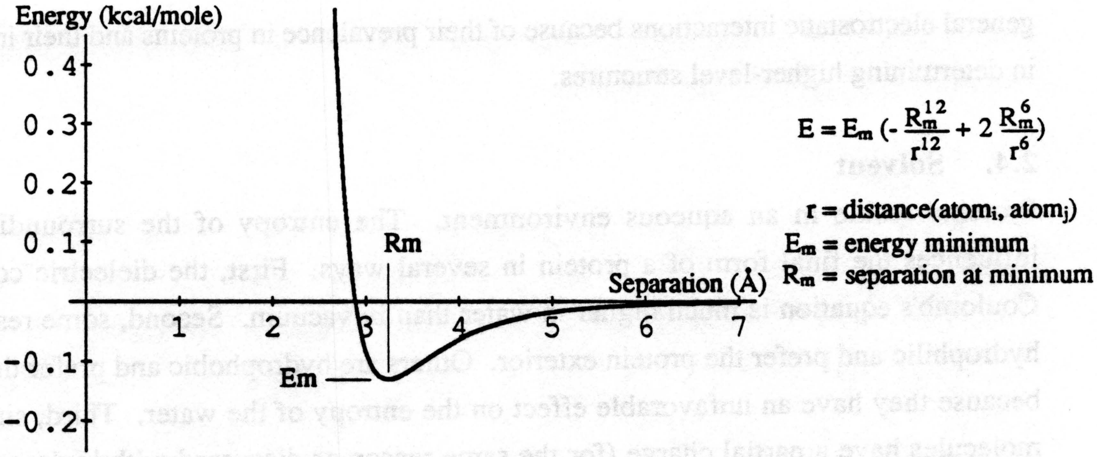


**Figure 2.4: Lennard-Jones formula and its graph using a minimum energy of -0.13 kcal/mole and a minimum-energy separation of 3.24 Ångstroms.**

## 2.2. Electrostatic interactions

Some atoms are fully or partially charged. In particular, the oxygens at the end of glutamate and aspartate are negative, the nitrogens at the end of lysine and arginine are positive, and the ring nitrogens in histidine are positive when the surrounding solution has a pH lower than 7.0. Also, bonds between some atoms leave an asymmetric distribution of electrons around the nucleus that forms a dipole, or partial charge. These fully and partially charged atoms interact with each other. The potential energy of the interaction is modeled with Coulomb's Law: Energy = $\frac{1}{\varepsilon}\frac{q_i q_j}{d}$, where *d* is the distance between atoms *i* and *j*, *qi* is the partial charge of atom *i*, and ε is the dielectric constant of the surrounding medium.

Coulomb's Law provides an approximation of the potential energy between charged atoms but is not entirely correct. The dielectric constant in Coulomb's Law assumes the conductivity of the medium is constant, which is not true in solvent, on the microscopic scale of bond lengths. The appropriate relationship for use with proteins is still an open research problem.

## 2.3. Hydrogen bonds

Many covalently-bonded hydrogens have a substantial partial charge. A nitrogen or oxygen that bonds with a hydrogen (e.g. a peptide N-H) pulls the hydrogen's one electron into its shell. This leaves the hydrogen with just one proton, only part of an electron, and a large positive partial charge. The resulting positive partial charge can bond with a negatively charged atom. These bonds, denoted *hydrogen bonds*, are approximately two orders of magnitude weaker than covalent bonds. Hydrogen bonds are distinguished from general electrostatic interactions because of their prevalence in proteins and their importance in determining higher-level structures.

## 2.4. Solvent

Proteins reside in an aqueous environment. The entropy of the surrounding water influences the final form of a protein in several ways. First, the dielectric constant in Coulomb's equation is much higher in water than in vacuum. Second, some residues are hydrophilic and prefer the protein exterior. Others are hydrophobic and prefer the interior because they have an unfavorable effect on the entropy of the water. Third, since water molecules have a partial charge (for the same reason as discussed with hydrogen bonds) and are mobile, they continuously form and break bonds with exterior, hydrophilic sidechains. This causes constant movement of some sidechains. Many unanswered questions remain concerning the role of water in protein structure.

# 3. Secondary structure

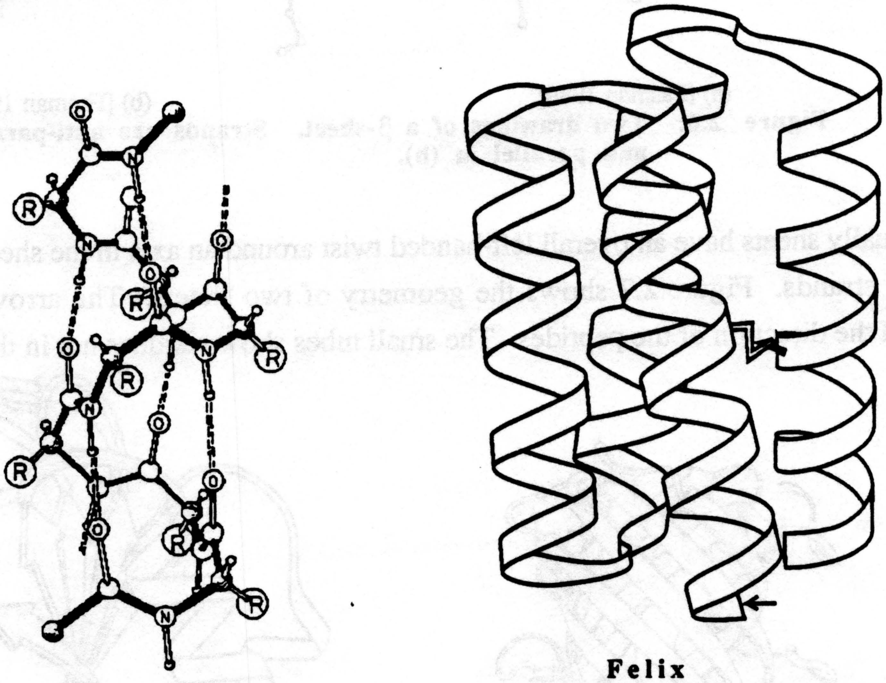
Secondary structures are regular arrangements of the backbone peptides. The structures form distinguishable geometries which are found in proteins. The most common structures are the helix and sheet. Together, two properties determine many of the structures: steric hindrance limits possible rotations between peptides, and hydrogen bonds between backbone hydrogen and oxygen atoms of different peptides hold peptides together. Richardson presents a thorough survey of secondary structures [Fasman 1989]. The following sections discuss the helix and sheet.

## 3.1. $\alpha$-helix

Successive peptide units with identical $(\varphi,\psi)$ angles and hydrogen bonds between adjacent turns form a helix. Helices can be described by the rise per residue, *d*, the number of residues per turn, *n*, and the radial distance, *r*, between the $C_\alpha$ and the helix axis. Steric hindrance and the limited number of potential hydrogen bonding sites allow only a few combinations of these parameters.

The right-handed $\alpha$-helix is the most prevalent secondary structure (and helix) found in proteins. It has 3.6 residues per turn, 1.5 Ångstroms rise per residue, and a 2.3 Ångstrom radius. The $\alpha$-helix has more favorable energy than other helices. The oxygen in one peptide forms a hydrogen bond with the hydrogen (attached to the nitrogen) three peptides later. This helps hold the helix together. Also the radius allows attractive van der Waal potentials across the helix. A left-handed $\alpha$-helix has a higher energy than a right-handed helix because the $C_\beta$ (first atom in sidechain) bumps into the next helical turn. The bump comes from the asymmetry of the atoms bonded to the $C_\alpha$ (refer to Section 1). No one has observed a natural, left-handed helix longer than three residues.

Figure 2.5 shows two schematics of an α-helix. The left figure shows peptide bonds (solid lines) and hydrogen bonds (dotted lines). The right figure shows the general path of the peptide with less detail.

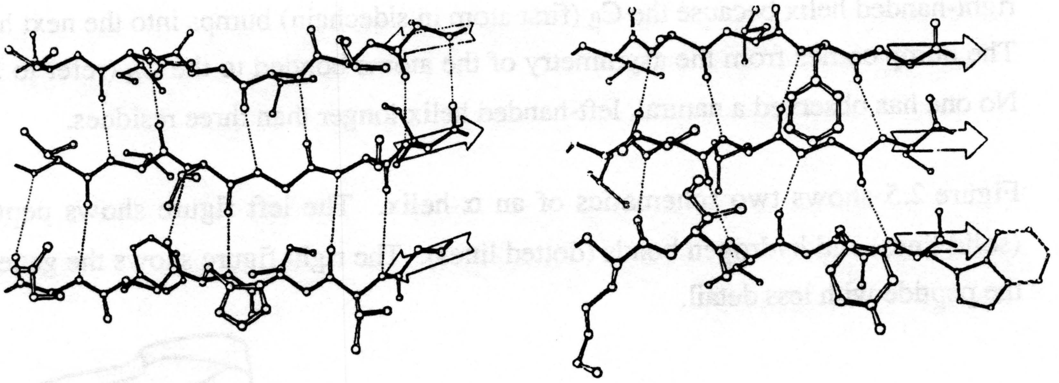


(a) [Schulz 1979] (b) [Richardson 1992]

**Figure 2.5: Two drawings of an α-helix. (a) Solid and dashed lines denote covalent and hydrogen bonds, respectively. (b) Ribbon follows path of backbone in the Felix protein.**

## 3.2. β-sheet

The second most prevalent structure found in proteins is the β-sheet. On average fifteen percent of a protein structure is β-sheet. Sheets contain extended, nearly-straight strands of peptides beside one another. Hydrogen bonds hold the strands together. Figure 2.6 shows two types of sheets, parallel and anti-parallel, whose names refer to the relative direction of adjacent strands.

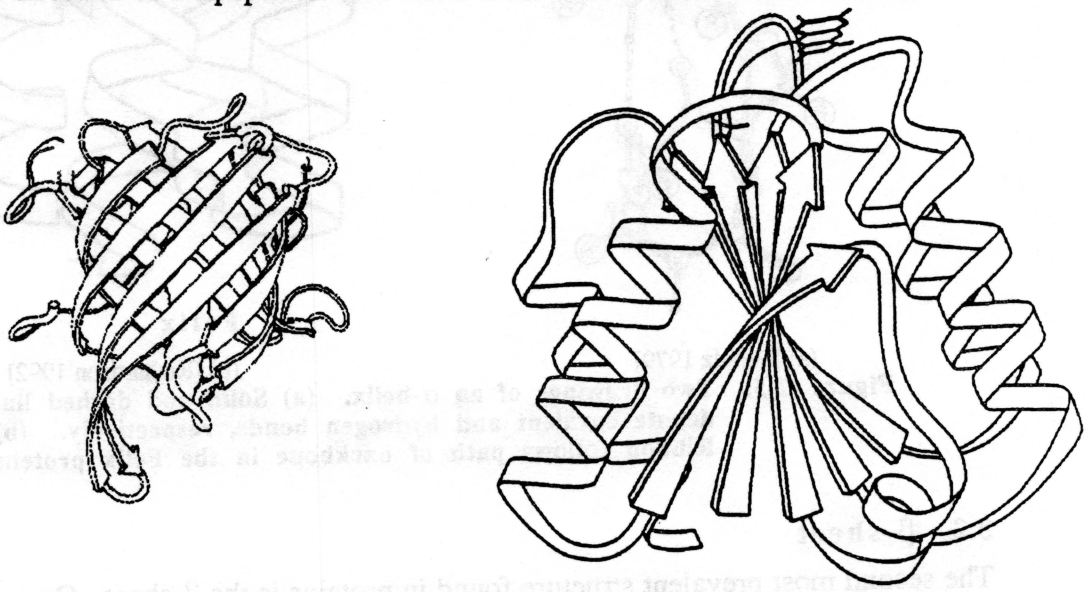


(a) [Fasman 1989]

(b) [Fasman 1989]

**Figure 2.6: Two drawings of a β-sheet. Strands are anti-parallel in (a) and parallel in (b).**

Usually sheets have an overall left-handed twist around an axis in the sheet perpendicular to the strands. Figure 2.7 shows the geometry of two sheets. The arrows show the sheet and the direction of the peptides. The small tubes show residues not in the sheet.



(a) [Fasman 1989]

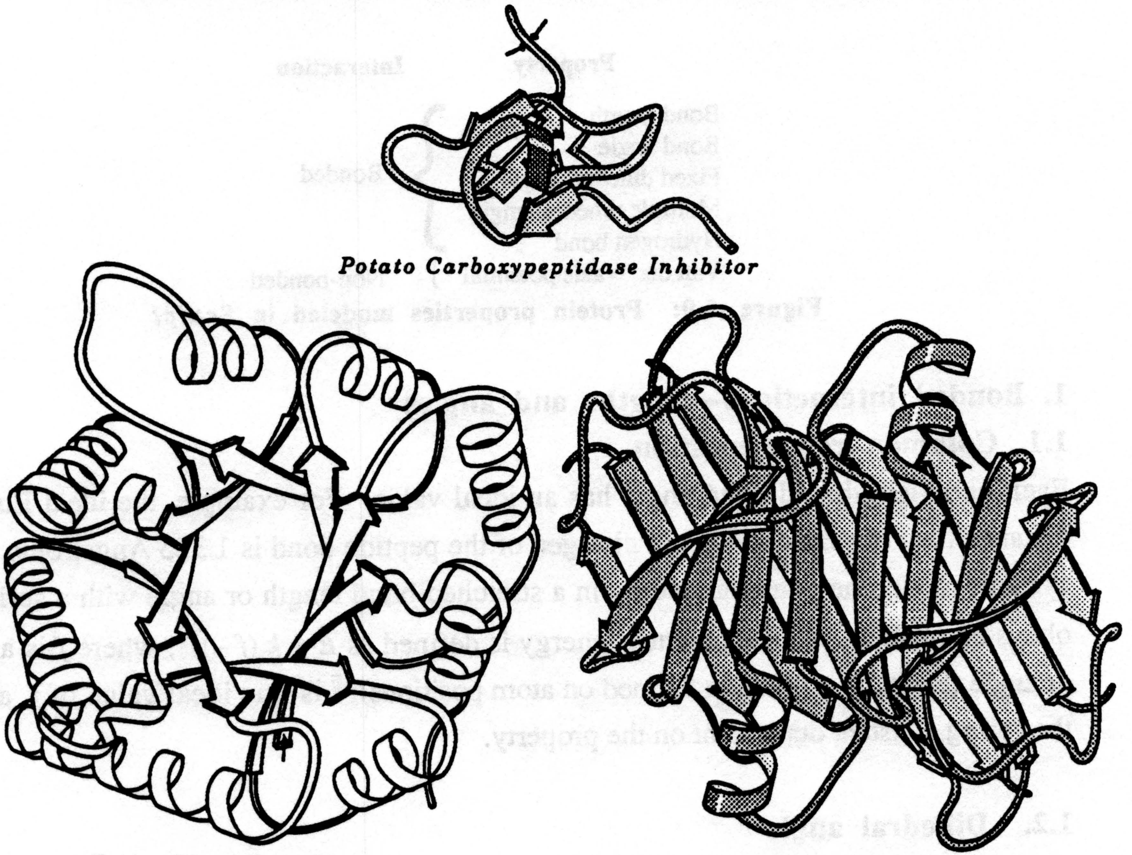
(b) [Richardson 1992]

**Figure 2.7: Anti-parallel (a) and parallel (b) sheets denoted with a ribbon that represents the direction of the backbone.**

### 3.3. Globular proteins

All the properties just discussed play crucial roles in determining a protein's three-dimensional structure. The van der Waals and Coulomb potential energies pull non-bonded atoms together yet restrict their minimum separation. Hydrogen bonds help hold peptides together in helices and sheets.

Figure 2.8 shows three globular proteins. The figure emphasizes helices with tubular spirals and sheets with broad arrows. The number of non-hydrogen atoms in the proteins of Figure 2.8 are approximately 400 (top), 3000 (lower left), and 2500 (lower right).



*Potato Carboxypeptidase Inhibitor*

*Triose Phosphate Isomerase*

*Prealbumin Dimer*

[Richardson 1992]

**Figure 2.8: Illustrations of three globular proteins.**

### Part II – The *Sculpt* Model

This part presents the mathematical model of each property used in *Sculpt.* Figure 2.9 lists the protein properties that *Sculpt* models. The first five are bonded interactions, and the last one is a non-bonded interaction. The bonded and non-bonded interactions are first discussed separately and then combined into a general mathematical model.

| **Property** | **Interaction** |
|---|---|
| Bond length | Bonded |
| Bond angle | |
| Fixed dihedral angle | |
| Multiple dihedral angle | |
| Hydrogen bond | |
| Van der Waals potential | Non-bonded |

**Figure 2.9: Protein properties modeled in *Sculpt***

### 1. Bonded interactions—lengths and angles

#### 1.1. Common representation

Each bond length and bond angle has an ideal value. For example, the ideal Euclidian distance between the carbon and nitrogen of the peptide bond is 1.325 Ångstroms. Many systems model the potential energy in a stretched bond length or angle with a spring that obeys Hooke's Law. The potential energy is defined as $E = k\,(f - \bar{f})^2$, where $f$ is a length or angle (typically a function defined on atom positions), $\bar{f}$ is the ideal value of $f$, and $k$ is the spring constant dependent on the property.

#### 1.2. Dihedral angle

A *dihedral* angle is the angle between one vector and a plane defined by two other vectors. Part A in Figure 2.10 shows a view of a dihedral angle. *Sculpt*, and most systems, represent planarity and local steric hindrance with dihedral angles. Two examples illustrate uses of dihedral angles. First, three dihedral angles specify the planarity of the peptide atoms (Part B). Second, a dihedral angle specifies which angles are preferred for a hydrogen after steric hindrance reduces the possible values (Part C).

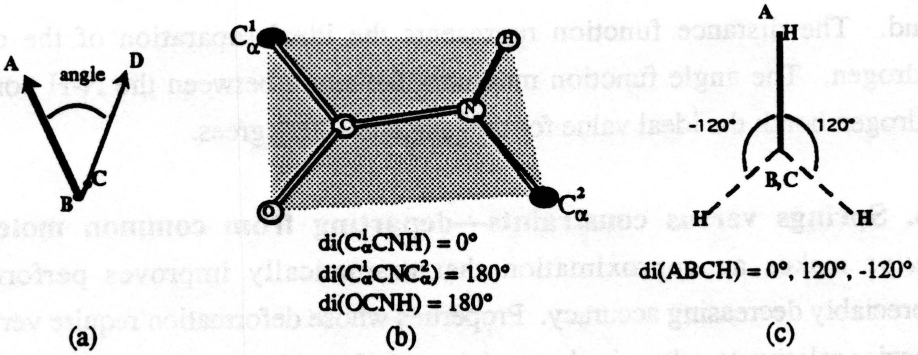
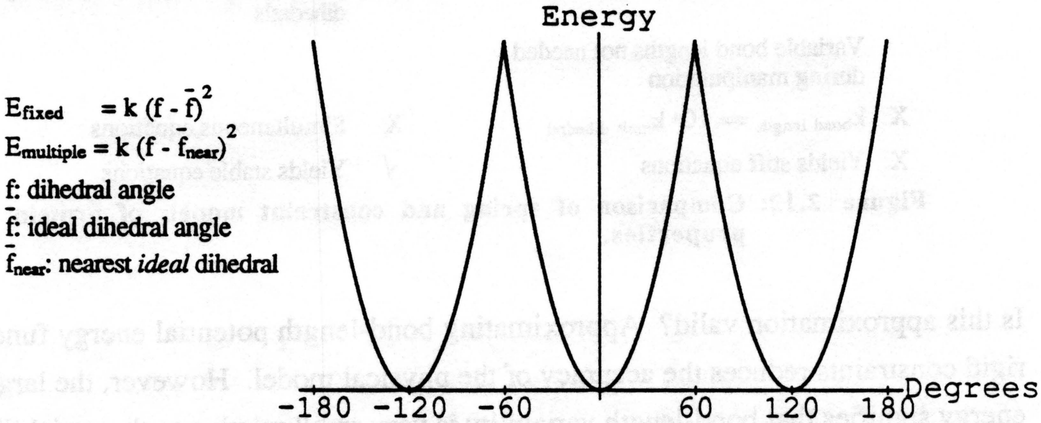

A
angle
D
B
C
$C^1_\alpha$
H
C
N
O
$C^2_\alpha$
$di(C^1_\alpha CNH) = 0°$
$di(C^1_\alpha CNC^2_\alpha) = 180°$
$di(OCNH) = 180°$
A
H
-120°
120°
B,C
H
H
di(ABCH) = 0°, 120°, -120°
(a)
(b)
(c)

**Figure 2.10: (a) Illustration of a dihedral angle. (b) Dihedral angles that define the planarity of a peptide. (c) Three equally-favorable dihedral angles.**

Notice Part C has three equally-favorable dihedral angles. A *multiple* dihedral angle expresses this. A multiple dihedral is specified with two parameters, the multiplicity of the ideal angle and the first (reference) ideal angle. Figure 2.11 shows the energy functions associated with multiple and fixed (single) dihedral angles. The graph shows three equally-spaced energy wells for a dihedral with multiplicity three and starting angle zero (the same dihedral given in Part C above). Dihedral angles are only used to model steric hindrance that restricts the dihedral angle among four atoms to one value or multiple, equally-spaced values; otherwise, a van der Waals interaction is used.



$E_{fixed} = k\,(f - \bar{f})^2$

$E_{multiple} = k\,(f - \bar{f}_{near})^2$

f: dihedral angle

$\bar{f}$: ideal dihedral angle

$\bar{f}_{near}$: nearest *ideal* dihedral

**Figure 2.11: Energy function of a dihedral angle with multiplicity three.**

## 1.3. Fixed hydrogen bonds

Hydrogen bonds may be manually inserted into the model rather than forming and breaking them as atoms move. This helps hold a secondary structure together during the sculpting process. *Sculpt* models a hydrogen bond with a distance and an angle function. Consider a hydrogen bond formed between the oxygen of a C-O bond and the hydrogen of a N-H bond. The distance function represents the ideal separation of the oxygen and the hydrogen. The angle function measures the angle between the N-H bond and the H-O hydrogen bond; the ideal value for this angle is 180 degrees.

### 1.4. Springs versus constraints—departing from common molecular models

*Sculpt* makes an approximation that dramatically improves performance without appreciably decreasing accuracy. Properties whose deformation require very large potential energies relative to others in the model are replaced by rigid constraints. Bond lengths are modeled as constraints since the potential energy increase for changing a bond length is five orders-of-magnitude larger than the energy increase associated with a comparable change in distance between two attractive atoms. Minimizing functions with similar potential energies that are subject to constraints requires less computation, in this application, than minimizing all the energies without any constraints. This approximation provides interactive performance with more complicated models. Figure 2.12 lists some pros and cons for representing bonded properties with springs or constraints.

| | **Spring: $k(f - \bar{f})^2$** | | **Constraint: $f - \bar{f} = 0$** |
|---|---|---|---|
| √ | Allows plasticity | | Brass/mechanical model |
| √ | Allows multiple dihedrals | X | Prevents moving among multiple dihedrals |
| | Variable bond lengths not needed during manipulation | | |
| X | $k_{\text{bond length}} == 10^4\, k_{\text{mult dihedral}}$ | X | Simultaneous equations |
| X | Yields stiff equations | √ | Yields stable equations |

**Figure 2.12: Comparison of spring and constraint models of protein properties.**

Is this approximation valid? Approximating bond-length potential energy functions with rigid constraints reduces the accuracy of the physical model. However, the large potential energy signifies that bond-length variability is very small relative to the variability of other properties. Since the bond lengths hardly change, constraining them for increased performance is justified. *Sculpt* lets a chemist trade performance for accuracy when desired, by modeling lengths with potential energy functions.

| **Property** | **Model** |
|---|---|
| **Bond length** | **Constraint** |
| **Bond angle** | |
| **Fixed dihedral angle** | |
| **Multiple dihedral angle** | **Spring** |
| **Hydrogen bond** | |

**Figure 2.13: Default division of properties in *Sculpt*.**

At run time a chemist specifies which properties are constrained and which vary. Figure 2.13 shows the default division of the bonded properties in *Sculpt*. Variable dihedral angles and hydrogen bonds use springs since they are not as stiff.

## 2. Non-bonded interactions—van der Waals and electrostatic

### 2.1. Neighborhoods

For each atom *Sculpt* maintains a (neighbor) list of the other atoms within a fixed radius. The van der Waals potential, electrostatic charge, and hydrogen bond properties are only applied to atoms in one's neighbor list. The non-bonded energy between two atoms approaches zero as the separation increases. Since the potential energy in each property approaches zero at a different rate, *Sculpt* needs multiple neighbor lists with various radii (e.g. four Ångstroms for hydrogen bonds, six Ångstroms for van der Waals potential, and ten Ångstroms for electrostatic interaction).

Many nearby atoms are removed from the neighbor lists. Since covalent bonds keep atoms closer than the van der Waals potential allows, all atoms reachable via one, two, or three bonds from a given atom are not in its neighbor list. The neighbor-list technique is used in most molecular modeling systems.

### 2.2. Van der Waals potential and the 4–8 approximation

The van der Waals potential energy models repulsion of intersecting electron shells and attraction of induced dipoles from orbiting electrons. The Lennard-Jones model combines the repulsion and attraction with $E = E_m (- \frac{R_m^{12}}{r^{12}} + 2 \frac{R_m^6}{r^6})$ where $r$ is the separation, $E_m$ is the energy minimum, and $R_m$ is the separation at the energy minimum. The gray line in Figure 2.14 shows a graph of this function for a particular $E_m$ and $R_m$. Two aspects of this function cause computational difficulties: the extremely sharp increase in energy when the separation is less than $R_m$, and the infinite extent of the attractive distance. The minimizer discussed in Chapter 3 moves atom positions a distance proportional to the magnitude of the energy. Thus a large energy occurring when two atoms are more than, say, ten percent closer than $R_m$ results in large atom movements. Since the distance for attraction extends to infinity, the Lennard-Jones model would require that *Sculpt* calculate the pair-wise attraction between all atoms in a molecule.

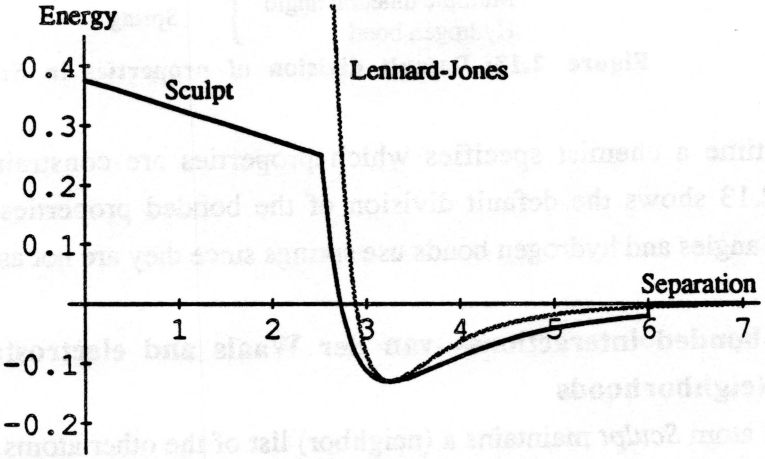


**Figure 2.14: Plot of *Sculpt* and Lennard-Jones model of attraction and repulsion between two atoms.**

*Sculpt* uses a similar model that allows faster update rates but is slightly less accurate. Figure 2.14 illustrates the three differences between the two models. First, the separation distance of attraction is limited. The discussion of neighbor lists justifies this. Second, the energy well is wider because the *Sculpt* model replaces the 6 and 12 exponents in the Lennard-Jones model with 4 and 8. The slower ascent when the separation is less than $R_m$ allows faster solution of the equations. However, the wider energy well allows smaller separation and stronger attraction between atoms. Third, the repulsive energy increases linearly after a given threshold. The positive energy still repels atoms but avoids large atom movements that arise when the minimizer encounters large repulsive energies. A chemist can invoke the Lennard-Jones model when accuracy is more important than speed.

## 2.3. Electrostatic charge

*Sculpt* does not currently model the potential energy between charged atoms. Calculating this energy with Coulomb's Law fits into the *Sculpt* model. However, electrostatic energy falls off so slowly with distance that computing it destroys interactive performance on current machines. The goal of the system is to provide a means for sculpting proteins while interactively modeling as many basic properties as possible. Modeling electrostatic interaction is not so crucial as modeling local electron shell interactions and hydrogen bonds. This application is different from docking (bonding) two molecules where electrostatic interaction plays a crucial role in the outcome. Electrostatic interaction will be added when faster computers arrive. For now a chemist can recognize important, local interactions and use tools provided in *Sculpt* to pull them together or apart.

### 3. Implicit hydrogens

Many hydrogen atoms are not used explicitly in the calculations. Instead atoms bonded to hydrogens are modified to model them implicitly. Roughly half the atoms in a globular protein are hydrogens. About a quarter of the hydrogens can form hydrogen bonds, the rest just get in the way (sterically speaking). *Sculpt*, and most molecular modeling systems (e.g. [Biosym 1991; Polygen 1991; Tripos 1988]), only explicitly model hydrogens that can form hydrogen bonds (the hydrogen in the backbone, for example). The rest are implicitly modeled by appropriately increasing the radius of each atom bonded to a hydrogen.

### 4. Pluggable model

*Sculpt* reads the ideal values, spring constants, van der Waals coefficients, and bond topology from an input file (described in Section 3 of Chapter 8). None of the coefficients in the *Sculpt* model are hard-wired into the system. This lets a chemist use different modeling coefficients. The values used in this research come from Jan Hermans' molecular dynamics system, *Cedar* [Hermans 1989], which is based on the *CHARMM* model of proteins [Brooks 1983].

### 5. Solvent

*Sculpt* does not model solvent. No existing computational model of solvent runs interactively on current computers. Without the entropic force exerted by surrounding solution, proteins slowly drift apart. *Sculpt* lets a chemist pull structures together by attaching springs (discussed in Chapter 5). A central gravity that attracts atoms inward is another possible approximation to the properties of solution. *Sculpt*, at present, does not implement this approach.

## 6. Summary

Figure 2.15 lists the protein properties *Sculpt* models. The figure shows the specification and model of each.

| Specification | Protein Property | Mathematical Model |
|---|---|---|
| Input file | Bond length<br>Bond angle<br>Fixed dihedral angle | Constrained to ideal value |
| | Variable dihedral angle<br>Hydrogen bond | Spring energy from nearest ideal value |
| Neighbor list | Van der Waals potential | 4–8 Lennard-Jones potential |
| | Electrostatic charge | Coulomb potential (not implemented) |
| | Solvent interaction | Not treated |

**Figure 2.15: Summary of specification and model of protein properties in *Sculpt*.**

In conclusion I present some mathematical notation that casts the *Sculpt* model into a general framework. Chapter 3 discusses the solution of this model.

First, assume the cartesian coordinates of the atoms are represented in a column vector $\mathbf{x} = [x_1\, x_2 \cdots x_n]^T$, where $T$ is the transpose operator. Second, separate the constraint functions from the energy functions. Let *Energy(x)* equal the sum of all the energies in the protein including potential energies in springs representing multiple dihedral angles and hydrogen bonds, and potential energies from van der Waals and electrostatic interactions. Let ***Constraint(x)*** represent a column vector of the constraint functions (**bold** face denotes a vector). Each constraint—bond length, bond angle, and fixed dihedral angle—gets one row in the vector. For example, if a protein contains only two constraints—the distance between atoms one and two must equal two, and the angle formed by atoms one, two, and three must equal one—then ***Constraint(x)*** equals the following:

$$\mathbf{Constraint(x)} = \begin{bmatrix} \text{distance}(\text{atom}_1, \text{atom}_2) - 2 \\ \text{angle}(\text{atom}_1, \text{atom}_2, \text{atom}_3) - 1 \end{bmatrix} = \begin{bmatrix} c_1(\mathbf{x}) - \bar{c}_1 \\ c_2(\mathbf{x}) - \bar{c}_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

In general if a protein contains $m$ functions (denoted $c_i(x)$, where $i = 1..m$) constrained to $m$ ideal values ($\bar{c}_i$), the vector of constraints is the following:

$$\mathbf{Constraint(x)} = \begin{bmatrix} c_1(\mathbf{x}) - \bar{c}_1 \\ c_2(\mathbf{x}) - \bar{c}_2 \\ \vdots \\ c_m(\mathbf{x}) - \bar{c}_m \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

This notation shows the protein model contains a total energy, *Energy(x)*, subject to a set of constraints, ***Constraint(x)***. *Sculpt* continuously adjusts the atom positions to maintain a local energy minimum that simultaneously satisfies the constraints. This problem is equivalent to finding a local solution to

$$\begin{array}{l}\text{minimize Energy}(\mathbf{x}) \\ \text{such that } \mathbf{Constraint}(\mathbf{x}) = 0\end{array}$$

The next chapter discusses how *Sculpt* solves this problem.

# Chapter 3
# Mathematical Background

This chapter discusses the mathematical methods selected for finding a constrained minimum. The general problem addresses how to find a minimum of a real-value function, ***$e(x)$***, such that a vector of equality constraint functions is satisfied, ***$c(x) = 0$***. The last section of Chapter 2 casts the protein sculpting problem into an energy function and a set of constraints.

This chapter contains four parts. The first part defines terms in constrained optimization and introduces goals for the *Sculpt* algorithm. The second part discusses two algorithms for finding an unconstrained minimum. Constrained minimization algorithms contain steps that use these techniques. The third part describes classes of constrained minimization problems and positions the sculpting problem within these. This part also presents two general-purpose algorithms for finding a constrained minimum in this class. The fourth part discusses the *Sculpt* algorithm for finding a constrained minimum. The algorithm is an adaptation of others in the chapter to the interactive minimization problem. The fourth part also presents the central decisions that led to this algorithm.

The core material in this chapter comes from one book: *Practical Methods of Optimization,* by Fletcher [Fletcher 1987]. Two other books contain useful explanations and derivations for readers interested in exploring these techniques in greater depth. *Optimization Theory,* by Hestenes [Hestenes 1975], concentrates on theoretical issues and illustrates them with numerous figures. *Practical Optimization,* by Gill et al. [Gill 1981], is a cookbook of techniques every implementor should read.

Prose is usually chosen over symbols, but clarity and precision require some use of symbols. Figure 3.1 summarizes the symbols and notation found in the chapter. Column vectors are used throughout. Vectors, in lower case, and matrices, in upper case, are shown with bold type. A subscript denotes an element in a vector, and a superscript denotes an iteration within an algorithm. For example, $a_i$ is the $i^{th}$ element of vector a, and $a^i$ is the value of a at the $i^{th}$ iteration. The superscript $T$ denotes transposition so that $a^T$ is a row vector. The inner product of two vectors $\mathbf{a}^T$ and $\mathbf{z}$ is $\mathbf{a}^T\mathbf{z} = \Sigma_i a_i z_i$. The square root of the inner product of a vector with itself ($\sqrt{\mathbf{a}^T\mathbf{a}} = \sqrt{\Sigma_i\, a_i^2}$) is the Euclidean distance, or $l_2$ norm, denoted $\|\mathbf{a}\|$.

| | | |
|---|---|---|
| Vector notation: | $\mathbf{a}$ | column vector |
| | $a_i$ | $i^{th}$ element of vector a |
| | $a^i$ | value at $i^{th}$ iteration |
| | $a^T$ | transpose |
| | $a^Tz$ | inner product; $\Sigma_i a_i z_i$ |
| | $\|a\|$ | length of a; $\sqrt{a^Ta}$ |
| Differential operators: | $\frac{\partial f}{\partial a_i}$ | partial derivative of *f* with respect to $a_i$ |
| | $\nabla f(a)$ | gradient vector; $\begin{bmatrix} \frac{\partial f}{\partial a_1} \\ \vdots \\ \frac{\partial f}{\partial a_n} \end{bmatrix}$ |
| | $\nabla^2 f(a)$ | Hessian matrix; $\begin{bmatrix} \frac{\partial^2 f}{\partial a_1 \partial a_1} & \cdots & \frac{\partial^2 f}{\partial a_1 \partial a_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial a_n \partial a_1} & \cdots & \frac{\partial^2 f}{\partial a_n \partial a_n} \end{bmatrix}$ |
| Reserved names: | n | number of variables |
| | m | number of constraints |
| | $\Re^k$ | k-dimensional space of real numbers |
| | x | variables in minimization problem; $x \varepsilon \Re^n$ |
| | $\lambda$ | vector of Lagrange multipliers; $\lambda \varepsilon \Re^m$ |
| | e(x) | energy function; $e:\Re^n \rightarrow \Re$ |
| | $c_i(x)$ | $i^{th}$ constraint function; $c_i:\Re^n \rightarrow \Re$ |
| | c(x) | column vector of constraint functions; $c:\Re^n \rightarrow \Re^m$ |
| | $x^*, \lambda^*$ | value at local minimizer |
| | $\nabla c(x)$ | Jacobian of constraints; $\begin{bmatrix} \frac{\partial c_1}{\partial x_1} & \cdots & \frac{\partial c_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial c_1}{\partial x_n} & \cdots & \frac{\partial c_m}{\partial x_n} \end{bmatrix}$ |

**Figure 3.1: Summary of the notation contained in the chapter.**

This chapter reserves the letters *e* and *c* in mathematical notation to represent the *e*nergy function and a *c*onstrained function. Throughout this chapter assume there are *n* variables,

$\boldsymbol{x} \in \Re^n$, and $m$ constraints. The real-valued function *Sculpt* minimizes, $e(\boldsymbol{x})$, is defined from $\Re^n$ to $\Re$. An individual constraint is in plain type with a subscript, for example $c_i(\boldsymbol{x})$, and the column vector of all constraints is the bold face $\boldsymbol{c}(\boldsymbol{x})$. The vector of first partial derivatives of a constraint with respect to each variable (its gradient) is denoted by the column vector $\nabla c_i(\boldsymbol{x})$. Often the first derivatives of all the constraints are collected into columns of the n × m *Jacobian* matrix $\nabla \boldsymbol{c}(\boldsymbol{x})$.

## 1. Introduction to optimization

Optimization theory concentrates on finding the best, or optimal, configuration of a system that has a fixed set of variables and a means of comparing different configurations. The sculpting problem uses optimization theory to find an arrangement of atoms that satisfies a set of constraint functions and minimizes a potential energy function.

This research seeks a *local* minimum and not a *global* minimum. A point $\boldsymbol{x}^*$ is a local minimum if the function value is lower than the value at all neighboring points (i.e. $e(\boldsymbol{x}^*) < e(\boldsymbol{x})$ for all $\boldsymbol{x}$ near $\boldsymbol{x}^*$). If the function value is lower than the value at all possible points, then $\boldsymbol{x}^*$ is called a global minimum (i.e. $e(\boldsymbol{x}^*) < e(\boldsymbol{x})$ for all $\boldsymbol{x} \neq \boldsymbol{x}^*$). Figure 3.2 illustrates the difference between global and local minima. Finding the global minimum of an arbitrary function is HARD. Methods exist, such as simulated annealing [Kirkpatrick 1983], that find a global minimum, but may require enormous computation time. Maintaining a local, rather than a global, minimum during the sculpting process lets a chemist explore different energy configurations en route, hopefully, to the global minimum.

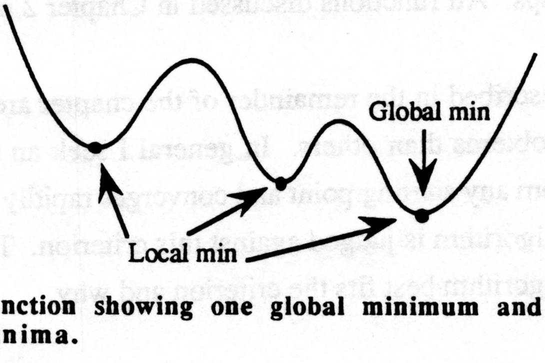


**Figure 3.2: Function showing one global minimum and three local minima.**

A *constrained minimum* is a minimum that additionally satisfies one or more constraint functions. Optimization theory studies both constrained equality and inequality functions, though this research only uses equality constraints. Figure 3.3 shows a graph of a two-dimensional energy function and one constraint. The constraint requires that the variables

$x$ lie on the line $c(x) = 0$. The dashed lines show contours of the energy function $e(x)$ on which the value is constant.

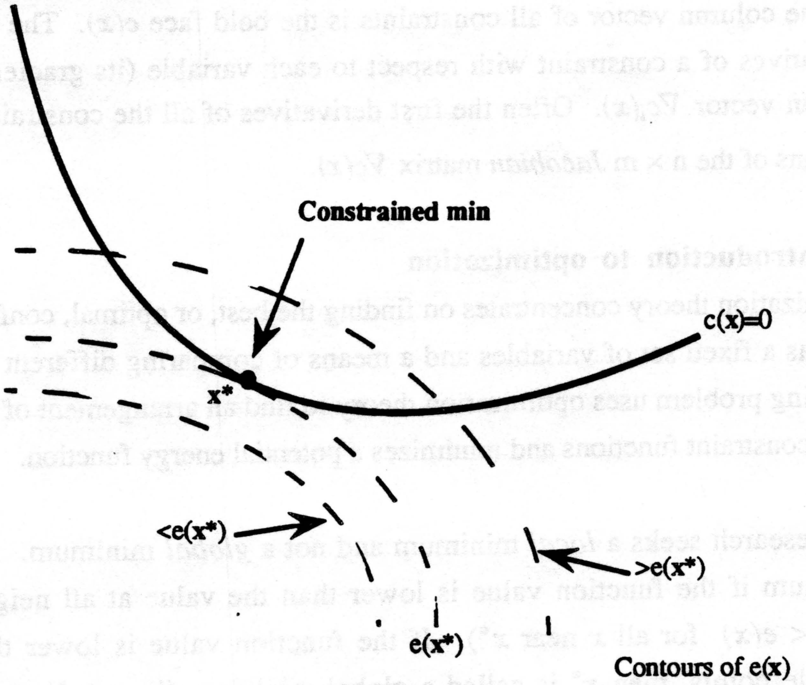


**Figure 3.3: A constrained minimum. The dashed lines show contours of constant energy and the solid line shows where the constraint function is satisfied.**

This research places two conditions on all the energy and constraint functions: the first and second partial derivatives *exist* and are *continuous*. Most optimization theorems place these conditions to prevent difficulties arising from disjoint energy functions and energy functions with cusps. All functions discussed in Chapter 2 meet these requirements.

The algorithms described in the remainder of the chapter are general purpose but perform better on some problems than others. In general I seek an algorithm that converges to a local minimum from any starting point and converges rapidly in the neighborhood of a local minimum. Each algorithm is judged against this criterion. The final section in the chapter discusses which algorithm best fits the criterion and why.

## 2. Unconstrained minimization

This part examines algorithms that find an unconstrained local minimum. What are the necessary and sufficient conditions for a point $x^*$ to be a local minimum of a function $e(x)$? The slope must equal zero and the curvature must be non-negative. That is equivalent to requiring the gradient at $x^*$ equal zero, $\nabla e(x^*)=0$, and, for any nonzero vector $s$,

$s^T \nabla^2 e(x^*) s \geq 0$. A sufficient condition for an *isolated* local minimum at $x^*$ requires strictly positive curvature. The necessary and sufficient conditions are used throughout the following algorithms.

No known analytic solution exists that finds a minimum of an arbitrary polynomial with degree greater than four. Two methods that *iteratively* find a local minimum of an unconstrained function are presented next. Both methods follow the algorithm listed in Figure 3.4, differing only in the determination of the direction in step 2.

```
0. Given starting point $x^0$; $k \leftarrow 0$.
1. while not StoppingCriterion($x^k$)
       2. Calculate direction $d$
       3. Find step length $\alpha$ that $\min_{\alpha \in [0..1]} e(x^k + \alpha d)$
       4. $x^{k+1} \leftarrow x^k + \alpha d$; $k \leftarrow k + 1$
```

**Figure 3.4: Iterative algorithm for finding an unconstrained minimum.**

StoppingCriterion() is typically a function of $x$ and/or the energy derivative that determines when the current value of $x$ is close enough to the minimum. One criterion for stopping is when the first derivative equals zero. Such a position is called a *stationary* point. A stationary point occurs at a local minimum, maximum, or saddle. Usually StoppingCriterion() incorporates other information that ensures the point is a minimum. Step 2 determines a direction vector, $d$, in which to move. Step 3 minimizes the function over one scalar variable. The minimization searches along the line from $x^k$ in the direction $d$ for a minimum function value. The search limits $\alpha$ to lie between zero and one. The following sections present two algorithms that determine the direction and discuss the line search in greater detail.

## 2.1. Steepest descent

The simplest choice of the direction in Step 2 is the direction of steepest descent. At a given point, $x^k$, the direction in which the function decreases the fastest is the negative gradient evaluated at that point (*i.e.* $d = -\nabla e(x^k)$). An infinitesimal movement in the direction of the negative gradient is guaranteed to give the fastest decrease in the function's value.

In practice we want algorithms that make larger movements than infinitesimally small. The $\alpha$ in Step 3 states how far to move in the direction. The function at some points along the direction vector can have larger values than at the current point since the function is nonlinear. The line search over $\alpha$ ensures that the function at the new point has a smaller value than the current point. A common algorithm for the line search is the *bisection method* which starts with $\alpha$ set to one and then repeatedly halves $\alpha$ until the function value is less than the initial value.

The steepest descent algorithm converges to a local minimum from any starting point. The algorithm evaluates the first partial derivative of the function at each step. The algorithm is easy to implement and useful to employ when far from the minimum. Some algorithms, however, converge to a minimum with fewer iterations than the steepest descent.

## 2.2. Newton's method

Newton's method repeatedly approximates the function with a quadratic function and minimizes the quadratic function. Minimizing a quadratic function takes one iteration since there is an analytic solution. The curvature information in the quadratic approximation provides more information about the energy function than the slope information in the steepest descent.

The quadratic approximation comes from truncating the Taylor series expansion of e(x) about $x^k$. First, expand $e(x) = e(x^k + d)$, where $d = x - x^k$, in a Taylor series about the current iterate, $x^k$. Next, let $q(d)$ represent the quadratic approximation of $e(x)$ that results from truncating the series after the second-order term; that is, let $q(d) = e(x^k) + d^T \nabla e(x^k) + \frac{1}{2} d^T \nabla^2 e(x^k) d$. Recall that the slope of a function at a minimum point equals zero. For $q(d)$ to be a minimum, $\nabla q(d)$ must equal $0$. Taking the gradient of the quadratic approximation with respect to $d$ gives $\nabla q(d) = 0 + \nabla e(x^k) + \nabla^2 e(x^k) d$ which must equal zero. Rearranging the equation shows the minimum occurs when $d$ satisfies $\nabla^2 e(x^k) d = - \nabla e(x^k)$. This $d$ is the direction of movement in Step 2 of the unconstrained minimization.

The second derivative of the energy requires calculating the Hessian matrix, the matrix of second partial derivatives. Determining $d$ then requires solving the $n \times n$ system of equations, where $n$ is the number of variables. Since the equations only have a unique solution when the matrix is invertible, Newton's method is applicable only under certain circumstances. The major advantage of Newton's method is that the rate of convergence is quadratic when the matrix is invertible. Hybrid, or quasi-Newton, methods exist that try balancing the guaranteed convergence of the steepest descent and the quadratic convergence in Newton's method. A discussion of hybrid methods is found in *Introduction to Linear and Nonlinear Programming* [Luenberger 1973].

## 3. Constrained optimization

This part describes two algorithms that find a local solution of

minimize e(x)
such that $c_i(x) = 0$ (i = 1..m), or equivalently, $c(x) = 0$.

Minimizing an arbitrary function subject to arbitrary constraints is an unsolved research problem. Different algorithms exist for solving constrained optimization problems that can be classified by specific properties. Some of these classes include the following: the energy function is linear, quadratic, or nonlinear; the constraints are linear or nonlinear; the constraints are equality and/or inequality; the functions are smooth and continuous in zero or more derivatives; the initial point does or does not satisfy the constraints. The class of problems this research addresses possess the following properties:

1. nonlinear energy and constraint functions;
2. continuous energy and constraint functions through the second derivative;
3. only equality constraint functions;
4. initial point satisfies or nearly satisfies the constraints.

There is no general agreement on how this class of problems should be solved in practice. The remainder of this chapter presents two algorithms that theoretically can solve this problem. One algorithm, the penalty method, recasts constraints as energy functions that rapidly increase value as the constraints are violated. The second algorithm, the Lagrangian method, minimizes a new function that is the sum of the original energy and the inner product of the constraints and some additional variables. This research uses an adaptation of the Lagrangian method.

### 3.1. Penalty method

Interactively solving a nonlinear constrained minimization problem balances reducing the energy function and maintaining, or nearly maintaining, the constraints. A penalty method penalizes (increases) the energy function when constraints are violated. A penalty method minimizes a new function that is the sum of the original energy function and the penalized constraints (equivalently, this models constraints with springs). Consider the function $\phi(x) = e(x) + \frac{1}{2}pc(x)^Tc(x)$, where $p$ is a scalar penalty (spring stiffness) applied to each constraint. When the constraints are satisfied this function reduces to the energy function. As $p$ approaches infinity the solution of minimize $\phi(x)$ converges to the solution of the original constrained minimization problem, but as $p$ approaches zero constraint violations are ignored.

The constrained problem now requires choosing a penalty and minimizing $\phi(x)$. Steepest descent and Newton's method follow a multiple of the gradient of the function at each step. The gradient of $\phi(x)$ is $\nabla\phi(x) = \nabla e(x) + p\nabla c(x)c(x)$. Notice as $p$ approaches infinity, the magnitude of the gradient approaches infinity even though a constraint may only be slightly violated. The range in magnitude is hard to model accurately in a computer due to limited precision. As the penalty increases the step length, $\alpha$, must decrease, and thus more iterations are needed to converge. The same problem arises when modeling springs that are arbitrarily stiff.

The technique of penalizing errors is also used in conjunction with other techniques. For example, the Lagrangian method provides a theoretical approach to constrained optimization problems but, in practice, is augmented with a penalty term to increase convergence.

### 3.2. Lagrangian method

The Lagrangian method is derived from the first-order conditions of a locally constrained minimum $x^*$. The first-order necessary condition of an unconstrained problem requires the gradient of the function at the minimum is zero. The constrained problem has two conditions: the constraints are satisfied, $c(x^*) = 0$; and the gradient of the energy function is a linear combination of the gradient of the constraint functions, $\nabla e(x^*) = \Sigma_i \nabla c_i(x^*)\lambda_i^* = \nabla c(x^*)\lambda^*$. There is one scalar $\lambda_i$, called a *Lagrange multiplier*, for each constraint. The vector $\lambda^*$ denotes the value of the multipliers at the minimum, $x^*$. Figure 3.5 illustrates these conditions. At the point $x'$ the gradient of the energy is not a linear combination of the constraint gradients. There exists a step, $d$, in the direction, $-\mu$, that reduces the energy and maintains the constraints. At the local minimum, $x^*$, the conditions are satisfied and no descent direction remains that satisfies the constraints. For a problem with $n$ variables and $m$ constraints the first-order conditions give $n+m$ nonlinear (in $x$) equations and $n+m$ unknowns.

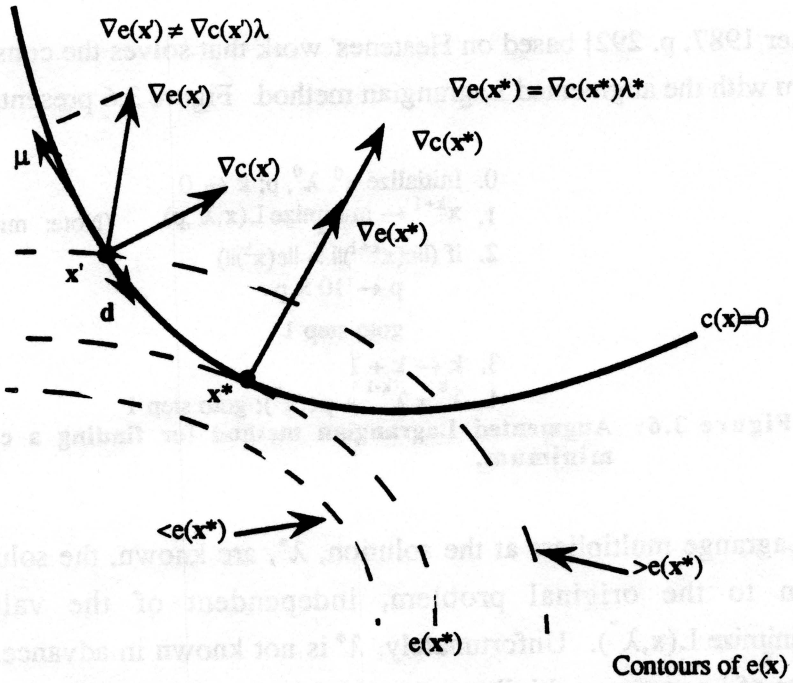


**Figure 3.5: Sufficient conditions of a constrained minimum show the constraint is satisfied and the constraint gradient is a scalar multiple of the energy gradient.**

The necessary conditions are concisely stated by introducing a new function, called the Lagrangian. The *Lagrangian* is defined as $L(x,\lambda) = e(x) - \lambda^T c(x)$. If $(x^*,\lambda^*)$ is a stationary point of the Lagrangian (i.e. $\nabla L(x^*,\lambda^*) = 0$) then it is also a constrained local minimum of the original problem. Notice that when a point $x$ satisfies the constraints, minimizing the Lagrangian is equivalent to minimizing the energy function. The techniques discussed in Section 2 can be used to solve this problem.

Unfortunately, the Lagrangian may not have a minimum even when the original constrained problem has a solution. This is because the solution depends solely on first-order information. This problem in general disappears if convexity is considered. Hestenes describes a method, called the *augmented* Lagrangian method, that adds a penalty term, $pc(x)^T c(x)$, to the Lagrangian [Hestenes 1975, p. 308]. For sufficiently large $p$ (though not infinite!) a point $x$ minimizes the augmented Lagrangian if and only if it is a constrained minimum of the original problem.

In practice the augmented Lagrangian method converges faster than the penalty method because its penalty term is bounded rather than required to approach infinity. In the molecular modeling problem the penalty term is quite small. Fletcher gives an algorithm

[Fletcher 1987, p. 292] based on Hestenes' work that solves the constrained minimization problem with the augmented Lagrangian method. Figure 3.6 presents the steps.

```
0. Initialize $x^0, \lambda^0, p$; $k \leftarrow 0$
1. $x^{k+1} \leftarrow \underset{x}{\text{minimize}}\, L(x,\lambda^k,p)$      (Note: minimize over $x$ only)
2. if $(\|c(x^{k+1})\| > \|c(x^k)\|)$
       $p \leftarrow 10 \times p$
       goto step 1
3. $k \leftarrow k + 1$
4. $\lambda^k = \lambda^{k-1} + pc(x^k)$; goto step 1
```

**Figure 3.6: Augmented Lagrangian method for finding a constrained minimum.**

If the Lagrange multipliers at the solution, $\lambda^*$, are known, the solution to Step 1 is the solution to the original problem, independent of the value of $p$. That is $x^* = \underset{x}{\text{minimize}}\, L(x,\lambda^*)$. Unfortunately, $\lambda^*$ is not known in advance. Step 4 generates a sequence of Lagrange multiplier *estimates* that converges to $\lambda^*$. Each estimate is used to find the next value of $x$. The penalty term pulls the solution towards one that satisfies the constraints. When the constraints are far from satisfied, the penalty term is large and the algorithm is similar to the penalty method. When the constraints are satisfied or nearly satisfied, the penalty term is negligible and the algorithm approximates the Lagrangian method.

*Sculpt* bases its algorithm on the steps just presented. The system estimates the Lagrange multiplier, performs an unconstrained minimization, and if necessary, adjusts the penalty term. The remainder of this chapter presents the approximations made in *Sculpt* to Steps 2–4. A comparison and explanation is made to justify the differences between steps in the preceding algorithm and the *Sculpt* algorithm.

## 4. *Sculpt's* mathematical model

An important feature of the sculpting problem is that the system always maintains a near local minimum. Before any external springs are placed in the model, I assume the constraints are satisfied and the system is in or very near a constrained local minimum. This assumption gives tremendous computational advantages. First, the minimization begins with the constraints satisfied. Second, a new tug only adds potential energy to the model. Third, since the additional energy is small relative to the total energy, a new minimum is usually near the last minimum. Figure 3.7 lists the general steps taken in the *Sculpt* model. Each step is discussed next.

```
0. Initialize $x^0$; $k \leftarrow 0$
1. Estimate the Lagrange multipliers, $\lambda^k$
2. Determine a penalty term, p
3. $x^{k+1} \leftarrow \underset{x}{\text{minimize}}\, L(x,\lambda^k,p)$
```

**Figure 3.7: *Sculpt* algorithm for finding a constrained, local minimum.**

### 4.1. Estimate the Lagrange multipliers

The augmented Lagrangian method (Figure 3.6) gives a method for estimating the Lagrange multiplier that uses the previous estimate and the current value of the constraints; specifically, $\lambda^k = \lambda^{k-1} + pc(x^k)$. The accuracy of the estimate is higher if first-order information about the constraints is included. A more accurate estimate lets the constrained minimizer converge in fewer iterations.

Gill presents a method for estimating the Lagrange multipliers called the *first-order multiplier estimate* [Gill 1981]. This method uses first-order information about the constraints rather than just zero-order. Section 3.2 shows at a local minimum the gradient of the energy function is a linear combination of the gradient of the constraints ($\nabla e(x^*) = \nabla c(x^*)\lambda$). This algorithm finds the Lagrange multipliers that best satisfy this equation at each iteration. Thus Gill's algorithm estimates the ideal Lagrange multipliers at each iteration by solving the following equations for $\lambda^k$: $[\nabla c(x^k)]\lambda^k = \nabla e(x^k)$.

The dimension of the Jacobian matrix of the constraints, $\nabla c(x)$, is $n \times m$, and the Lagrange multiplier vector is $m \times 1$. In this system there are $n$ equations and $m$ unknowns. Since $m$ is typically less than $n$, the system is over-constrained. *Sculpt* finds a least-squares approximation to the system of equations by first multiplying both sides of the equality by the transpose of the matrix and then solving for $\lambda^k$:
$[\nabla c(x^k)^T \nabla c(x^k)]\lambda^k = \nabla c(x^k)^T \nabla e(x^k)$.

Estimating the Lagrange multiplier in each iteration of the *Sculpt* algorithm requires the following steps. Evaluate the first derivative of the constraint and energy functions at the current point. Pre-multiply the gradient of the energy by the transpose of the constraint Jacobian to get an $m \times 1$ vector, denoted $\boldsymbol{b}$. Pre-multiply the Jacobian by its transpose to get an $m \times m$ matrix, denoted $\boldsymbol{A}$. Finally solve the $m \times m$ simultaneous equations, $\boldsymbol{A}\lambda^k = \boldsymbol{b}$, for $\lambda^k$.

The sparsity and structure of the Jacobian lets an algorithm perform the matrix multiply and solve the equations in operations linearly proportional to the number of constraints. Chapter 5 discusses the details of those algorithms and analyzes their complexity. In practice the multipliers attained through this approach let the entire algorithm converge in one or two iterations.

### 4.2. Determine a penalty term

The penalty term, $p$, pulls the solution towards one that satisfies the constraints. The previous step estimates the Lagrange multipliers using a first-order approximation of the constraints. Since the constraints are nonlinear, this approximation lets the solution drift from the constraints. The penalty term keeps this from moving beyond some limit. The penalty term is set to the error in the least-squares approximation: $p \leftarrow \| \nabla e(x) - \nabla c(x) \lambda \|$.

### 4.3. Find a minimum of L(x,λ,p)

This step finds an $x$ that minimizes the augmented Lagrangian using the multipliers and penalty term of the previous steps. That is, the step finds an $x$ that minimizes $\phi(x)$ where $\phi(x) = L(x,\lambda,p) = e(x) - \lambda^T c(x) - \frac{1}{2} p c(x)^T c(x)$. *Sculpt* finds the minimum with the steepest descent method. The direction of descent is $-\nabla\phi(x)$. *Sculpt* restricts the step length to 0.25 Ångstrom (about one-fourth of a bond length). The direction vector is normalized if it is greater than one. The following algorithm lists the steps that find a local minimum of the augmented Lagrangian function, given $\lambda^k$ and $p$.

```
1. $\mathbf{d} \leftarrow -\nabla\phi(\mathbf{x}^k) = -\nabla e(\mathbf{x}^k) + \nabla c(\mathbf{x}^k)\lambda^k + p\nabla c(\mathbf{x}^k) c(\mathbf{x}^k)$
2. if ($\|\mathbf{d}\| > 1$) then $\alpha \leftarrow \frac{1}{\|\mathbf{d}\|}$
   else $\alpha \leftarrow 1$
3. $\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k + \alpha\mathbf{d}$
```

**Figure 3.8: Direction used in the *Sculpt* algorithm.**

This algorithm repeats using the same penalty and Lagrange multipliers until it satisfies a convergence condition. In practice the algorithm converges in one or two iterations.
A different method such as Conjugate Gradient can be used if convergence requires more iterations. Since the steepest descent converges so rapidly in this application, I did not try other methods.

Another method for finding a minimum of the augmented Lagrangian function uses Newton's method. I implemented Newton's method because I assumed the starting point is near a minimum and wanted quadratic convergence. However, several drawbacks of the implementation caused me ultimately to settle for the steepest descent method. First, implementing the second-derivative evaluation requires a large amount of code—the second-derivative code for the angle function was five times the length of the code for the first derivative, resulting in an additional six hundred lines of C++ code. Second, the time to evaluate the second derivatives increases linearly with code length. Third, the method requires solving an $n \times n$ system of equations which consumes additional time. The version of the system that used Newton's method only modeled distance and angle functions. The large increase in code size and computation and the discovery that steepest descent converges quickly led me to chose the steepest descent method rather than Newton's method.

### 4.4. Summary of algorithm

The *Sculpt* algorithm for finding a local minimum of an energy function such that equality constraints are satisfied is based on the augmented Lagrangian method. The original problem is converted into finding a saddle point of the Lagrangian function. That is, convert the problem of finding a solution, $x^*$, of *minimize* $e(x)$ *such that* $c(x) = 0$ into finding the pair $(x^*,\lambda^*)$ that is a saddle point of $L(x,\lambda,p)$. This is done by first estimating the Lagrange multipliers and setting the penalty term. The method then finds an $x$ that minimizes $L(x,\lambda,p)$ by using the steepest descent method.

More details about the implementation are presented in Chapters 5 and 8. Chapter 5 discusses properties of the protein model that yield linear computational complexity with the technique described here. Chapter 9 gives timing results for computing the minimum.

# Chapter 4
# Related Systems

This chapter discusses molecular modeling and computer graphics systems related to *Sculpt*. The first section describes energy models and applications of molecular modeling. The second section discusses dynamics simulations and constrained modeling used in computer graphics. The second section also describes algorithms that let constraints be added to a model and find model configurations that satisfy arbitrary sets of constraints. The systems and algorithms are analyzed for their applicability to the sculpting problem without regard for their intended application.

## 1. Molecular modeling

Two computational models of proteins, *CHARMM* [Brooks 1983] and *Amber* [Weiner 1984; Weiner 1986], are commonly used to calculate the total energy. Both model the total energy as the sum of the energies in the bonds, bond angles, fixed and variable dihedral angles, van der Waals interactions, electrostatic interactions, and hydrogen bonds:

$$E = \sum_{\text{bonds}} k_b(r - \bar{r})^2 + \sum_{\text{angles}} k_\theta(\theta - \bar{\theta})^2 + \sum_{\substack{\text{fixed}\\ \text{dihedrals}}} k_\omega(\omega - \bar{\omega})^2 + \sum_{\substack{\text{variable}\\ \text{dihedrals}}} |k_\phi| - k_\phi \cos(n\phi) +$$
$$\sum_{i<j} \left(\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}}\right) + \sum_{i<j} \frac{q_i q_j}{\varepsilon r_{ij}} + \sum_{\text{H-bonds}} \left(\frac{C_{ij}}{r_{ij}^{12}} - \frac{D_{ij}}{r_{ij}^{10}}\right)$$

They differ in the parameters for the spring constants and ideal values and in the model of electrostatic interactions, hydrogen bonds, and solvent interactions. *CHARMM* allows different electrostatic models other than the one given; these include letting the dielectric constant vary with distance and computing a dipole moment for a group of atoms far from other atoms. *CHARMM* and *Amber* allow van der Waals interactions, electrostatic interactions, and hydrogen bonding with surrounding solvent. Both models use a constant dielectric for nearby interactions with solvent; however, *CHARMM* uses the group dipole approximation for long-distance electrostatic interaction. *CHARMM* also allows a different hydrogen-bond model that includes the angle between the hydrogen bond and the covalent bond attached to the hydrogen.

These models evaluate the van der Waals interaction on atoms within a five to seven Ångstrom neighborhood radius. The same list is used for multiple iterations of algorithms that calculate the energy. Both models let a user choose among modeling all the hydrogens, only hydrogens that can form hydrogen bonds (*Sculpt's* approach), and no hydrogens. The goal of these models is physical realism, so no constraints are placed on bond lengths or angles.

The parameters in *Sculpt* are based on the *CHARMM* model. *Sculpt,* however, does not model electrostatic and solvent interactions. *Sculpt* also constrains the bond lengths, angles, and fixed dihedral angles to their ideal values rather than let them vary. *Sculpt* uses a different variable dihedral angle function and hydrogen bond model. The variable dihedral angle model uses a quadratic centered at the nearest ideal angles as an approximation of the cosine function (expanding $\cos x$ in an infinite series gives $\cos x = 1 - \frac{1}{2}x^2 + O(x^4)$). The quadratic is faster to evaluate but has a slightly wider energy well. Hydrogen bonds are manually specified at *Sculpt's* program initialization and remain throughout the session rather than being continually evaluated between atoms in neighbor lists. *Sculpt* models a hydrogen bond with a length and angle spring.

These energy models and variants of them are used in commercial molecular modeling systems such as *Sybyl* [Tripos 1988], *Insight* [Biosym 1991], and *Quanta* [Polygen 1991] for several purposes. First, determining the energy allows comparison among multiple conformations. Second, energy minimization moves atoms into locally minimum energy wells. Third, the energy gradient gives a force used in the simulation of the Newtonian motion of atoms. Fourth, the second derivative of the energy allows analysis of the normal (vibrational) modes in a given configuration.

### 1.1. Energy minimization

Energy minimization is used to move atoms into locally minimum energy configurations. A user often lets an energy minimizer shift atoms that are too close and adjust torsional and angular violations after manually editing a molecular model. However, energy minimizers often resolve bad atom separations by moving atoms more than a user intends. Energy minimization is also used as a pre-processing step to find a stable atom configuration before molecular dynamics simulation and normal mode analysis.

### 1.2. Molecular dynamics

Molecular dynamics are used to examine the motion of atoms in a molecule and to move atoms out of local energy minima. Molecular dynamics simulates the motion of atoms in a molecule by simultaneously solving Newton's equation of motion for each atom [Mc Cammon 1977; van Gunsteren 1977]. This requires solving the set of coupled, second-order differential equations given an initial position and velocity for each atom *i* with mass $m_i$:

$$\frac{\partial^2 \mathbf{x}_i}{\partial t^2} = \frac{\mathbf{F}_i}{m_i}\text{, where the force on atom } i \text{ is } \mathbf{F}_i = -\nabla_{\mathbf{x}_i} E(\mathbf{x}).$$

Numerically integrating these equations through time requires time steps shorter than the inverse of the highest frequency in the model. For bond length changes, that frequency is about $10^{10}$ hertz. In proteins this requires time steps on the order of femtoseconds. Because of the small time step and large computation required during each step, dynamics of proteins have only run for simulated time periods of a few nanoseconds [Brooks III 1988; Hermans 1992]. Removing the highest frequency component, flexible bond lengths, allows a two-to-three performance increase [van Gunsteren 1977]. The short time steps, however, prevent molecular dynamics from being a viable method for interactively sculpting protein.

### 1.3. Normal modes

A normal mode analysis gives principal vibration axes in a structure (the most flexible axes of change). A normal mode analysis requires solving an $n \times n$ system of equations that results from taking the second partial derivative of the energy function. This requires batch processing on all but very small molecules. The analysis is valid assuming only small changes in atom positions [Brooks III 1988; Pentland 1989]; large movements require a new analysis. Sculpting a protein by interpolating the normal modes is not a viable approach since a new analysis must run after the protein is deformed.

### 1.4. Interactive modeling

In most molecular modeling systems the user manually positions atoms by a series of purely geometric dihedral angle rotations. For example, commercial modeling packages such as [Biosym 1991; Polygen 1991; Tripos 1988] display a graphical (vector) representation of the covalent bonds and let a user apply rotations about particular bonds. Along the backbone only rotation about the N-$C_\alpha$ and $C_\alpha$-C bonds (refer to Chapter 2) are allowed. This keeps the peptide atoms planar and distances and angles valid. However, no restriction is placed on non-bonded atom separation. Atoms can move closer than allowed by the van der Waals repulsion. In practice a user manually moves the atoms to a good separation and then invokes a batch energy minimizer for fine-grain adjustments. Often the results are not acceptable and the manual/automatic process is repeated.

A research modeling package called *FORME* [Tuffery 1991] lets a user interactively deform a protein backbone (rotations are currently not allowed in sidechains). *FORME* lets a user specify end positions of two atoms in a backbone chain. *FORME* then determines incremental rotations about the N-$C_\alpha$ and $C_\alpha$-C bonds in the backbone that will move the two atoms to their target positions. *FORME* determines the rotations by analyzing the possible rotations allowed if the electron shells of all atoms are modeled with hard spheres (a CPK model). As yet no energy model is used to determine the conformations along the path.

A few systems have allowed limited interactive energy minimization. The GRIP system let a user manually disconnect a bond in the backbone and move a new residue into place [Britton 1977]. Then the user can invoke a structure idealizer that pulls the new residue into proper geometry while keeping the rest of the structure fixed. In crystallography a combination of constraints and restraints (energy model) is used to fit known structure into observed data [Hendrickson 1980].

## 2. Physically-based modeling in computer graphics

### 2.1. Simulation of constrained Newtonian dynamics

Physically-based modeling frequently aids computer animations by automating detailed motion planning and complex object interactions. Miller generates realistic snake motions by modeling muscle contractions with springs [Miller 1988]. Witkin models the energy and momentum of a Luxo lamp jumping hurdles and ski jumps [Witkin 1988]. Terzopoulos models energy in elastically deformable objects such as cloth to create animations of flags [Terzopoulos 1987]. These examples simulate the motion of objects by first stating application-specific conditions about the objects and scene and then solving Newton's equations of motion.

Similar applications use *constraints* to restrict the allowable states of objects and maintain dependencies among objects. Barzel uses constraints in computer animation to specify paths for objects [Barzel 1988]. Witkin uses geometric constraints to assemble models

[Witkin 1987], and he describes a system that lets a user interactively connect and manipulate objects such as a mechanical assembly or tinker-toy [Witkin 1990]. Constraints maintain constant volume in incompressible solids [Platt 1988] and restrict penetration when a ball strikes a trampoline [Platt 1989].

## 2.2. Satisfying new constraints

Drawing systems such as [Claris 1988] let new objects be inserted that have simple constraints, independent of other objects (e.g. constraining a line to be horizontal or two inches long). Satisfying such constraints simply eliminates some degrees of freedom of the object. Some systems allow constraints on objects that are dependent on another object's property. For example, *Sketchpad* let a user create a line that was constrained to be perpendicular or parallel to another line [Sutherland 1963]. Since such constraints are dependent only on another property, they are also easily solved. These constraints are active only during object creation; arbitrary changes are allowed later.

Satisfying arbitrary constraints defined among objects that also are constrained is much more complex and is a subject of active research. Analytic techniques can find solutions to some constraint problems by symbolically solving the set of constraint equations. However, a closed-form solution is usually not possible. Leler surveys many analytical techniques for satisfying constraints [Leler 1987]. Unfortunately, as the complexity of a constraint and the number of constraints increases, analytic methods often fail. Analytic methods have only been successfully applied to systems with a few dozen to one hundred constraints.

An interactive system, called *Thinglab* [Borning 1979], lets a user construct constrained objects in an object-oriented programming environment. Each object has a user-programmed method that satisfies internal constraints (e.g. extending a side to maintain a constant area in a rectangle). If an initial analytical method fails, the system iteratively calls each object until the constraints are solved. The research in *Thinglab* centered on programming-language design and analytic constraint satisfaction, so only models with a few dozen objects and constraints were used.

Grant discusses a system that finds a configuration of objects that best satisfies an arbitrary set of constraints [Grant 1991]. An associated error function measures the violation of each constraint (constraints are modeled as restraints in his work). The system finds an optimal solution (a global minimum of the sum of error functions) using simulated

**annealing [Kirkpatrick 1983] which iteratively surveys a random sampling of the parameter space. The approach is very useful when one has little intuition about the optimal model configuration and the model contains many constraints. Theoretical results give a probability that the computed solution is a global minimum. This probability depends on the number of iterations run and other parameters of the algorithm; the optimum is not guaranteed without an infinite number of iterations. In practice acceptable probability requires on the order of hours of computation.**

# Chapter 5
# Algorithms and Analysis

This chapter describes and analyzes the algorithms in *Sculpt* that tie the problem in Chapters 1 and 2 and the mathematical model in Chapter 3 together into an interactive modeler. The first section discusses how atoms are interactively manipulated while satisfying the mathematical model. The remainder of the chapter analyzes the computational complexity of each algorithm. This chapter restricts the discussion to those algorithms necessary for protein sculpting on a single-processor workstation. Chapter 6 discusses which algorithms benefit from parallel processing, and Chapter 7 examines an approximation that drastically reduces computation.

## 1. Interaction with springs and nails

A chemist moves an atom in *Sculpt* by *tugging* it to the desired position. Tugging follows the pick-and-drag methodology found in many window systems. A chemist positions a cursor over the atom, invokes a pick command, and drags (moves) the cursor to the desired direction (Chapter 8 presents a detailed description of the user interface). The system attaches a zero-length spring between the picked atom and the cursor. The system reads the cursor's position multiple times during dragging and changes the end of the spring. At each read the system adds the potential energy in the spring to the total energy in the protein. The system then minimizes the energy and displays the new atom positions. Therefore, moving an atom is accomplished through a series of individual *tugs* on a spring (attached between the atom and the cursor) and energy *minimizations*.

The motion of the tugged atom is not always as expected. Assuming no opposing forces or constraints, the atom does move in the direction of the tug (towards the cursor). Since the mathematical model only considers atom positions, and not velocities, the atom can be placed exactly at the desired position. However, an opposing force or constraint can prevent the atom from following the cursor. Suppose a tugged atom is bonded to another atom exactly opposite the direction of the tug. If the bonded atom cannot move, the atom will not follow the tug (*Sculpt* does not model the breaking and formation of bonds).

Part A in Figure 5.1 illustrates this case with atom $F$ fixed. The tangential component of the tug direction is followed if another direction is picked as in Part B.

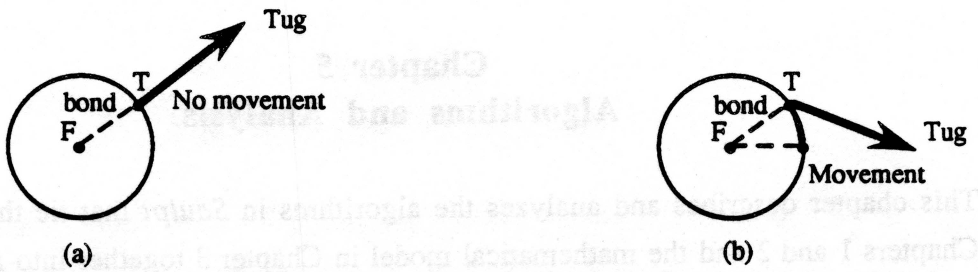


**Figure 5.1: The point $T$ is constrained to the circle. (a) shows a tug that does not move the point, and (b) shows a tug that moves $T$ along one component of the tug direction.**

Tugging provides a method for moving between local minima. Figure 5.2 shows a case where tugging pulls an atom, $T$, from one energy minimum to another. Assume atoms $F_1$ and $F_2$ are fixed. The arrows show the direction and magnitude of the atom attractions. Initially (Part A) atom $T$ is closer to $F_1$ and has a stronger attraction to $F_1$ than $F_2$. The tug in Part B pulls the atom towards $F_2$. Part C shows the final result. This example shows where user intervention can overcome a local energy minimum.

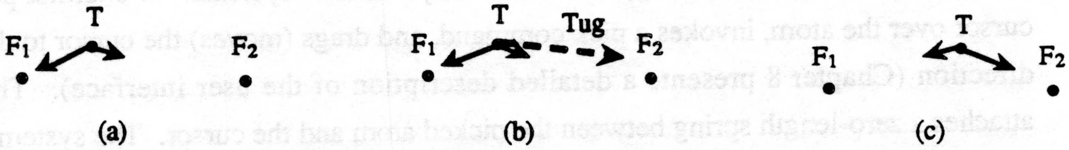
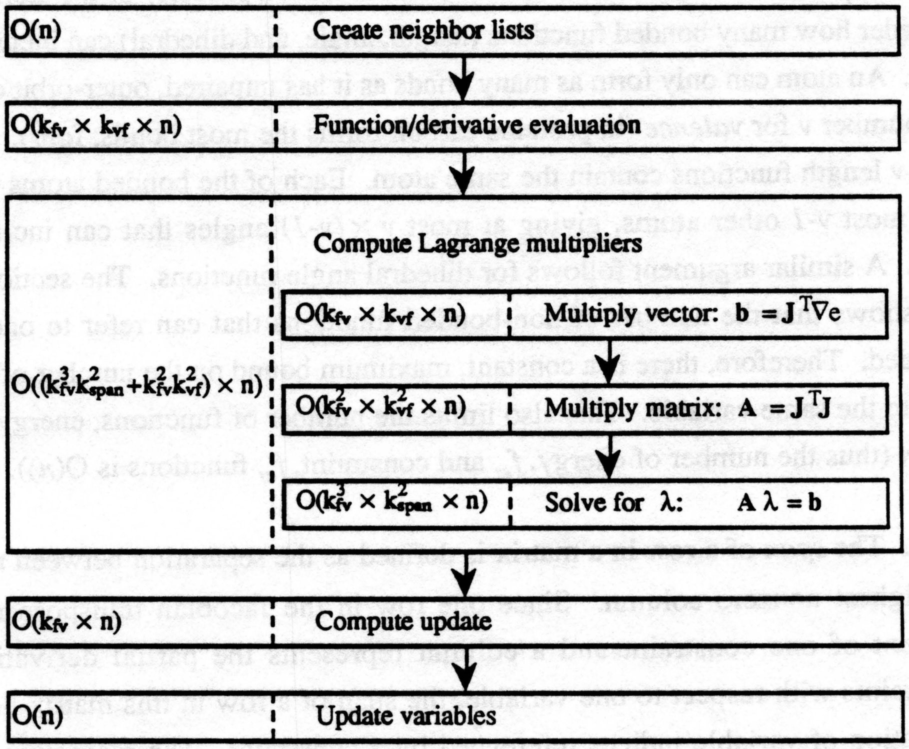


**Figure 5.2: A series of figures showing a tug that pulls $T$ between minima. (a) shows attractions before the tug; (b) shows a tug applied to T; (c) shows attractions afterwards.**

*Nails* let a chemist fix an atom's position to a three-dimensional position in space. Nails provide additional control over atom motion. Often many atoms follow the motion of one atom due to attraction, even though a chemist only wants to move the one atom. For example atom $F_1$ in Figure 5.2 would follow atom $T$ if it was not nailed to its position. Nails fit into the mathematical model in two ways—with an equality constraint on the atom's position or a strong, zero-length spring between the atom and the position.

## 2. Computational complexity

This section begins a detailed analysis of the computational complexity of the minimization algorithm described in Chapter 3. I use big-O, or order, notation to describe the worst-case complexity of the algorithms. Big-O analysis is a formalism for expressing the intuitive idea that the number of steps required to execute an algorithm is proportional to some function of the number of items processed. For instance, an $O(n^2)$ algorithm executes in a number of steps proportional to the square of the number of items. Figure 5.3 lists the computational complexity of each step in the minimizer. The remainder of this chapter explains how this algorithm solves the protein sculpting problem in time that increases linearly, $O(n)$, with the number of variables.


$O(n)$
Create neighbor lists
$O(k_{fv} \times k_{vf} \times n)$
Function/derivative evaluation
$O((k_{fv}^3 k_{span}^2 + k_{fv}^2 k_{vf}^2) \times n)$
Compute Lagrange multipliers
$O(k_{fv} \times k_{vf} \times n)$
Multiply vector: $b = J^T \nabla e$
$O(k_{fv}^2 \times k_{vf}^2 \times n)$
Multiply matrix: $A = J^T J$
$O(k_{fv}^3 \times k_{span}^2 \times n)$
Solve for $\lambda$: $A \lambda = b$
$O(k_{fv} \times n)$
Compute update
$O(n)$
Update variables

| **Variable** | | **Useful derived quantities** | |
|---|---|---|---|
| n | number of variables = 3 x number of atoms | $f_c$ | number of constraint functions $\leq k_{fv} \times n$ |
| | | $f_e$ | number of energy functions $\leq k_{fv} \times n$ |

| **Constants** | | **Notation** | |
|---|---|---|---|
| $k_{vf}$ | maximum variables in a function = 12 | $\nabla e$ | gradient of energy, (n x 1) |
| $k_{fv}$ | maximum functions containing same variable | $\lambda$ | Lagrange multipliers, ($f_c$ x 1) |
| $k_{span}$ | maximum range of indices in a constraint | J | gradient of constraint functions, $\nabla c$ (n x $f_c$) |
| | | $J^T$ | transpose of J, ($f_c$ x n) |

**Figure 5.3: Computational complexity of *Sculpt's* constrained minimization algorithm.**

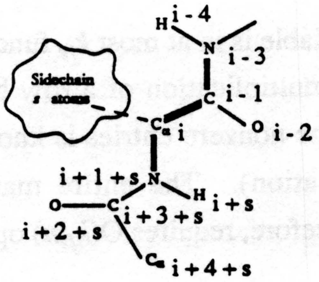
## 2.1. Notation

**n.** The only free parameter in the algorithm is the number of atoms. Each atom has three variables that specify its position. The total number of variables, *n*, equals three times the number of atoms.

**$k_{domain}$.** The domain of the functions gives the maximum number of variables each function can reference, denoted $k_{domain}$. The dihedral angle function uses the most variables—twelve, four atoms, each with three variables.

**$k_{fv}$.** The number of functions that can reference the same variable is bound from above, denoted $k_{fv}$ (functions per variable). This comes from a property of the atoms in a protein. Consider how many bonded functions (length, angle, and dihedral) can include a particular atom. An atom can only form as many bonds as it has unpaired, outer-orbit electrons—call this number $v$ for *valence* (in proteins carbon forms the most bonds, four). This means at most $v$ length functions contain the same atom. Each of the bonded atoms is also bonded to at most $v$-$1$ other atoms, giving at most $v \times (v\text{-}1)$ angles that can include the initial atom. A similar argument follows for dihedral angle functions. The section on neighbor lists shows that the number of non-bonded functions that can refer to one atom is also bounded. Therefore, there is a constant, maximum bound on the number of functions that refer to the same variable. This also limits the number of functions, energy or constraint, to $k_{fv}n$ (thus the number of energy, $f_e$, and constraint, $f_c$, functions is O($n$)).

**$k_{span}$.** The *span* of a row in a matrix is defined as the separation between the lowest and the highest nonzero column. Since one row in the Jacobian transpose represents the gradient of one constraint and a column represents the partial derivative of all the constraints with respect to one variable, the span of a row in this matrix depends on the separation of variable indices referenced by a constraint. The maximum separation in proteins is bound by a constant, $k_{span}$. This requires an atom numbering scheme so that a constraint only references atoms within a fixed range of indices. A scheme that I use sequentially numbers atoms along the protein backbone with breaks at the sidechains. Part A in Figure 5.4 shows the numbering scheme used for proteins. Note the only discontinuity in the numbering is at the sidechain. The constant, $k_{span}$, is derived by examining a dihedral angle function (formed by four atoms) defined on atoms before and after a tryptophan sidechain. This is the worst case because a dihedral function references the most variables and a tryptophan sidechain contains the most atoms, ten. The right side of the figure derives the constant.



(a)

$$
\begin{aligned}
k_{span} &= \underset{\text{sidechain } s}{\text{maximum}} \text{ span dihedral}(N^{i-3}, C^{i-1}, C_{\alpha}^{i}, N^{i+1+s}) \\
&= \underset{\text{sidechain } s}{\text{maximum}} \,[\,(i+1+s) - (i-3)\,]\text{ atoms} \\
&= 3\frac{\text{variables}}{\text{atom}} \times (4 + 10\frac{\text{atoms}}{\text{tryptophan sidechain}}) \\
&= 42 \text{ variables}
\end{aligned}
$$

(b)

**Figure 5.4: Derivation of maximum span of a constraint function. (a) lists the numbering scheme, and (b) derives $k_{span}$.**

This restriction on constraint functions does not prevent interactions between arbitrary atoms. Constraint functions are only applied among covalently-bonded atoms. Arbitrary interactions between non-bonded atoms are modeled with *energy* functions. Separation of variable indices in energy functions is not restricted. Similarly, hydrogen bonds are modeled with energy functions. A hydrogen "bond" is actually an attraction between two partially charged atoms—not a bond formed with shared electrons.

A disulfide bond violates this restriction. This covalent bond occurs between atoms in separate sidechains which can have indices arbitrarily separated. Most proteins do not contain disulfide bonds; those that do usually contain from one to ten. The complexity analysis remains valid when a model contains a fixed number of constraints among arbitrary atoms. I do not factor this into the definition of $k_{span}$ but address it separately at the end of Section 4.2.

### 2.2. Vector operations

**Function and derivative evaluation.** Each function contains at most $k_{domain}$ variables. Evaluating a function and the partial derivative with respect to each of its variables requires O($k_{domain}$) operations. This is done for each of the $f_e$+$f_c$ functions, resulting in O($k_{domain}(f_e+f_c)$) = O($k_{domain}k_{fv}n$) operations.

**Compute update.** Section 4.3 in Chapter 3 gives the following equation for computing the update: **d ← -∇e + Jλ + pJc**. Assuming previous steps provide the function gradients and the Lagrange multipliers, this algorithm involves two vector sums requiring O($n$) operations and two matrix-vector multiplications. Both multiplications involve the $n \times f_c$ matrix of the constraint gradients and an $f_c \times 1$ vector. The multiplication requires multiplying each of the $n$ rows in $J$ by the vector. A row $r$ contains the partial derivatives of each function with respect to variable $r$. Since a variable is in at most $k_{fv}$ functions, each row contains at most $k_{fv}$ nonzero entries. Thus the multiplication of a row by a vector requires $O(k_{fv})$ operations, assuming the location of the nonzero entries is known (a data structure presented in Section 4 keeps this information). The entire matrix-vector multiplication (and the computation of the update), therefore, requires $O(k_{fv}n)$ operations.

**Update variables.** This step just adds the update vector to the variables, involving $O(n)$ operations.

## 3. Neighbor lists

The non-bonded interactions, van der Waals and electrostatic, are computed for atoms near one another. *Sculpt* keeps a list of the atoms within a spherical neighborhood of radius $r_{neigh}$ of each atom. This radius is different for the van der Waals and the electrostatic interactions, so *Sculpt* actually needs two neighbor lists (*Sculpt* does not currently compute electrostatic interactions, so only one list is presently computed). Chapter 2 discusses advantages and disadvantages for using neighbor lists; this section only presents the algorithm that determines the members of the lists.

An algorithm can compute these lists in $O(n)$ steps by exploiting a property of atoms. Each atom occupies a nonzero volume, referred to as its electron shell in Chapter 2. Levinthal shows from this fact that only a finite number of atoms can fit within a finite volume [Levinthal 1966]. A conservative bound on this number is $r^3_{neigh}$ (the neighborhood volume divided by the volume of the smallest atom, hydrogen, with a radius of one Ångstrom).

Bentley gives an algorithm with linear complexity that computes neighbor lists that have a bounded number of entries [Bentley 1979]. First, uniformly subdivide space into cubes with $r_{neigh}$ on each side. Second, deposit each atom into the cube at its position. Third, set the neighbor list for each atom to the atoms in its cube and the adjacent cubes. This algorithm takes $O(n)$ steps.

Storing all the cubes can require a lot of memory if the molecule is large and the cubes are small. Rather than storing a full three-dimensional grid of cubes, I use a hashing scheme based on the three-space coordinate. Only cells that contain atoms are stored. Collisions in the hash function are not resolved, so two atoms with the same hash index are placed in the same cell regardless of their separation.

The list of neighbors for each atom is initially set to the list in its cell. A final step of the algorithm removes those atoms in the list that have the same hash index but are far apart. In practice this pruning can reduce the number of interactions by forty percent.

## 4. Finding Lagrange multipliers

Finding the Lagrange multipliers requires matrix multiplication and solution of linear equations. The system of equations is very sparse (typically more than ninety-five percent of the entries are zero). Exploitation of this property by data structures and algorithms provides drastic reduction in memory and computational requirements. The first section presents the data structure that holds the matrix. The second section examines the complexity of the algorithms that find the Lagrange multipliers.

### 4.1. Data structure

*Sculpt* stores the transpose of the Jacobian (gradient) of the constraints, $J^T$. The matrix has $f_c$ rows and $n$ columns. Element *(i,j)* holds the first partial derivative of constraint $i$ with respect to variable $j$. Since each constraint is defined over a small number of variables—much smaller than the total number of variables—most entries in the row are zero. In fact an average row in a small test model with 300 atoms is 98.7 percent zero. The percentage increases with the problem size.

**Basic structure.** A new sparse-matrix data structure, optimized for this application, is used. The data structure is a variant of one described by Knuth [Knuth 1973, p. 300] that stores each nonzero element of a matrix in a node of a linked list. Knuth's node contains the element's value, the row and column indices, and pointers to the next row element and column element. Figure 5.5 shows the data structures used in this research. The structure does not store the row and column index at each node, but instead, stores pointers to the row and column headers which contain the index. Inserting a row or column requires changing the indices in the row and column headers rather than the matrix elements.

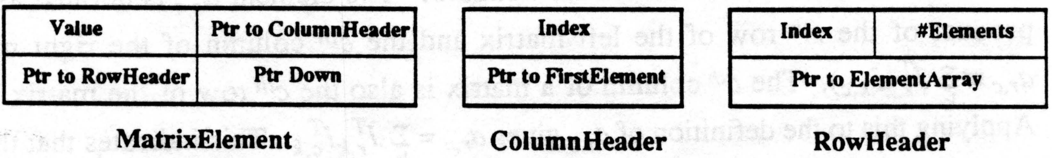


**Figure 5.5: Three data structures used in the sparse-matrix definition.**

The matrix element does not contain a pointer to the next element in the row. Since each row holds the partial derivatives of a specific constraint, the number and location of the nonzero elements is known when a constraint is created. The row header, therefore, contains the number and a pointer to an *array* of nonzero elements. Figure 5.6 shows a full matrix in the upper-left and its associated sparse-matrix data structure. The additional arrays, *RowDirect* and *ColumnDirect,* provide direct access to the rows and columns, respectively.

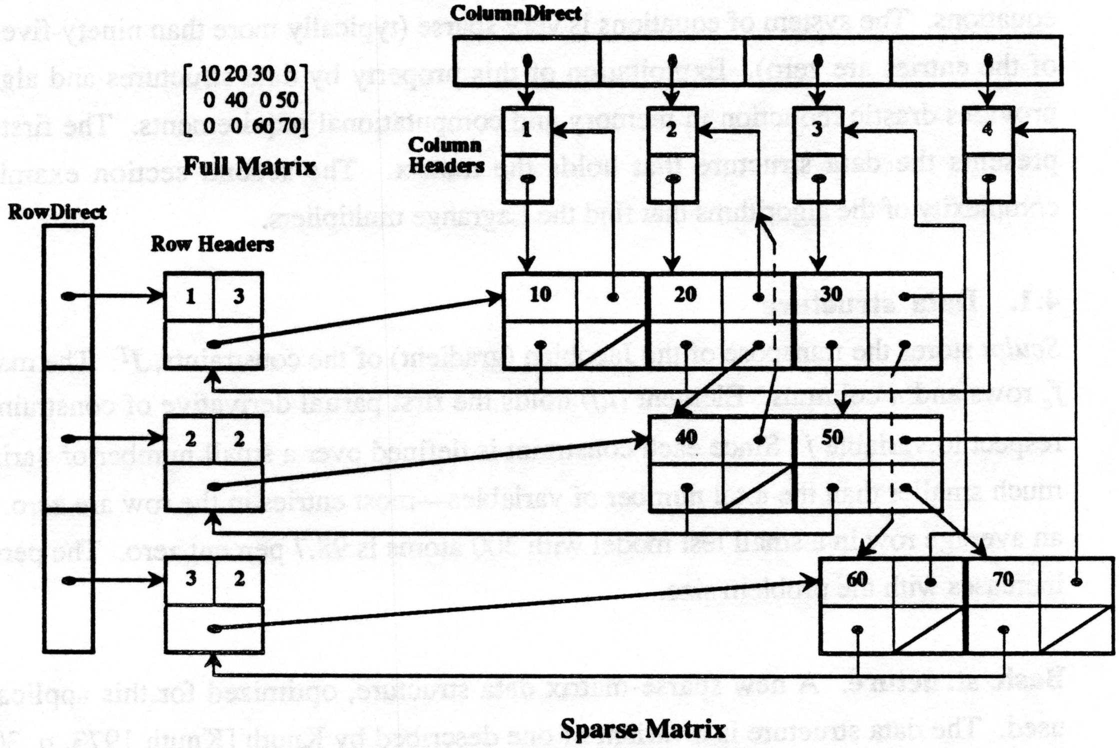


**Figure 5.6: Full matrix in upper-left represented in *Sculpt's* sparse-matrix data structure.**

**Sparsity of matrix product, A≡J$^T$J.** The system also uses the data structure to store a second matrix, the product of the Jacobian transpose and the Jacobian, **A≡J$^T$J.** The sparsity pattern of matrix *A* closely resembles the pattern in $J^T$.

Which elements in the *A* matrix are nonzero? The element $a_{r,c}$ is defined as the inner product of the $r^{th}$ row of the left matrix and the $c^{th}$ column of the right matrix (i.e. $a_{r,c} \equiv \sum_k J^T_{r,k} J_{k,c}$). The $c^{th}$ column of a matrix is also the $c^{th}$ row of the matrix transpose. Applying this to the definition of $a_{r,c}$ gives $a_{r,c} = \sum_k J^T_{r,k} J^T_{c,k}$. This indicates that the element is nonzero only when rows *r* and *c* in the Jacobian transpose contain a nonzero entry in the same column. Comparing each row of the Jacobian transpose against its other rows for common, nonzero columns gives the nonzero elements of $A$. Since no rows (constraints) are inserted or removed during a program session, the sparsity pattern of $J^T$ and $A$ remains constant. (Chapter 7 relaxes this assumption in limited cases.) A pre-processing step analyzes the sparsity pattern of $J^T$ to create the sparse matrix $A$.

A matrix times its transpose, and vice versa, is symmetric. A symmetric matrix equals its transpose (i.e. $\mathbf{A} = \mathbf{A}^T$). The maximum number of elements in a row of this matrix, and likewise in a column, is bound by a constant. Each nonzero element with index $(r,c)$ implies that constraints $r$ and $c$ share a common variable. A given constraint uses at most $k_{domain}$ variables that are each in at most $k_{fv}$ functions. Therefore, a given constraint has common variables with at most $k_{domain}k_{fv}$ other constraints. This implies a given row in $A$ has at most $k_{domain}k_{fv}$ nonzero elements.

## 4.2. Algorithm

Finding the Lagrange multipliers requires three steps: matrix-vector multiplication, $\mathbf{b} \leftarrow \mathbf{J}^T\nabla\mathbf{e}$; matrix-matrix multiplication, $\mathbf{A}\leftarrow\mathbf{J}^T\mathbf{J}$; linear equation solution for $\lambda$, $\mathbf{A}\lambda=\mathbf{b}$. Each step is discussed separately.

**Matrix-vector multiplication.** This step multiplies the $f_c \times n$ Jacobian transpose matrix by an $f_c \times 1$ vector. The algorithm multiplies each row of the matrix by the vector. Since each row contains at most $k_{domain}$ nonzero elements, multiplication of one row requires $O(k_{domain})$ operations. The entire multiplication takes $O(k_{domain}f_c) = O(k_{fv}k_{domain}n)$ operations.

**Matrix-matrix multiplication.** This multiplication requires calculating the value for each nonzero element in matrix $A$. The location of the nonzero elements is determined at program initialization. The maximum number of entries in matrix $A$ is the number of rows, $f_c$, times the maximum number of elements per row — $O(f_ck_{domain}k_{fv}) = O(k_{domain}k_{fv}^2n)$. Determining the value of a given entry, $(r,c)$, requires multiplying rows $r$ and $c$ of the Jacobian transpose. This requires $O(k_{domain})$ operations since the maximum number of nonzero entries in a row is $k_{domain}$. The computational complexity for the entire multiplication is $O(k_{domain}^2k_{fv}^2n)$.

**Linear equation solution.** Solving the system of equations in linear time critically depends on a sort of the rows in the Jacobian transpose. At program initialization the rows of the Jacobian transpose are sorted relative to the smallest index (column) referenced. Matrix-matrix multiplication of this sorted matrix yields a band-diagonal matrix, one whose nonzeros lie within a constant distance (band) from the diagonal. The number of operations in Gaussian elimination on a banded-diagonal matrix increases linearly with row dimension. The remainder of this section analyzes the sparsity pattern in the matrix *A* and the complexity of Gaussian elimination applied to it.

The worst-case structure of matrix *A* depends on the largest possible span of an arbitrary row, say *i*. What are the smallest and largest column numbers that are nonzero in the row? Column *j* in the row is nonzero if and only if constraints *i* and *j* reference the same variable. Assume constraint *i* references variable indices *L* and $L+k_{span}$, the maximum separation of variable indices in a constraint. Figure 5.7 shows the arrangement of the sorted Jacobian transpose around constraint *i* that yields the largest span in *A* after matrix-matrix multiplication.

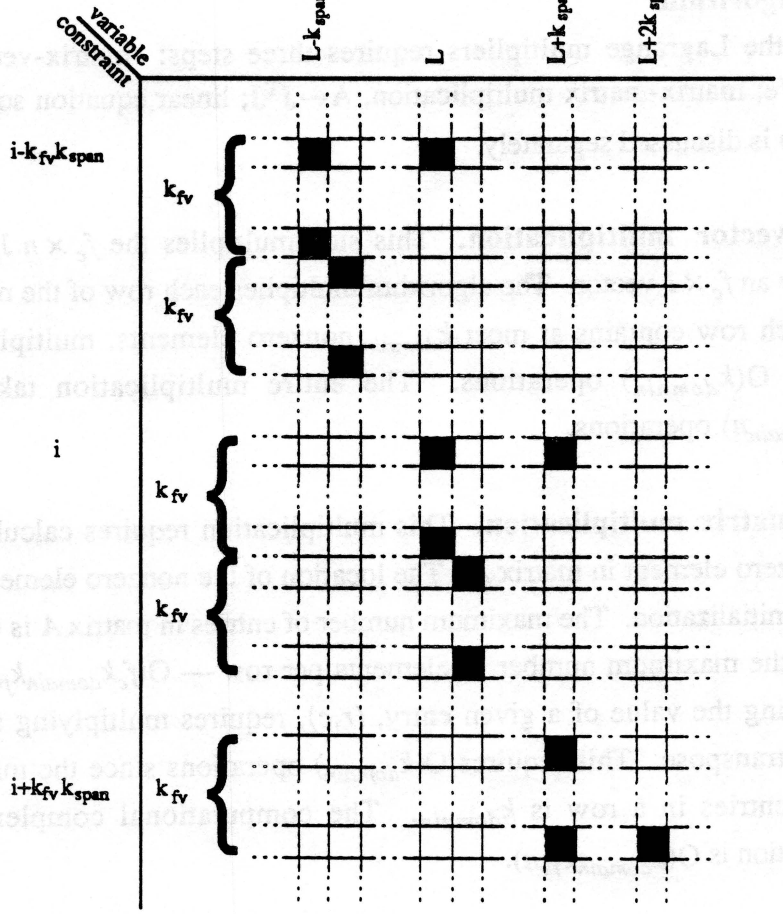


**Figure 5.7: Worst-case structure of the Jacobian transpose.**

The figure shows one constraint that references variable indices $L$ and $L\text{-}k_{span}$, and one that references $L+k_{span}$ and $L+2k_{span}$. The fact that a variable appears in at most $k_{fv}$ functions limits the number of constraints that can appear between the first constraint in the figure and constraint $i$. Only $k_{fv}$ constraints can reference each of the variable indices between $L\text{-}k_{span}$ and $L$. Therefore the smallest-numbered constraint that can reference a variable in constraint $i$ is $i$ - $k_{fv}k_{span}$. Similarly, the largest-numbered constraint is $i + k_{fv}k_{span}$. Therefore, row $i$ in matrix $A$ can only contain nonzero entries $k_{fv}k_{span}$ columns before and $k_{fv}k_{span}$ columns after the diagonal element. The same argument holds for each row in $A$. Matrix $A$ is called band-diagonal with bandwidth $k_{fv}k_{span}$. All the nonzero elements in a band-diagonal matrix with bandwidth $b$ lie within $b$ elements of the diagonal. Figure 5.8 shows such a matrix with bandwidth $b$.

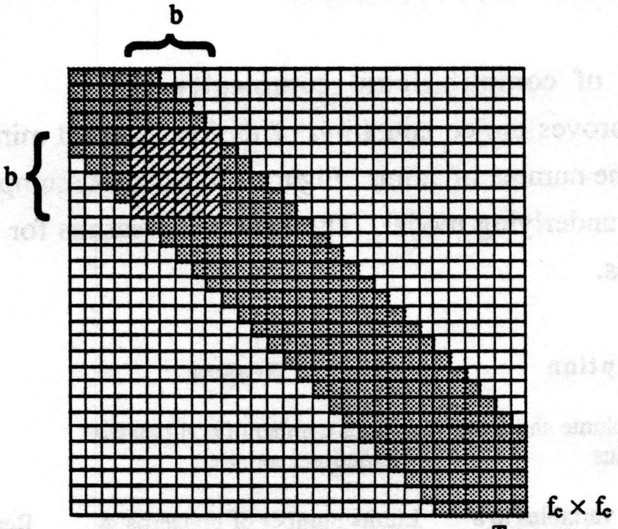


**Figure 5.8: Worst-case structure of the product $J^TJ$. Empty squares represent zeros.**

Gaussian elimination reduces a band-diagonal matrix in $O(b^2f_c)$ operations. Figure 5.8 highlights one block of a band-diagonal matrix. Consider the operations necessary to eliminate the elements under the diagonal in the left-most column of the highlighted block. This requires eliminating the value in the $b\text{-}1$ rows following the first row in the block. Each row elimination requires multiplying and adding $b$ elements in the row. So eliminating the column elements below a diagonal element requires $O(b^2)$ operations. Repeating this for each diagonal element gives the computational complexity listed. Substituting constants gives the following number of operations to solve the linear equations: $O(b^2f_c) = O((k_{fv}k_{span})^2k_{fv}n) = O(k_{fv}^3k_{span}^2n)$. In general some elements within a band are also zero. Equation solvers can often use this information to reduce the average number of operations.

Assume a model now contains one distance constraint, *i*, defined between two arbitrary atoms (e.g. a disulfide bond). This constraint only references variables in $k_{cv}$ other constraints, but a given constraint, *j*, can be arbitrarily separated from *i*. The nonzeros in this matrix are either within the original bandwidth or at a few stray places such as *(i,j)*. The computational complexity of Gaussian elimination on such a matrix does not increase, since only the elements in the column under these stray elements are filled. A similar argument proves that the complexity of Gaussian elimination remains linear when there is some constant number of distance, angle, and dihedral angle constraints defined among arbitrary atoms. However, an unbounded number of these constraints yields a general matrix which requires O($m^3$) operations.

## 5. Summary of computational complexity

This chapter proves the computation of the constrained minimization algorithm scales linearly with the number of atoms. Figure 5.9 lists the assumptions the preceding analysis makes on the underlying model. The table lists reasons for each assumption and some additional notes.

| Assumption | Reason | Note |
|---|---|---|
| Nonzero object volume and fixed neighborhood radius | Bounds number of non-bonded interactions per atom | |
| Bound number of variables in a function | Limits number of nonzeros in Jacobian | Reasonable but prevents functions requiring global information |
| Bound number constraints that reference same variable | Same as above | Reduces probability of an over-constrained model |
| Bound range of variable indices in a constraint | Yields block-diagonal matrix | Gaussian elimination is cubic without this; relaxed in certain cases |
| Constraints are not added or removed | Allows Jacobian matrix pre-processing | Relaxed in Chapter 7 |

**Figure 5.9: Assumptions made in analysis of *Sculpt's* minimization algorithm.**

# Chapter 6
# Parallel Computation

This chapter discusses improving system performance with parallel processing. *Parallel* computation divides *data* into portions that can be processed independently by different processors (often called ***processing elements***). ***Concurrent*** computation divides an ***algorithm*** into portions that can be processed simultaneously. The *Sculpt* minimization algorithm contains steps that can proceed in parallel and concurrently. These techniques reduce the constant of proportionality of the algorithm, but do not reduce the linear complexity. The solution of linear equations contains a property that requires O($n$) operations regardless of the number of processors. Chapter 9 lists some timing results made using the techniques described in this chapter.

Figure 6.1 presents the steps in the minimization algorithm that operate in parallel and concurrently. The vertical axis represents data dependencies; a stage cannot begin until the previous stage completes. The horizontal axis shows parallel processing within an algorithm and current processing between algorithms. Boxes with round corners represent stages in the algorithm. Round-corner boxes beside one another (within Stages 2 and 4), execute concurrently. Small square boxes represent processing elements that execute in parallel. The figure contains three types of computation: neighbor-list determination, vector and function operations, and solution of linear equations. The remainder of the chapter discusses each of these.

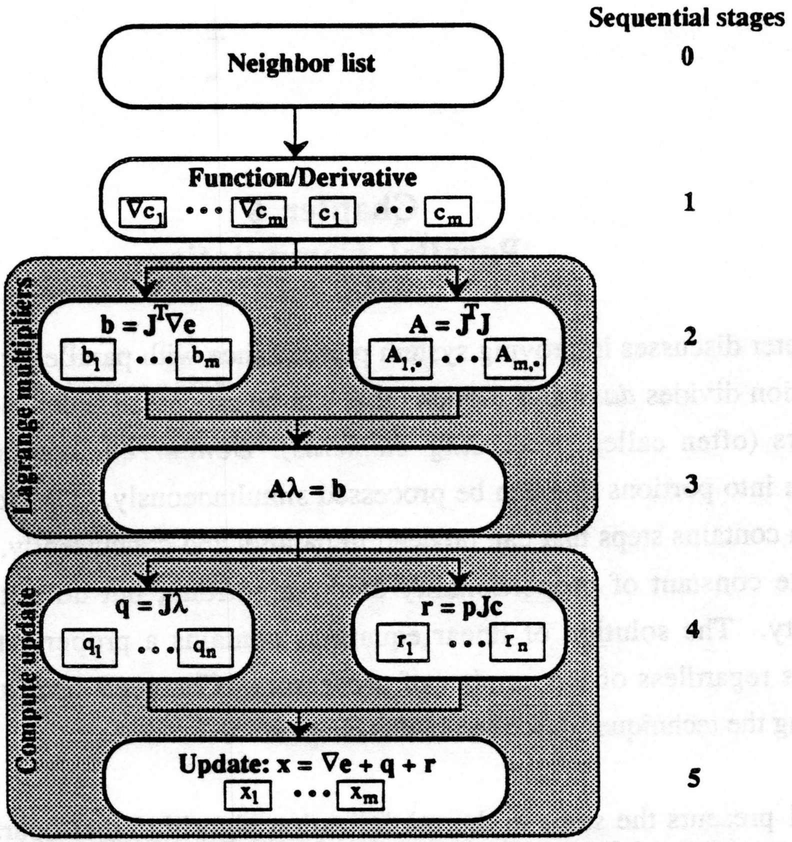


**Figure 6.1: Block diagram of concurrent and parallel segments of *Sculpt's* minimization algorithm. Execution of a block must wait until preceding blocks finish. Blocks beside one another execute in parallel.**

## 1. Neighbor list determination

Function and derivative evaluation must wait until the neighbor-list computation completes because functions that model non-bonded interaction (Van der Waals and electrostatic) interactions require a neighbor-list for each atom. This dependency can be avoided if the neighborhood radius is enlarged. The functions that model non-bonded interactions can use the same neighbor-lists for multiple iterations if the neighborhood encompasses the maximum possible atom movement during the iterations. This approach lets computation of new neighbor-lists proceed concurrently with function and derivative evaluation. *Sculpt* partially implements this approach. A user can choose whether or not a list is computed on each iteration. However, the list is not determined concurrently with the rest of the algorithm.

## 2. Vector and function operations

Stage 1 evaluates the constraint and energy functions and their derivatives in parallel. *Sculpt* stores each constraint value in one element of an array and the gradient of each constraint in one row of the Jacobian transpose matrix. Simultaneously writing the values to memory does not cause contention. Memory contention does arise when energy gradients are stored. *Sculpt* stores the sum of all the energy gradients. Since each energy function can reference the same variable, parallel updating of the sum can cause errors. *Sculpt* adjusts for this by computing the energy gradients on a small number (four or eight) of processors, each of which holds a local copy of the sum. When all the processors complete, a final step combines the local copies.

Stage 5 performs a vector addition, and Stages 2 and 4 calculate a matrix-vector multiplication. Processors can compute the elements of the resulting vector in parallel. If the number of elements in a vector result equals the number of processing elements, a stage requires only O(1) computations. Note that stages 2 and 4 contain algorithms that can proceed concurrently.

### 3. Linear equation solution

A band-diagonal matrix requires O(*kn*) steps to solve using Gaussian elimination. Parallel processing can reduce the constant of proportionality, *k,* but cannot lower the linear complexity. Before any row is reduced, all prior rows must be reduced. Consider a tridiagonal matrix—a band-diagonal matrix with one band above and below the diagonal. Eliminating the rows below a diagonal element, say *(i,i),* affects the next row, *i+1*. This prevents processing row *i+1* until completion of row *i*. Therefore, processing row *1* must complete before beginning row *2,* which must complete before beginning row *3,* etc. The dependencies between rows keep this algorithm O(*n*) regardless of the number of processors.

Iterative solution of the system of equations holds promise for reducing the average computation. Iterative solutions of a band-diagonal system of equations require O(*n*) operations in the worst-case, but may require fewer on average. These methods hold promise for improved performance and should be considered in future research.

# Chapter 7
# Structural Hierarchy

The sculpting process, as described so far, only allows tugs on atoms. This works well for changing relative atom positions and twisting backbone segments into secondary structures (e.g. a helix). However, moving an entire section of secondary structure as a unit is cumbersome with this method. Consider the steps needed to change the orientation of a helix without changing its internal configuration. Tugging one atom after another changes the internal arrangement. Instead, simultaneous tugs applied to multiple atoms are needed to move the entire helix simultaneously. This approach eventually orients the helix properly, but any variation in the relative tug strengths changes the internal arrangement of the helix.

Arbitrary atoms in *Sculpt* can be grouped into higher-level objects that facilitate movement of secondary structures. Moving the object moves the atoms it contains according to a programmable model. In a simple case, a translation of an object translates all of its atoms the same amount. Two examples help illustrate applications and benefits of higher-level objects. A chemist freezes the backbone of a helix, representing its atoms with a cylinder. When the chemist tugs the cylinder, the backbone atoms within it move with the cylinder. The sidechain atoms still interact with other atoms as usual, but the backbone atoms maintain their relative orientations. In this example the atoms move exactly the same as the rigid cylinder. A second application lets the chemist bend the cylinder (not yet implemented). Now the backbone atoms further from the cylinder center move more than those near the center. Mapping a change in cylinder parameters to a change in atom positions is programmable. Discovering and then applying individual atom tugs that solve the two examples is much more difficult and time-consuming than using groups.

This chapter discusses how a hierarchical model of protein structure, coupled with arbitrary grouping of atoms within deformable objects, significantly improves the sculpting process. The first section details some advantages, including enhanced user interaction and performance. The second section lists conditions placed on grouped atoms. The third section presents modifications of the algorithm and data structures described in previous chapters. Grouping atoms eliminates variables and constraints. It also introduces new variables that define the group and constraints that connect the group to the protein. The third section also discusses how these changes affect the sparse matrix and complexity analysis. The fourth section addresses how a non-bonded interaction between an atom in a group and one outside a group is handled in *Sculpt*. Simply eliminating grouped atoms from all non-bonded interactions lets grouped atoms move arbitrarily close to other atoms. The concepts presented in this chapter that are not implemented in *Sculpt* are explicitly noted.

## 1. Motivation

Sculpting proteins, like many assembly tasks, requires working with a model at multiple levels of detail. Consider the steps in assembling an automobile. At one stage, a worker stitches seat upholstery, and another puts screws into an alternator. At a later stage, someone combines these objects by placing assembled seats into the passenger compartment and the alternator onto the engine. At the final stage, quality control tightens particular screws, though not necessarily all. This process requires multiple levels of detail. Similarly, protein sculpting requires moving backbone atoms into a secondary structure, orienting the structure relative to other assembled structures, and then adjusting sidechain atoms for better fit between structures. At each stage of the process, a system should only model what is important. Therefore, *Sculpt* may not need to model all the atom positions individually while changing the relative position of secondary structures.

### 1.1 Hierarchy and groups

*Sculpt* maintains a hierarchical model of a protein's primary structure that is shown in Figure 7.1. The figure shows a protein contains a sequence of residues that each contain a backbone and a sidechain, each of which contains atoms.

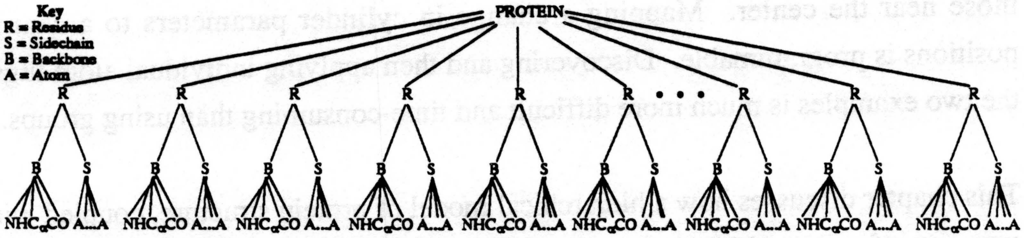
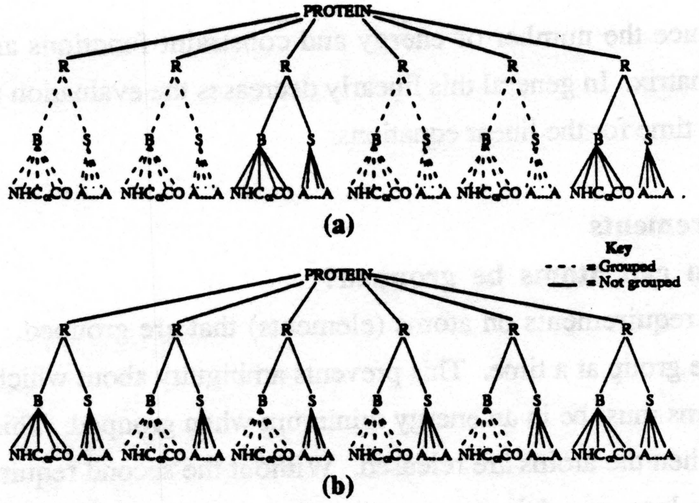

Key
R = Residue
S = Sidechain
B = Backbone
A = Atom
PROTEIN
R R R R R R • • • R R R
B S B S B S B S B S B S B S B S B S
NHC$_\alpha$CO A...A NHC$_\alpha$CO A...A NHC$_\alpha$CO A...A NHC$_\alpha$CO A...A NHC$_\alpha$CO A...A NHC$_\alpha$CO A...A NHC$_\alpha$CO A...A NHC$_\alpha$CO A...A NHC$_\alpha$CO A...A

**Figure 7.1: Protein hierarchy modeled in *Sculpt*.**

*Sculpt* lets a chemist form arbitrary groups of atoms, though in practice, particular groups prove more useful than others. A chemist can group any set of nodes in the structural tree and all their children. A chemist can group a secondary structure by specifying each residue in it. However, a future version of *Sculpt* will facilitate this common operation by maintaining a data structure that keeps lists of residues that are in particular secondary structures. Using these lists, a chemist will first specify the secondary structure and then form a group with the entire structure or just its backbone. The system will then automatically group nodes in the structural hierarchy. Figure 7.2 shows two illustrations of a protein hierarchy with elements in a group represented with dashed lines. The top figure shows grouped residues in different segments of the chain (for example, a group of all the atoms in a sheet). The bottom figure shows a group composed of the backbone of a continuous segment of residues (for example, a group of the backbone atoms in a helix).



(a)

(b)

**Figure 7.2: Dashed lines illustrate two examples of atom groups: (a) shows a group of nonadjacent residues; (b) shows a group of adjacent backbone segments.**

Creating the structural hierarchy and the lists of residues in secondary structures is specific to proteins. The frequency of operations anticipated on secondary structures justifies implementing the lists. The minimizer, however, only uses an application-independent structural hierarchy whose leaves represent variables in the minimization (in this case, atom positions). The minimizer is informed when sets of nodes in the hierarchy are grouped. Therefore, creating the protein hierarchy is application-specific; using it is not.

### 1.2 Performance improvement

Grouping atoms not only makes protein sculpting easier and more natural, it can also dramatically improve system performance. The variables that define a group replace the variables for atoms within it. Similarly, functions that connect a group to the protein replace functions that connect atoms within the group (bond lengths, angles, dihedral angles, and hydrogen bonds; non-bonded interactions are discussed later). The number of variables and functions introduced is smaller (usually much smaller) than the number removed. For example, consider replacing a helix containing 194 backbone and sidechain atoms and 561 bonded functions with a rigid cylinder. The group replaces 582 variables (three times the number of atoms) and 561 functions with 6 variables, representing position and orientation, and 30 functions, connecting the cylinder to the rest of the protein. If one-fourth of the functions in the protein model are defined within the helix, this grouping yields a twenty-five percent improvement in performance. Section 3 discusses adding and removing variables and constraints in greater detail.

Groups reduce the number of energy and constraint functions and the dimension of the constraint matrix. In general this linearly decreases the evaluation time of the functions and the solution time for the linear equations.

## 2. Requirements

### 2.1. When can atoms be grouped?

I place two requirements on atoms (elements) that are grouped. First, an atom can only reside in one group at a time. This prevents ambiguity about which group moves the atom. Second, atoms must be in an energy minimum when grouped. This prevents problems that can occur when the atoms are released. Without the second requirement, one could freeze atoms with a large repulsion among them by grouping them together. When the group is released later, the repulsion would cause the atoms to fly apart.

### 2.2. Group representation

The minimization module requires that a group perform the following four operations. First, a group must inform the minimizer how many variables define it. This lets the minimizer allocate internal arrays to accommodate the new set of variables. Second, a group must state the position of atoms within it. Third, a group must evaluate the derivative of those atom positions with respect to the group's variables. Fourth, a group must update its atom positions when the minimizer updates the group's variables. Rigid-body movement of an object simply maps the same movement to the atoms. For example, a translation of an object translates the atoms within it the same amount. However, the change in atom positions that results from a general deformation (e.g. twisting a helix or sheet) is not as obvious.

I implemented a rigid cylinder to represent the grouping of a helix. The cylinder axis is aligned along the helix axis, and the length and radius are set to those in the helix. The cylinder variables are a center and an orientation; the length and radius cannot vary. Additional representations are subject of future work. In particular a deformable cylinder could represent bending and twisting a helix, and a thin slab that twists like a piece of paper could represent changing a β-sheet. Implementing deformable models requires addressing two outstanding problems: deformations of geometric objects, which has been studied to some extent in [Barr 1984] and [Sederberg 1986], and mapping geometric deformations to valid changes in atom positions. Both subjects should offer fruitful research.

## 3. Groups change the set of variables and functions

### 3.1. Groups add new variables and functions

Groups introduce new variables to the protein model. The variables associated with a fixed-length cylinder, for example, are the location of the center and the orientation of its axis (the length and radius are initialized when created, but do not vary). The atom positions within the group are defined on these variables. Consider the position of atom *i* before and after it is grouped into a fixed-length cylinder. Before it is grouped, the position is $(x_{3i}, x_{3i+1}, x_{3i+2})$, assuming $x_{3k}$ represents the x-ordinate of atom *k*. After it is grouped, the rigid cylinder defines the atom position. The position is defined by the position and orientation of the cylinder plus a constant offset from a reference point within the cylinder (e.g. the center of the cylinder axis).

Groups also introduce new constraint and energy functions that connect the group to the protein. Assume a strand of backbone atoms, those between atoms *i* and *j*, are grouped and represented with a cylinder. Since atom *i* is now defined by new variables (the group's), a function defined with it and some atom outside the group now affects the position and orientation of the group. Functions defined with atoms within the group and atoms outside the group help keep a particular orientation of the group.

The minimization algorithm is not affected by the new variables and functions. Even though the discussion of variables prior to this chapter concerned only the position (cartesian coordinate) of atoms, the minimization algorithm is not restricted to them. The minimization algorithm in Chapter 3 is based on a vector of variables, $x$, and a set of functions defined on the variables. The minimizer uses derivatives of the functions with respect to the variables, regardless of what they represent.

I implemented a rigid cylinder to represent the grouping of a helix. The cylinder axis is aligned along the helix axis, and the length and radius are set to those in the helix. The cylinder variables are a center and an orientation; the length and radius cannot vary. Additional representations are subject of future work. In particular a deformable cylinder could represent bending and twisting a helix, and a thin slab that twists like a piece of paper could represent changing a β-sheet. Implementing deformable models requires addressing two outstanding problems: deformations of geometric objects, which has been studied to some extent in [Barr 1984] and [Sederberg 1986], and mapping geometric deformations to valid changes in atom positions. Both subjects should offer fruitful research.

## 3. Groups change the set of variables and functions

### 3.1. Groups add new variables and functions

Groups introduce new variables to the protein model. The variables associated with a fixed-length cylinder, for example, are the location of the center and the orientation of its axis (the length and radius are initialized when created, but do not vary). The atom positions within the group are defined on these variables. Consider the position of atom *i* before and after it is grouped into a fixed-length cylinder. Before it is grouped, the position is $(x_{3i}, x_{3i+1}, x_{3i+2})$, assuming $x_{3k}$ represents the x-ordinate of atom *k*. After it is grouped, the rigid cylinder defines the atom position. The position is defined by the position and orientation of the cylinder plus a constant offset from a reference point within the cylinder (e.g. the center of the cylinder axis).

Groups also introduce new constraint and energy functions that connect the group to the protein. Assume a strand of backbone atoms, those between atoms *i* and *j*, are grouped and represented with a cylinder. Since atom *i* is now defined by new variables (the group's), a function defined with it and some atom outside the group now affects the position and orientation of the group. Functions defined with atoms within the group and atoms outside the group help keep a particular orientation of the group.

The minimization algorithm is not affected by the new variables and functions. Even though the discussion of variables prior to this chapter concerned only the position (cartesian coordinate) of atoms, the minimization algorithm is not restricted to them. The minimization algorithm in Chapter 3 is based on a vector of variables, ***x***, and a set of functions defined on the variables. The minimizer uses derivatives of the functions with respect to the variables, regardless of what they represent.

### 3.2. Groups remove variables and functions

Groups eliminate variables and constraints. Because a group solely determines the position of its atoms, the variables representing the atoms' positions are discarded. All functions defined entirely on those atoms are also removed.

### 3.3. Which constraints and variables must be removed and inserted?

The structural hierarchy shows the variables and functions that a group removes. If a node is grouped (e.g. a residue), the variables of the atoms under it in the tree are removed. Each node also contains the functions defined on the atoms under it. For example, a backbone node contains the distance and angle functions defined among the peptide atoms (N, H, $C_\alpha$, C, O), and the sidechain node contains the functions defined on its atoms. The residue node contains the functions that connect the backbone and sidechain, and the protein node contains connections between the residues. Functions defined solely with atoms in the same group are removed.

The tree also provides the new group and functions. Each node contains the object that represents its elements when grouped (currently, a rigid cylinder). The functions that connect the group to the protein are modifications of those functions defined with variables both inside and outside the group. For example, consider the distance function modeling the bond between the carbon in one residue and the nitrogen in the next. If the atoms in the first residue are grouped together, the variables for the carbon atom are removed. The distance function, however, is not removed. Instead it is redefined to reference a point defined within the group (the carbon) and the original nitrogen. These redefinitions connect the group to the rest of the protein.

### 3.4. Changes to the constraint matrix

Adding and removing constraints changes the structure of the constraint matrix (the Jacobian transpose of the constraints). First, rows (constraints) and columns (variables) are eliminated as a result of the preceding discussion. Second, new variables representing a group are inserted. Third, rows representing redefined constraints are modified to refer to the new variables.

The sparse-matrix data structure presented in Chapter 5 facilitates fast insertion and deletion of rows and columns. The structure stores row and column indices in headers rather than in matrix elements to reduce the computation for inserting and removing rows and columns (see Figure 5.6).

In practice many groups contain continuous segments of backbone and sidechains (e.g. atoms in a helix reside in a continuous sequence of residues). The new variables are inserted into the gap left by removed variables. The number of variables in a group is smaller than the number of variables it replaces. This reduces the span of the rows in the Jacobian transpose and the number of operations in the matrix multiplication. Most importantly, groups eliminate constraints. This reduction yields fewer linear equations which in general reduces the computational bottleneck in *Sculpt*, the solution of linear equations.

The drawback of groups is possible increased bandwidth in the linear equations. The matrix bandwidth (defined in Chapter 5) increases linearly with the maximum number of constraints that reference any variable. This, in turn, increases with the cube of the number of distance constraints defined on any variable (see Section 2.1 of Chapter 5). In proteins the maximum number of distance constraints that reference the same variable is four—at most four atoms can bond to any atom. In groups, however, if *c* bonds connect to a group, then *c* length constraints reference the group's variables. The linear complexity analysis still holds when a group is added to the model (assuming the group is not connected to all the other atoms). However, the system performance may not improve if many atoms bond to the group and few constraints are removed from the model. Keep in mind that the computational complexity increases with the *square* of the bandwidth, which in turn increases with the *cube* of the number of connections.

Groups are commonly used to model the atoms in both the backbone and sidechains of a helix and the atoms in just the backbone of a helix. The first case only slightly increases the matrix bandwidth. Only atoms at the two ends of the helix connect to the group. At most three atoms not in the group can connect to each of the two ends. The second case can significantly increase the bandwidth. A helix can commonly have ten to twenty residues. Grouping only the backbone atoms requires connections for each of the sidechains and both ends. Chapter 9 lists performance results for both cases.

## 4. Non-bonded interactions

Eliminating all the non-bonded interactions among atoms that are grouped and those not grouped allows inaccuracies. This lets atoms in a group move arbitrarily close to other atoms in a protein. Assume the atoms in a helix are grouped and then moved into the middle of another structure. Without the van der Waals repulsion modeled, this is not prevented. When the atoms are freed, the van der Waals repulsion among the atoms in the helix and the other atoms explodes the protein.

*Sculpt* prevents this by treating non-bonded interactions the same as bonded interactions: those defined solely between atoms in the same group are removed, and those defined between an atom inside and an atom outside a group are calculated. Since the atoms in a group are defined on the group's variables, a non-bonded interaction with the atom actually affects the group rather than the atom. Using this approach, the previous problem is avoided. As one moves the helix near another atom, the atoms begin to repel one another. The repulsion is applied to the helix and thus prevents it from interpenetrating the other structure.

# Chapter 8
# System Description

This chapter describes the implementation of the *Sculpt* system. The chapter describes the user interface and graphical display presented to a user. The system implementation consists of three modules: user interface, display, and minimizer (as described in Chapter 3). The chapter outlines their communication and function. Implementation details particularly relevant to the research and the sculpting system are presented. The input file that specifies a protein and its bonds is described. The input file consists of a list of points representing atoms and connections among them. Though the primary application is protein sculpting, only a small percentage of the code and input *file* is specific to proteins. Chapter 9 presents a user session and performance results.

*Sculpt* runs on a Silicon Graphics 240-GTXB [Akeley 1988]. This machine contains four general-purpose, MIPS R3000 processors that run at 25 MHz. *Sculpt* uses shared memory for multi-processing communication. *Sculpt* mainly uses the vector-rendering capability of the machine. The machine renders 400,000, 10 pixel, depth-cued, z-buffered vectors per second [Akeley 1988]. The machine also renders transparent polygons using alpha blending [Porter 1984]. Timing results presented in this dissertation are based on this system. The system also runs on Silicon Graphics *XYO* architectures where *X* represents the two-, three-, or four-hundred family of processors and *Y* represents the number of processors.

## 1. User interface and display

### 1.1. Workstation configuration and basic display

The system displays covalent bonds with colored, depth-cued vectors. Figures 8.1 and 8.2 show photographs of the Felix protein displayed by *Sculpt*. Both figures show the backbone bonds with cyan vectors. Figure 8.2 also shows the sidechain bonds (gray vectors) and marks the sulfur atoms (yellow tetrahedrons). The system differentiates two bonds in each peptide along the backbone: the C-O bond is red, and the N-H bond is brown. Notice that the backbone winds through four helices in this protein. Hydrogen bonds in each helix are displayed as purple vectors connecting the oxygen of the C-O and the hydrogen of the N-H.

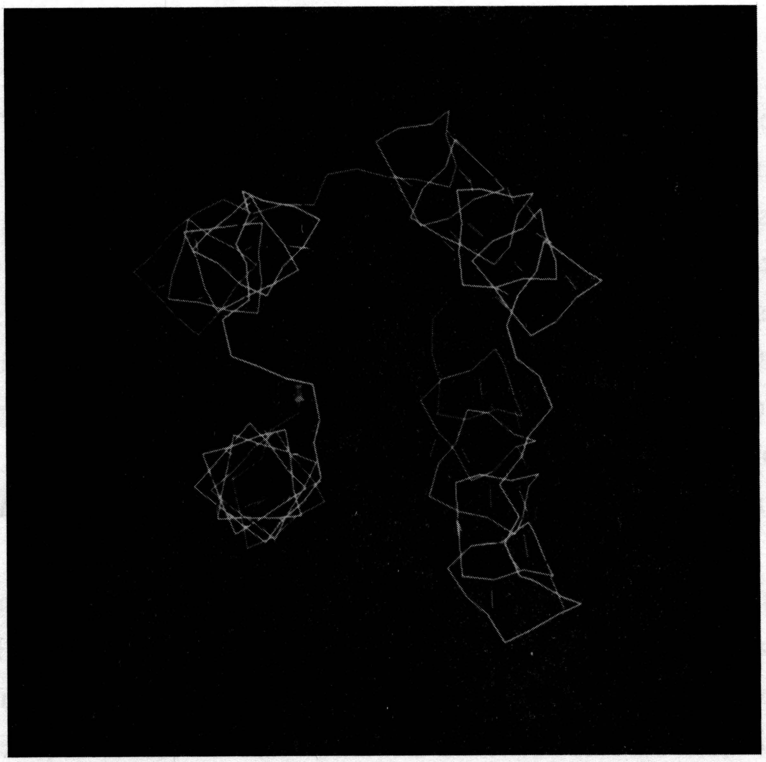


**Figure 8.1: Photo of *Sculpt* display using the Felix protein. Vectors represent bonds in the protein's backbone.**

The workstation consists of a liquid-crystal stereo plate attached to a monitor, a keyboard, a mouse, and a dialbox. The graphics are displayed in stereo to provide better three-dimensional cues. The dials rotate, translate, and scale the model. The mouse duplicates these transformations in case a dialbox is not available. Pop-up menus let a chemist toggle graphics parameters such as antialiasing, stereo, depth-cueing, and visibility of objects (e.g. sidechains and tetrahedrons). Menus also let a chemist toggle the modeling of non-bonded interactions and hydrogen bonds within the minimization.

### 1.2. Tugging atoms

A chemist moves an atom by first *picking* and then *tugging* it in a desired direction. Picking is done by placing the cursor over an atom and pressing the left mouse button. The system picks the atom nearest a ray shot beneath the cursor, perpendicular to the screen.

Pressing a particular key on the keyboard indicates that the pick begins an atom tug (other keys specify display of text or other items at the atom). Subsequent movement of the mouse moves the cursor in a plane parallel to the screen (rotating the model gives different planes through the model). This moves one end of a spring attached between the cursor and the atom (refer to Section 1 in Chapter 5). The system displays a gold coil that stretches along with the spring. The constrained energy minimizer runs as the chemist moves the spring. This moves the atoms, which are subsequently displayed.

When one releases the left mouse button during a tug, *Sculpt* leaves the spring attached to the current position, but no longer associates mouse movements with the spring. A gold-colored nail marks the position (refer to Figure 8.2). A chemist can now tug another atom. Releasing the key that enabled tugging removes all the springs. Figure 8.2 shows the previous model with multiple springs attached.

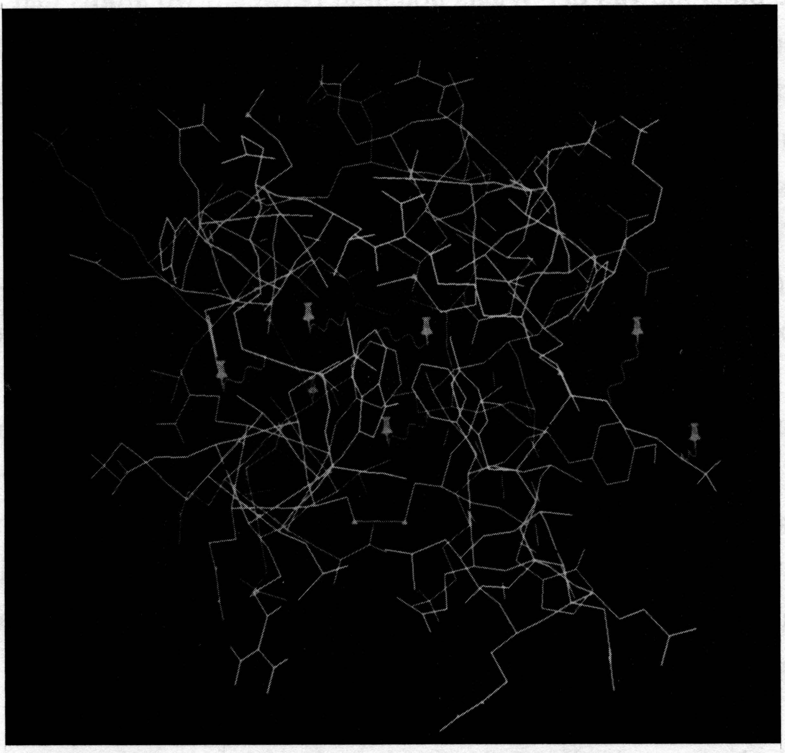


**Figure 8.2: Photo of *Sculpt* display with tugged atoms. Gold coils show tugs between atoms and fixed positions in space (marked with a thumbtack). Cyan vectors show backbone bonds; gray vectors show sidechain bonds.**

Often a user runs the minimizer for multiple steps without tugging an atom to a new position. This is done by either increasing the number of iterations taken in the minimization algorithm or by picking an atom and wiggling the cursor a small amount. In the latter approach *Sculpt* places a tug on the atom and runs the minimizer each time the cursor moves. Though the minimizer stays near a local constrained minimum, this operation lets the minimizer further settle into the minimum. This approach is most frequent when a user turns on van der Waals interactions after making drastic structural changes without the interactions modeled. The atoms move to reduce the strong repulsions as the user wiggles an atom. This is quite interesting to watch using the visualization described next.

### 1.3. Visualization of non-bonded interactions

Non-bonded interaction plays an important role in protein sculpting because a chemist typically wants tight-fitting contacts among internal sidechains. Unfortunately, non-bonded interaction is not as simple to display as a covalent bond. Figure 2.14 in Chapter 2 plots the van der Waals potential between two atoms that *Sculpt* models. The figure shows that an attractive (negative) potential energy appears between two atoms separated by 3.24 Ångstroms. As their separation decreases, the magnitude of the attraction increases nonlinearly until it reaches a maximum at an ideal separation. Further decreases in the separation increase the energy nonlinearly, but at a different rate than before. A small decrease from the ideal separation just diminishes the attraction; a greater decrease causes a repulsive (positive) energy. A useful display of non-bonded interactions should convey attractions, repulsions, their magnitudes, and the ideal separation.

My first attempt at showing non-bonded interactions displayed a sphere at each atom with its ideal (van der Waals) radius. I used wireframe spheres to reduce the occlusion introduced by the new objects. Intersecting spheres indicated repulsion and nearby spheres implied attraction. However, this technique did not indicate the magnitude of an attractive or repulsive energy. Also the spheres cluttered the display without significantly increasing the content.

*Sculpt* displays van der Waals interactions that have an energy magnitude greater than a user-defined threshold. A partial spherical shell is placed around both of the interacting atoms and aligned along a vector between them (see Figure 8.3). Currently a shell with a solid angle of $0.4\pi$ steradians (ten percent coverage) represents the weakest interaction. Solid angle increases with the magnitude of the interaction. Weak interactions are represented by dot spheres, and strong interactions are represented by wireframe spheres. A dot-sphere indicates that an interaction exists without distracting the user and consuming as much screen space as the wireframe sphere. Blue denotes attraction, and red denotes repulsion.

Figure 8.3 illustrates this visualization on a small model. Notice the wireframe shells around the two atoms labeled with text (one in the planar ring and the other in the backbone). Interpenetrating shells crush flat rather than intersecting so that the vectors in the two shells do not interfere visually. Intersecting wireframe shells are difficult to associate with their respective atoms.

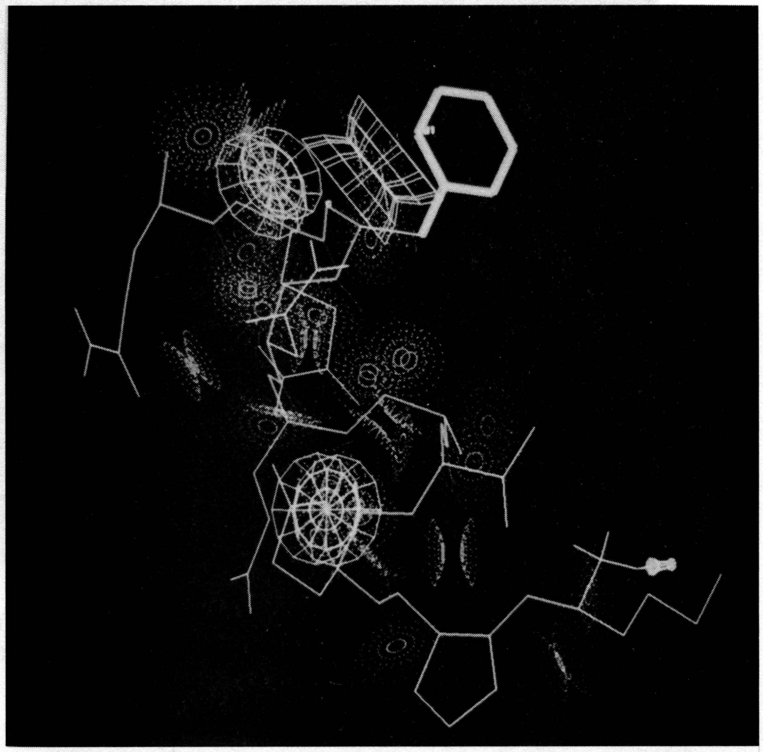


**Figure 8.3: Photo of *Sculpt* display with shells illustrating non-bonded interactions. Shell coverage increases with interaction strength. Blue denotes attraction and red denotes repulsion.**

## 1.4. Groups

The current version of *Sculpt* allows specification of groups only at initialization, through the input file discussed in Section 3. This reduced the development effort in the user interface, but future versions will allow on-the-fly creation of groups. The minimizer does handle groups and functions defined on atoms within groups. The sparse-matrix data structure correctly adds and removes rows and columns. Figure 8.4 shows two purple, translucent cylinders surrounding the backbone of the second and third helices in Felix. Each cylinder denotes a group of atoms. I use translucent objects to reduce occlusion of the model.

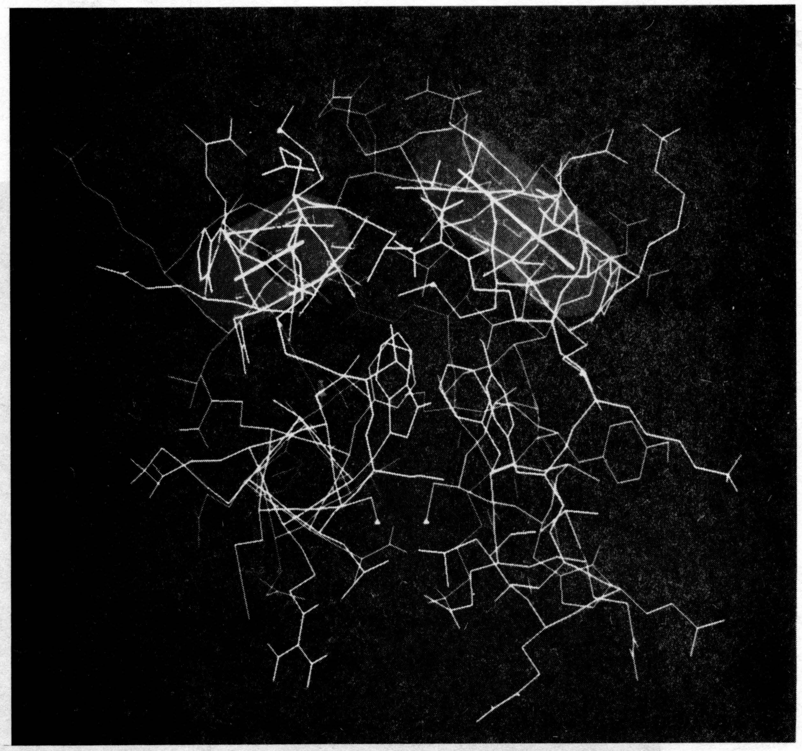


**Figure 8.4: Photo of *Sculpt* display with two helices grouped into rigid cylinders. A translucent purple cylinder represents each group.**

## 2. System structure

*Sculpt* contains display, user interface, and minimization modules. Figure 8.5 shows the main communication paths among the modules. The user interface module monitors user actions (e.g. view rotations or atom tugs). Most user actions modify display parameters. Beginning, ending, and moving a tug, however, cause the user interface to invoke the minimizer. A minimization then executes, and the minimizer module passes the new coordinates to the display module. The minimizer also adds and removes some graphical objects (e.g. shells). More details about the minimizer's implementation are given in Chapters 5 and 6.

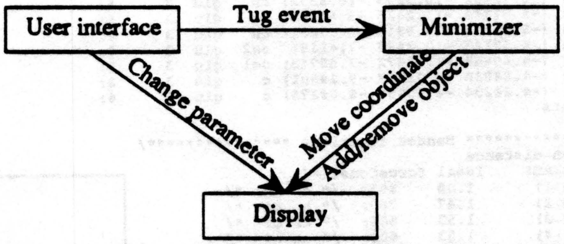


**Figure 8.5: System architecture contains three modules. Arrows show main communication paths between the modules.**

*Sculpt* is written in C++ version 2.0 [Stroustrup 1986]. The system contains approximately 28,500 lines of code divided as follows among the modules: 5,000 lines in the user interface, 5,700 lines in the display, 16,300 lines in the minimizer, and 1,500 lines in general-purpose routines. The only code specific to proteins or molecules consists of 1,000 lines of C++. The protein-specific code builds and traverses the hierarchical tree used in groups.

The minimizer uses a linear equation solver from the Harwell Sparse Matrix Library [Harwell 1988]. The package reduced system development effort and provides very stable equation solving. The linear equation package uses a direct Gaussian elimination with partial pivoting based on [Duff 1983]. Two routines are used to solve the linear equations. The first determines a pivot strategy based on the sparsity structure (location of nonzero elements) of the matrix. That routine is only called when the sparsity structure changes, which is at program initialization and group creation. The second routine uses this pivot strategy to solve the system of equations.

### 3. Input file

This section describes the input file that specifies the initial location of the points (atoms) and the topology of the model. The file lists the functions defined on the points. Each function contains an ideal value and an energy constant. Figure 8.6 shows an actual input file that specifies one glutamate residue (also drawn in the figure). An input file contains four parts: points, groups (not shown), bonded functions, and non-bonded functions. Only a small, optional section of the file is protein-specific. Text within /* ... */ are comments ignored by *Sculpt*. Each component in the file is described next.

```
points
/*  id:         (x y z)              atm   res res# class */
    0: (-5.36438 -8.06433 -11.4652) n     glu  3     0;
    1: (-5.17301 -8.60105 -12.287)  hn    glu  3     1;
    2: (-5.34536 -8.84922 -10.222)  ca    glu  3     2;
    3: (-5.12422 -10.3299 -10.5392) cb    glu  3     5;
    4: (-5.91854 -11.2423 -9.60156) cg    glu  3     6;
    5: (-5.58754 -10.9816 -8.13032) cd    glu  3     7;
    6: (-6.39143 -10.3493 -7.4115)  oe2   glu  3     8;
    7: (-4.49645 -11.3672 -7.65743) oe1   glu  3     9;
    8: (-4.24038 -8.34169 -9.29501) c     glu  3     3;
    9: (-4.24234 -8.60932 -8.08375) o     glu  3     4;
endpoints

/**************** Bonded functions ***************/
function distance
/* dist(AB)     ideal forceConst */
     (0 1)       1.00     895;  /* n  hn  */
     (0 2)       1.47     760;  /* n  ca  */
     (2 3)       1.53     600;  /* ca cb  */
     (3 4)       1.53     600;  /* cb cg  */
     (4 5)       1.53     600;  /* cg cd  */
     (5 6)       1.25    1300;  /* cd oe2 */
     (5 7)       1.25    1300;  /* cd oe1 */
     (2 8)       1.53     740;  /* ca c   */
     (8 9)       1.24    1390;  /* c  o   */
endfunction

function angle
/* angle(ABC)    ideal forceConst */
     (2 0 1)     1.989     88;  /* ca  n  hn  */
     (3 2 0)     1.920    112;  /* cb  ca n   */
     (8 2 0)     1.920    112;  /* c   ca n   */
     (8 2 3)     1.919    112;  /* c   ca cb  */
     (4 3 2)     1.955    112;  /* cg  cb ca  */
     (5 4 3)     1.955    115;  /* cd  cg cb  */
     (6 5 4)     2.094    120;  /* oe2 cd cg  */
     (7 5 4)     2.094    120;  /* oe1 cd cg  */
     (7 5 6)     2.094    120;  /* oe1 cd oe2 */
     (9 8 2)     2.112    124;  /* o   c  ca  */
endfunction

function dihedral   /* fixed dihedral angle */
/* angle(ABCD)  ideal forceConst mult */
     (7 5 4 6)  3.142   20.5       1; /* oe1 cd cg oe2 */
endkill

function dihedral   /* multiple dihedral angle */
/* angle(ABCD)  ideal forceConst mult */
     (0 2 3 4)  3.142   2.8        3;  /* n  ca cb cg  */
     (2 3 4 5)  3.142   2.8        3;  /* ca cb cg cd  */
     (3 4 5 7)  1.571   0.6        6;  /* cb cg cd oe1 */
endfunction

/************* Non-bonded interactions *********/
function waal
/* class     A           B  */
     0:  24.1284    635.4727; /* n   */
     1:   0.0000      0.0000; /* hn  */
     2:  54.6507   4140.3486; /* ca  */
     3:  23.6445    897.9717; /* c   */
     4:  23.2485    420.8784; /* o   */
     5:  46.6241   2905.5764; /* cb  */
     6:  46.6241   2905.5764; /* cg  */
     7:  23.6445    897.9717; /* cd  */
     8:  23.2485    421.3672; /* oe2 */
     9:  23.2485    421.3672; /* oe1 */
endfunction
```

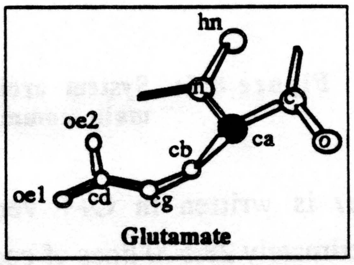


**Figure 8.6: Input file specifying atom positions and functions defined among them for a glutamate residue.**

### 3.1. Points

The first part of the input lists the points (atoms in this application) that represent the variables of the model. Each line within the *points ... endpoints* structure lists a different point. The format of the line is one of the three listed in Figure 8.7. The first line in the figure gives the minimal information: an identifier (usually an atom number) used for later reference and a three-dimensional cartesian coordinate. The second and third lines list protein-specific information: atom name, residue name, and residue number. This information is used to create the structural hierarchy and to display information about the point when requested. The third format in the figure contains an additional field that associates the atom with a class of atoms. Functions modeling non-bonded interaction (discussed in Section 3.3) use this classification.

```
id: (real real real);
id: (real real real) atom_name residue_name residue_number;
id: (real real real) atom_name residue_name residue_number atom_class;
```

**Figure 8.7: Three formats for a point in the *Sculpt* input.**

## 3.2. Bonded functions

Each line within a *function ... endfunction* structure lists the parameters of a specific function. Figure 8.8 lists the syntax for the bonded functions. Valid values of *function_type* for bonded functions are *distance, angle,* or *dihedral* angle. *Optional_name* identifies the set of functions for user reference (e.g. to turn off modeling or toggle between energy and constraint models). Each function is defined on the points given in *id_list* whose length depends on the function type (e.g. *id_list* contains *id1 id2 id3* if *function_type* is *angle*). The *ideal_value* and *spring_constant* fields are used for the mathematical model of the function. The system ignores *spring_constant* if the function is constrained.

```
function function_type name
    (id_list)  ideal_value spring_constant;
    (id_list)  ideal_value spring_constant;
          .
          .
          .
    (id_list)  ideal_value spring_constant;
endfunction
```

**Figure 8.8: Grammar for a set of bonded functions in the *Sculpt* input.**

### 3.2.1. Dihedral angle

The dihedral angle requires an additional field before the semicolon that specifies the multiplicity of the angle (number of ideal angles). The *ideal_value* serves as the first, or reference, ideal angle. Figure 8.6 separates the dihedral angles into the fixed (single-value) dihedral angles and the multiple dihedral angles.

### 3.2.2. Hydrogen bond

A hydrogen bond is specified with a distance and an angle function in the input file. Consider a hydrogen bond formed between the oxygen of a C-O bond and the hydrogen of a N-H bond. The input file requires a distance function defined on the *O* and *H* atoms and an angle function defined between the *O*, *N*, and *H* atoms. Figure 8.6 does not list hydrogen bonds because the residue does not contain any.

### 3.3. Non-bonded functions

Van der Waals interaction is specified in the input file with the *function waal ... endfunction* structure. Each line in the structure contains information for a class of atoms (refer to Section 3.1). The first field states the *atom_class* for the line. The remaining fields give constants used in the Lennard-Jones model (the *A* and *B* terms [Schulz 1979] used in most molecular modeling systems). The constants are intended for the 6-12 Lennard-Jones model. *Sculpt* converts them into parameters for its model of van der Waals interaction.

### 3.4. Groups

The input file contains a section, not shown in Figure 8.6, that specifies groups. The specification states the atoms in a group and the type of group that represents them. The left box in Figure 8.9 shows the grammar for a group; the right box shows one particular instantiation. The list *from_id to to_id* specifies the identifiers of the points in the group. The right box groups all the points between identifiers 220 and 343. The right box uses the cylinder to group the atoms. Each end of the cylinder is initialized by averaging the coordinates of the points referenced in *index(...)* (e.g. one endpoint is the average of the positions with identifiers 214, 224, 233, and 239).

```
group
     from_id1 to to_id1
     from_id2 to to_id2
                  .
                  .
                  .
     from_idN to to_idN

     into
          particular_group
endgroup
```

```
group
     220 to 343
     into
          cylinder
               index(214 224 233 239)
               index(317 326 335 348)
          endcylinder
endgroup
```

**Figure 8.9: The left box shows the grammar for specifying a group. The right box shows the atoms of a helix grouped into a cylinder.**

### 3.5. Summary of input

This input format is not specific to the protein application. This lets one model articulated figures with *Sculpt* by specifying a list of points with functions defined among them. This also lets a chemist choose the ideal values and energy constants used in the protein model. For more practical use with molecular modeling, a pre-processor is needed that transforms a common protein file format into *Sculpt's* input file format. Such a program could input a list of residues and the initial atom positions and output the functions describing the bond topology.

# Chapter 9
# User Session and System Performance

The chapter describes a modeling session in which a biochemist used *Sculpt* to make large structural changes to an existing protein model. The chemist at first used rigid bodies to move helices large distances and later modeled all the atoms (without rigid bodies) during subsequent fine-tuning. The chapter surveys the goals of the modeling operations and several attempted strategies for achieving them. The chapter presents the chemist's perceived contributions of *Sculpt* in molecular modeling; a separate paper [Surles 1992] is included in the Appendix that describes two other modeling sessions with *Sculpt* that are not as complex as the one described here. This chapter concludes with performance results based on data with and without rigid bodies from the session. The performance on several other data sets shows the linear decrease in performance as protein size increases.

## 1. A *Sculpt* session

Professors David C. and Jane S. Richardson of Duke University's Biochemistry Department originally conceived a system that would let a user interactively sculpt proteins. As members of my doctoral committee, they continually guided my research and *Sculpt's* development. They redesigned a protein model with a prototype version of *Sculpt*.

### 1.1. Problem

The problem involved large changes to the structure of the Felix protein. The Felix model contains seven hundred sixty atoms in seventy-nine residues that wind through four helices (refer to Figure 2.5 in Chapter 2 for a hand drawing of its backbone). The schematics in Figure 9.1 show end-views of the helices before (left), during (middle), and after (right) the session. The inner circle represents the backbone of the four helices (labeled A, B, C, and D). The line segments attached to the backbone illustrate the sidechains. For clarity the sidechains in the four helices are separated more than in the actual protein. The direction of the backbone is marked with arrow heads (•) and tails (X); a head means the backbone winds out of the page and a tail means the backbone winds into the page. The backbone begins in the page at the bottom of helix A, winds out of the page in helix A and crosses to helix B. This continues until the backbone winds into the page in helix D.

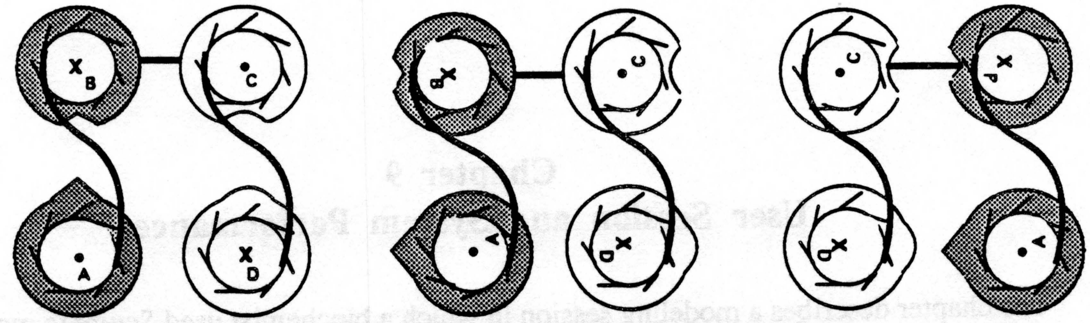


**Figure 9.1: Change in helix orientations during session.**

The pointed and rounded divots and extrusions indicate the orientation change between the initial and final models. Three operations change the model from the original to the final orientation. First, unwind (counter clockwise) helix A ninety degrees and wind (clockwise) helix B ninety degrees (similar to unrolling a scroll). Second, unwind (counter clockwise) helix C ninety degrees and wind (clockwise) helix D ninety degrees. The middle figure shows this intermediate stage. Third, translate helices C and D to the left. The third step reverses the left and right pairs of helices.

Three additional restrictions are placed on the structural changes allowed in the helices. The main helical structures cannot change. The backbone at the ends of the helices must have valid bond geometry. The sidechains between helices must have tight, but valid, packing.

The reason such large changes are needed is that in doing *de novo* design of proteins it is just as important to do negative design that avoids major alternative structures as it is to do positive design for the desired arrangement. The transformation done here is between the two major alternative arrangements of a four-helix bundle such as Felix.

### 1.2. Attempted solutions

Professor Jane S. Richardson (subsequently referred to as JSR) tried to solve this modeling problem with these four strategies: (1) model all the atoms using only atom tugs to change the structure, (2) first rotate the helices with rigid bodies and then model all the atoms, (3) automatically rotate the helices and let *Sculpt* automatically move the atoms so that the constraints are not violated, and (4) disconnect the segments between the helices before automatically rotating them and then let JSR reattach the segments while *Sculpt* maintains valid properties within the segments. The final approach worked the best. The remainder of this section describes in greater detail the four strategies and their problems. It concludes with photographs of the resulting structure.

### 1.2.1. Model all atoms throughout the session

JSR first tried to rotate each helix by tugging individual atoms. She rotated two of the helices in approximately one hour. However, this approach had two drawbacks. First, she often changed the conformation of a helix by tugging an atom too much. Second, the method was frustratingly slow. During this session she used a model that only contained two helices. On the full Felix model *Sculpt* requires approximately 1.2 seconds per update. Given enough patience JSR probably could have completed the entire modeling task using this strategy. However, we decided to try strategies that rotated the helices as rigid structures, both for convenience and to keep their geometry ideal.

### 1.2.2. Rotate helices with the rigid bodies

With the second strategy JSR modeled the atoms in each helix with a rigid body and explicitly modeled the atoms in segments connecting the helices. There was very little slack in the segment connecting the helices, so she rotated each helix by approximately ten degrees at a time (similar to unrolling a scroll).

We believe the rigid body strategy failed for two reasons. First, helix B and C were overconstrained. Both of the helices had constraints defined on bonds that connected to the helix at the top and at the bottom. A helix would oscillate back and forth when JSR tried turning it. Eventually, the helix would turn easily. We believe the oscillation shifted residues in the segments attached at each end until a degree of freedom appeared. Sometimes, when a helix eventually moved freely, it did not go in quite the intended direction.

The second reason for the failure of rigid bodies is a poor implementation. *Sculpt* ignored some of the constraints on atoms bonded to the rigid body. JSR would manually move the atoms so that the constraints were satisfied, but in subsequent iterations *Sculpt* would violate them.

### 1.2.3. Automatic repair after automatic rotations

In the final two strategies JSR let *Sculpt* initially move each helix to its goal position and orientation without modeling any constraints or energies! After this automatic step, many of the bonds and angles were extremely far from their ideal value. The problem now remained of how to adjust the residues in the connecting segments and in the first and last turn of each helix so that the model was physically realistic.

JSR first tried unsuccessfully to see if *Sculpt* would automatically move the atoms into positions that satisfied the constraints. However, the initial state of the model violated a principle assumption from Chapter 3—the constraints are satisfied, or nearly satisfied, at the start of each constrained minimization. *Sculpt* tended to distribute the large error in a few constraints into small errors in all the constraints. *Sculpt* never did bring the model back to a state with all the constraints satisfied.

### 1.2.4. Manual repair after automatic rotations

The strategy that ultimately succeeded used the manual positioning strategy (Section 1.2.1) after an initial, automatic movement of the helices (Section 1.2.3). We divided the model into four pieces: helix A with the connecting segment from A to B, helix B, helix C with the segments from B to C and from C to D, and helix D. *Sculpt* automatically positioned each of the four pieces. JSR tugged each of the three segments back to join its unconnected helix (e.g. moved the segment between A and B back to join B). During this part of the session, *Sculpt* modeled the constraints and energies in the segment and kept the helices fixed in space. Once JSR believed the segment end was close enough to the unconnected helix, she had *Sculpt* insert springs that pulled the final peptide into its proper position in the helix. Positioning each segment required approximately an hour and ended with a reasonably satisfactory model.

Next we combined the coordinates from the four pieces and ran *Sculpt* on the new Felix model. JSR repositioned many internal sidechains before turning on the van der Waals interactions. At that stage the model contained hundreds of atoms with overlapping electron shells. JSR let *Sculpt* resolve these contacts in batch mode. The session resulted in a complete three-dimensional model of Felix in this alternative folding pattern. This new model can now be used as the starting point for redesign.

## 1.3. The resulting model

Figures 9.2 and 9.3 show photographs of the Felix backbone before and after the session. The helices are numbered in clockwise order beginning in the lower-left corner in Figure 9.2 and in counter-clockwise order beginning in the lower-right corner in Figure 9.3. I use several visual cues to emphasize the changes between the figures (both use the same graphical parameters). Helices A, B, C, and D are respectively colored orange, green, blue, and magenta. The residues in the connecting segments are colored yellow. The residue name of sidechains that interact with another helix is placed at the first atom in the sidechain (the $C_\beta$ atom). White text labels sidechains in the initial model that interact between helix A and B and between helix C and D; orange text labels those that interact between helix A and D and between helix B and C. Finally, the cystine residues in helices A and D are highlighted with white tubes.

Two aspects of Figure 9.3 are notable. First, the interior (hydrophobic) sidechains remain in the interior but interact with different helices. Second, the yellow segment between each helix in the original model shifted position by one residue in the new model. The segment from helix A to B moved one residue out of A and into B, the segment from B to C moved into C, and the segment from C to D moved into D.

Figure 9.4 shows a side view of the resulting model with the van der Waals repulsive energies highlighted with red shells. The internal sidechain packing does not have overlapping electron shells. The only overlaps occur in the backbone turns entering and leaving the helices. The current model will now allow additional research in protein design. JSR can use this model as a basis to determine if other residues fit better in these new turns. JSR will now use the model both for negative design (i.e. to document the ways in which the Felix sequence suits the original model better than it suits this one) and also for designing minimal changes in the sequence that should make it prefer to fold into this alternative structure.

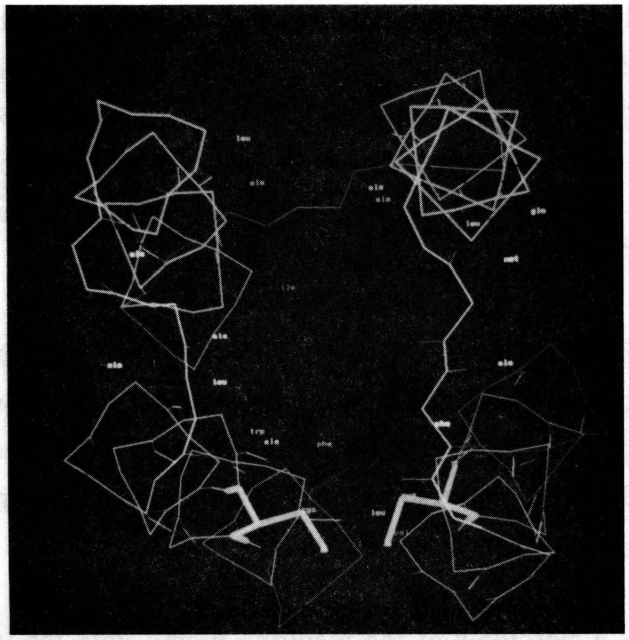


**Figure 9.2: Photograph of Felix backbone before modeling session.**

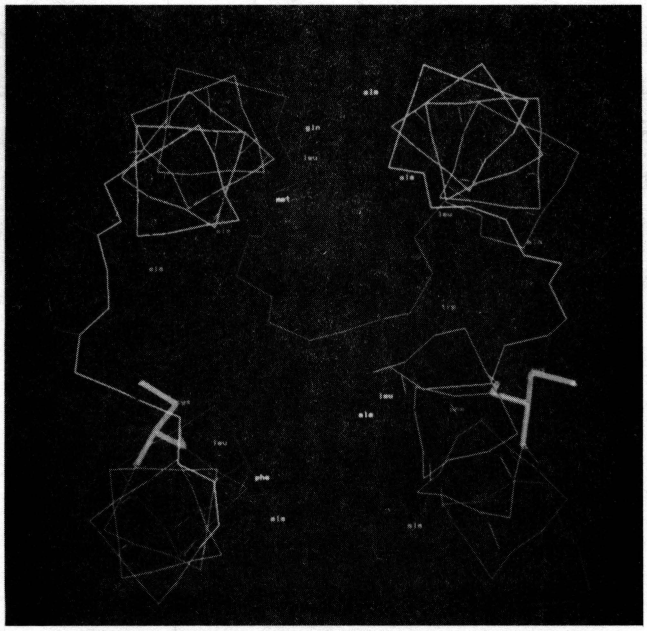


**Figure 9.3: Photograph of Felix backbone after modeling session.**

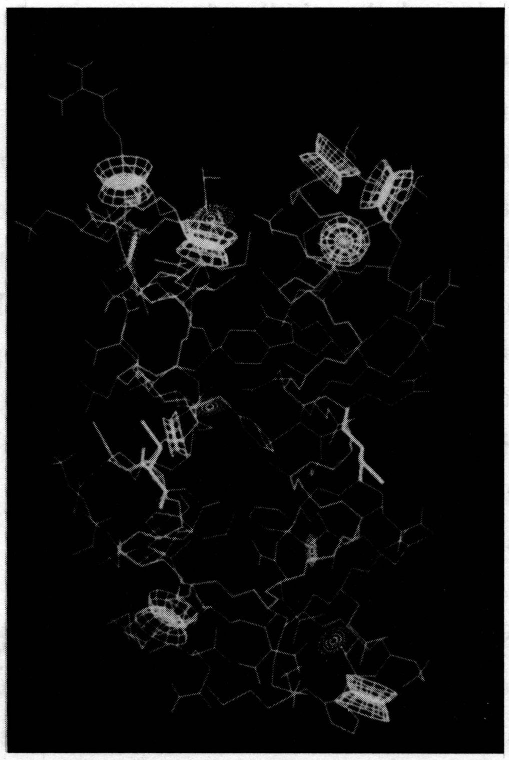


**Figure 9.4: Photograph showing atom repulsions after session.**

## 1.4. Comments

### 1.4.1. Pros

JSR believes solving this modeling task with *Sculpt* is significantly easier and faster than with interactive modeling systems that only allow rotations about (φ,ψ) angles. Such interactive systems let her turn the helices in one of two ways. The first way requires choosing appropriate rotation angles between segments. This is an extremely complex, inverse-kinematics problem involving hundreds of joints. The second way involves breaking the backbone connection between each helix, rotating each independently, and rejoining the connections. Rejoining the connection with proper geometry is very difficult, though easier than the inverse-kinematics problem. Once JSR turns the two helices she must then resolve hundreds of contacts among sidechain atoms. JSR tried solving this task manually but quit after several frustrating days and was never fully satisfied with the results.

JSR also believes *Sculpt* provides more control over atom movements than batch minimizers. With a batch minimizer JSR could solve this modeling task by specifying target positions for some of the atoms and running an energy minimization. The minimizer moves the atoms along a path towards the targets. However, certain paths can tear the model apart (e.g. through the middle of a structure). Instead of solving the problem with one set of targets and one minimization, JSR would choose subgoals along a path to the target and run a minimization with each subgoal. This approach requires less effort than the manual solution, but the turnaround time for a subgoal minimization limits the number of steps picked along the path. Whenever she has tried this process, it has resulted in distortions of helix geometry that are beyond convergence distance for the final structure, and in sidechains getting caught in local energy minima that are not natural. The sidechains can be manually fixed by moving to a geometric modeling program, but the backbone could not be fixed that easily. *Sculpt* takes the successive subgoal minimization to the extreme by continually running minimizations as JSR moves atoms to their targets. It also combines the interactive geometrical manipulation with the energy minimization approach in a single integrated process.

The graphical visualization of non-bonded interactions helped JSR identify close contacts among atoms and evaluate improvements made by moving atoms. When JSR turned on the van der Waals interactions after rotating the helices, she saw large red shells in the protein interior that illustrated strong atom repulsions. She then tugged atoms in that region to relieve the strain. As she tugged the atoms, the size and number of shells diminished, indicating more favorable contacts.

Interactive modeling of physical properties (e.g. non-bonded interactions) combines benefits from batch simulations with features from interactive graphics. Chemists use interactive graphics to study a static structure or series of structures from either experimental data or pre-computed simulations. Interactively controlling the view and display parameters provides more cues about a molecule's structure and nature than does viewing multiple, static images. Guiding an interactive simulation while immediately viewing the results lets a user remain continually engaged in the modeling process. JSR believes this provides greater situational awareness of complex relationships within a model than viewing cine loops of simulations. This may improve perception of subtle relationships within proteins. Guiding an interactive simulation lets a user stumble upon unexpected reactions in the model that may go unnoticed in batch simulations (the Aha! phenomenon). On several occasions JSR saw unexpected reactions that, upon closer examination, resulted from non-bonded interactions competing against other properties such as bond rotations. She believes a mature *Sculpt* system will actually help researchers gain an intuitive understanding of how molecules behave rather than just assist in a succession of individual modeling tasks. JSR also believes with shortened turnaround time, more users will experiment with protein models.

### 1.4.2. Cons

JSR's main complaint regarded the user interface. Rotating each helix when all the atoms were modeled required numerous tugs tangent to the helix backbone. She let the minimizer rotate the helix some and then replaced the tugs with ones pointing in new directions. She believes a high-level specification of user actions will relieve much of this burden. For example, turning a knob could require that the system place the tangential tugs for her. Similarly, the rigid bodies did not improve the user interface the way we originally thought it would. This drawback is probably more a fault of the primitive implementation of rigid bodies than an inherent flaw with the concept. With faster computers in the future, *Sculpt* may be able to model even more realistic behavior by including electrostatic interactions and hydrogen bonds.

One advantage a batch simulation, viewed with cine loops, has over an interactive simulation is the ability to replay the simulation. Since a cine loop is a sequence of frames, a user can easily move backwards in the sequence to study a particular property. Unless *Sculpt* saves all user actions, a user cannot readily return to a previous state. Like an on-going laboratory experiment, an event cannot be repeated without re-running the experiment from the beginning with the same steps.

## 2. Performance without rigid bodies

The following performance analysis uses four protein models without rigid bodies. The first table in Figure 9.5 lists the number of bonded and non-bonded interactions in the four models. Data sets (1) and (2) contain the number of length and angle springs modeling hydrogen bonds in the *H-bond springs* column. The second table summarizes the number of energy functions, with and without the near-neighbor interactions, and constraint functions. All data sets model bond lengths, bond angles, and fixed-value dihedral angles with constraints and model multi-value dihedral angles with energy functions. The models include (1) the Felix protein used in the user session, (2) two of the four helices from Felix, (3) a segment with ten residues (the approximate size of the segments in the modeling session), and (4) a small segment with four residues. The variable field equals the number of atoms times three (a three-dimensional coordinate).

| Model | Atoms | Variables | Lengths | Angles | Fixed dihedrals | Multiple dihedrals | H-bond springs | Neighbor interactions |
|---|---|---|---|---|---|---|---|---|
| 1 | 760 | 2280 | 770 | 1105 | 330 | 322 | 106 | 7601 |
| 2 | 355 | 1065 | 359 | 516 | 152 | 150 | 48 | 3267 |
| 3 | 99 | 297 | 100 | 142 | 40 | 43 | 0 | 745 |
| 4 | 36 | 108 | 36 | 51 | 9 | 18 | 0 | 198 |

| Model | Variables | Constraints | Bonded energies | Total energies |
|---|---|---|---|---|
| 1 | 2280 | 2205 | 428 | 8029 |
| 2 | 1065 | 1027 | 198 | 3465 |
| 3 | 297 | 282 | 43 | 788 |
| 4 | 108 | 96 | 18 | 216 |

**Figure 9.5: Statistics for the four models used in performance analysis.**

I ran the performance analysis on a Silicon Graphics 240-GTXB [Akeley 1988]. The machine contains four general-purpose, MIPS R3000 processors that run at 25 MHz. All calculations use double-precision floating-point arithmetic. Figures 9.6 and 9.7 show performance results with the four protein models. The performance (seconds per update) includes the time to receive a user tug, run a constrained minimization, and re-display the screen (single-precision provides faster performance, but less accurate results) . I list the performance for simulations with and without the near-neighbor interactions modeled. The performance results for simulations without near-neighbor interactions modeled are given for one and four processors. The performance results with near-neighbor interactions use four processors. The code that determines the list of neighbors for each atom is new and not optimized (e.g. creating each list requires inserting nodes into a linked list of atoms rather than a fixed length array). Therefore, I split the performance results for the near-neighbor interactions into two categories. The first uses the same neighbor list throughout a session while the second computes a new list on each iteration.

| | Without near-neighbor interactions | | With near-neighbor interactions (4 CPUs) | |
|---|---|---|---|---|
| Model | 1 processor | 4 processors | Same list | New list |
| 1 | 1.405 | 0.954 | 1.228 | 1.603 |
| 2 | 0.586 | 0.396 | 0.514 | 0.689 |
| 3 | 0.147 | 0.105 | 0.126 | 0.169 |
| 4 | 0.045 | 0.048 | 0.047 | 0.054 |

**Figure 9.6: Performance (seconds per update) with four models using an SGI 240-GTX.**

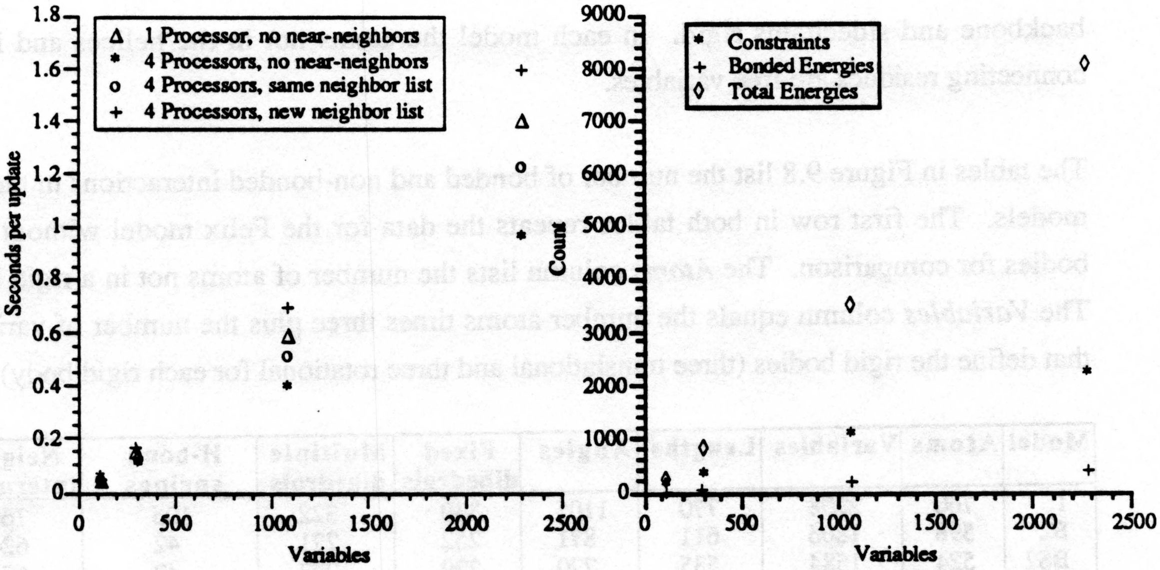


**Figure 9.7: Left plot shows linear increase in compute-time with model size; right plot shows linear increase in the number of constraints and energies with model size.**

Most of the computationally-intensive routines run on four processors. The bonded energy and constraint functions (length, angle, and dihedral angle) and their derivatives are evaluated on four processors. Matrix-matrix and matrix-vector multiplication also execute on four processors. Only one processor solves the system of linear equations. Currently only one processor creates neighbor lists and evaluates non-bonded interaction energy; this can be parallelized.

The update rates, though far from those needed for smooth interaction, allow productive new research in biochemistry. The bottleneck in the minimization algorithm is the solution of linear equations. With four processors, approximately fifty percent of the computation time in model (1) is used solving the linear equations. Chapter 10 describes two areas of future research that may yield significant improvement in this performance.

## 3. Performance with rigid bodies

I used four models with rigid bodies for this performance analysis; two contained four rigid bodies and two contained two rigid bodies. The Felix model (1) discussed in Section 2 is the base model for this analysis. Models (B2) and (BS2) had two of the four helices in Felix rigid (helices 2 and 3). (B2) held only the backbone of the two helices rigid, and (BS2) held the backbone and sidechains of the two helices rigid. Models (B4) and (BS4) had all four of the helices rigid. Again, (B4) held only the backbone and (BS4) held the backbone and sidechains rigid. In each model the atoms not in the helices and in the connecting residues are free variables.

The tables in Figure 9.8 list the number of bonded and non-bonded interactions in the four models. The first row in both tables repeats the data for the Felix model without rigid bodies for comparison. The *Atoms* column lists the number of atoms not in a rigid body. The *Variables* column equals the number atoms times three plus the number of variables that define the rigid bodies (three translational and three rotational for each rigid body).

| Model | Atoms | Variables | Lengths | Angles | Fixed dihedrals | Multiple dihedrals | H-bond springs | Neighbor interactions |
|---|---|---|---|---|---|---|---|---|
| 1 | 760 | 2208 | 770 | 1105 | 330 | 322 | 106 | 7601 |
| B2 | 598 | 1806 | 611 | 871 | 252 | 271 | 42 | 6249 |
| BS2 | 524 | 1584 | 535 | 770 | 229 | 227 | 42 | 5754 |
| B4 | 426 | 1302 | 441 | 621 | 168 | 219 | 0 | 5101 |
| BS4 | 234 | 726 | 241 | 352 | 91 | 121 | 0 | 3431 |

|  |  |  |  | Near neighbors |  |  |
|---|---|---|---|---|---|---|
| Model | Variables | Constraints | Bonded energies | In list | Computed | Total energies |
| 1 | 2280 | 2205 | 428 | 7601 | 7601 | 8029 |
| B2 | 1806 | 1734 | 313 | 7601 | 6249 | 6562 |
| BS2 | 1584 | 1534 | 269 | 7601 | 5754 | 6023 |
| B4 | 1302 | 1230 | 219 | 7601 | 5101 | 5320 |
| BS4 | 726 | 684 | 121 | 7601 | 3431 | 3552 |

**Figure 9.8: Statistics for four models containing rigid bodies.**

*Sculpt* does not calculate the van der Waals interaction energy between neighboring atoms in the same rigid body. However, the current implementation of the algorithm that creates the neighbor lists does not take advantage of this. The algorithm determines all neighboring atoms even if they are in the same rigid body. The second table in Figure 9.8 gives the number of neighbors placed in the lists (the *In list* column). Notice the number does not decrease as the number of grouped atoms increases. When *Sculpt* computes the van der Waals interaction energy neighboring atoms in the same rigid body are removed. This happens on each iteration, even if the same neighbor list is used for multiple iterations. The *Computed* column lists the number of van der Waals interactions actually computed on each iteration.

Figures 9.9 and 9.10 show performance results with the four rigid body models and the Felix model for comparison. The performance with the near-neighbor interactions is poorer than necessary since the number of neighbors does not decrease as the number of grouped atoms increases. One processor transforms (updates) the position of each grouped atom according to the change in the group's translation and orientation.

| | Without near-neighbor interactions | | With near-neighbor interactions (4 CPUs) | |
|---|---|---|---|---|
| Model | 1 processor | 4 processors | Same list | New list |
| 1 | 1.405 | 0.954 | 1.228 | 1.603 |
| B2 | 1.164 | 0.851 | 1.169 | 1.511 |
| BS2 | 1.945 | 0.712 | 1.002 | 1.368 |
| B4 | 1.035 | 0.742 | 1.078 | 1.440 |
| BS4 | 0.460 | 0.334 | 0.644 | 0.995 |

**Figure 9.9: Performance (seconds per update) with Felix and four models containing rigid bodies.**

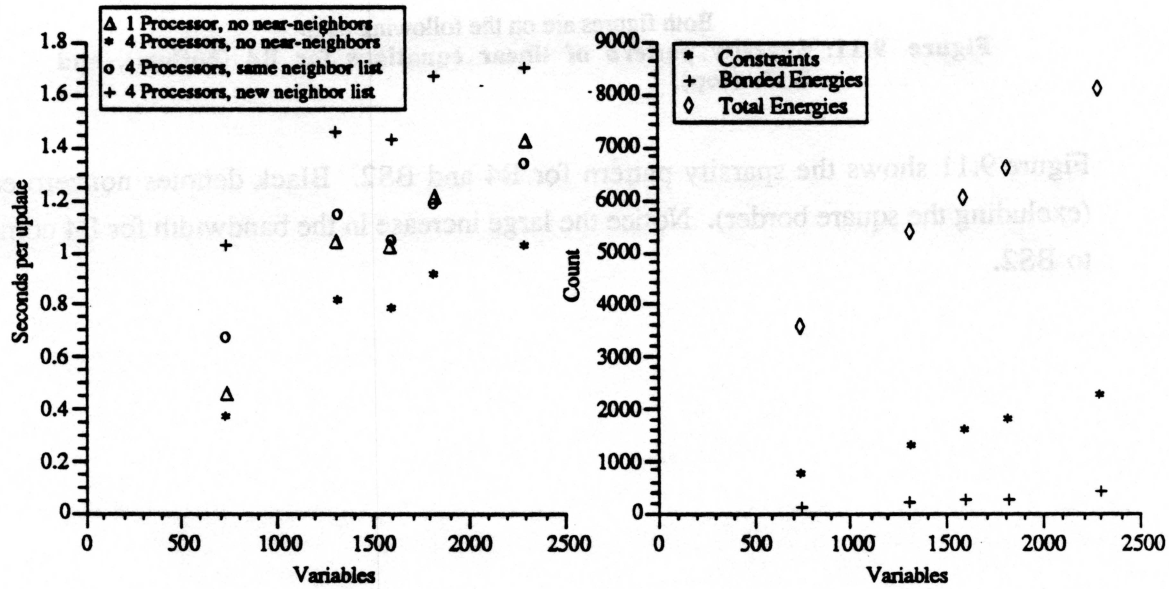


**Figure 9.10: Left plot shows performance with rigid bodies; right plot shows the number of constraints and energies for each model.**

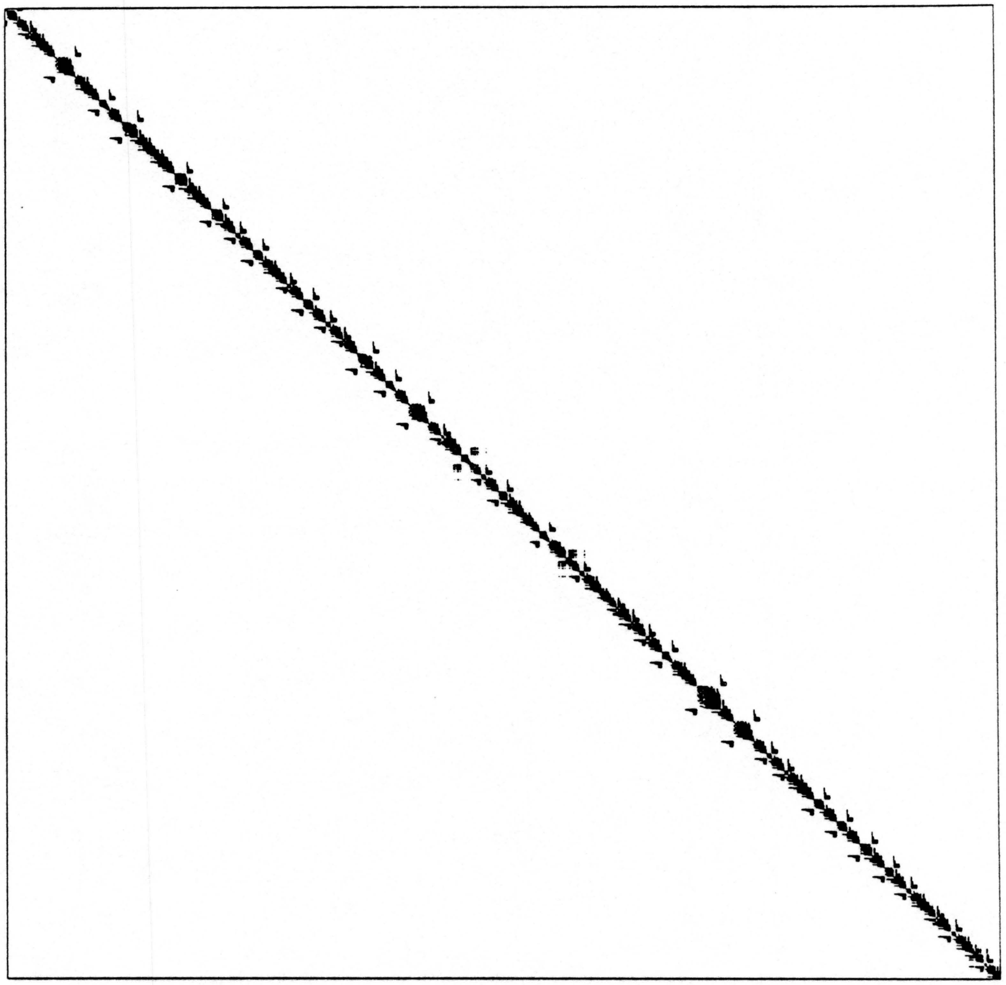
The performance is not solely dependent on the number of variables in these models. Another factor, the matrix bandwidth, is also different. Section 3.4 in Chapter 7 states that groups can increase the bandwidth of the linear equations. Also, rigid bodies that have many connections have a larger bandwidth than rigid bodies with few connections. Models BS2 and BS4 have groups with bonds connecting only at each end of the helices. Models B2 and B4, however, also have bonds connecting each sidechain. The bandwidth for BS2 and BS4, therefore, is smaller than the bandwidth for B2 and B4.

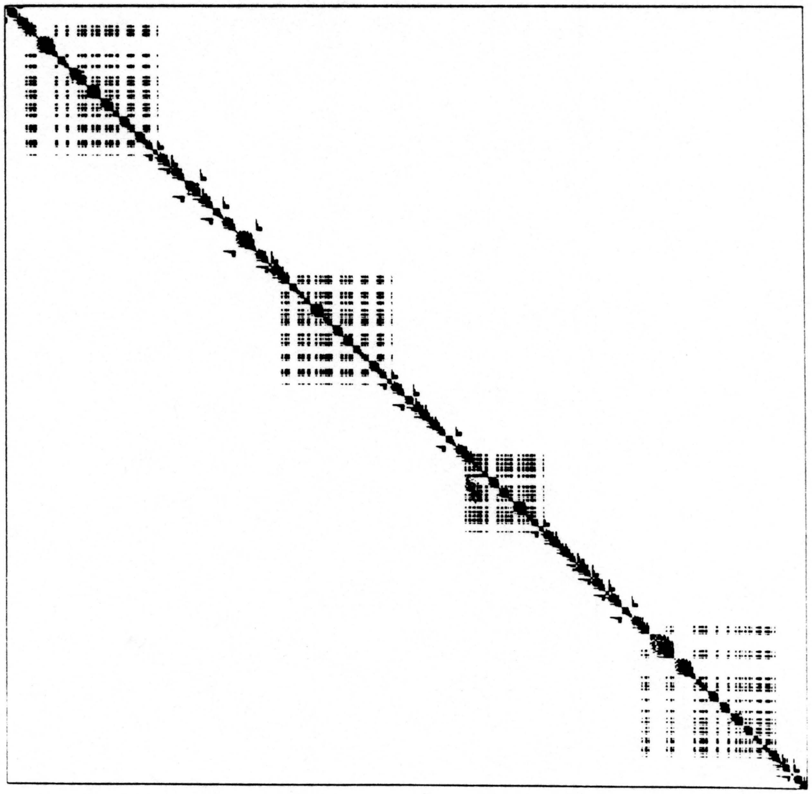
Compare the performance for models B4 and BS2 (the second and third columns from the left in the two plots). B4 requires more computation than BS2 even though it contains fewer energy functions, constraint functions, and variables! More than fifty percent of the computation with both models is spent solving the system of equations. B4 has 1302 equations and BS2 has 1537 equations, an increase of 235. However, the bandwidth of B4 is 205 and the bandwidth of BS2 is 55, a decrease of 150. The performance is *linearly* proportional to the number of equations and *quadratically* proportional to the bandwidth. In this comparison, the increase in the bandwidth outweighs the decrease in the number of equations.

**Both figures are on the following page**

**Figure 9.11: Sparsity pattern of linear equations for B4 (bottom), and BS2 (top).**

Figure 9.11 shows the sparsity pattern for B4 and BS2. Black denotes nonzero entries (excluding the square border). Notice the large increase in the bandwidth for B4 compared to BS2.





# Chapter 10
# Future Work

The immediate goal for future work is to place *Sculpt* in a chemistry lab and gather results about its usefulness for solving daily protein-modeling problems. This should offer the best direction for future enhancements to the modeling, visualization, and user-interface components of the system. Below I describe five areas known to be ripe for future research.

## 1. Enhanced molecular model

*Sculpt* does not model electrostatic interaction, formation of hydrogen bonds, or interaction with solution. Future versions of the system should address approximate models for each of these, in order to offer interactivity to trade for accuracy. Current workstations are too slow to calculate electrostatic interactions over the commonly used ten Ångstrom radius. However, applying electrostatic interactions to atoms in the shorter van der Waals neighborhood will improve the current model and not drastically degrade performance. The hydrogen bonds are specified in the input file at initialization; a more accurate model lets hydrogen bonds form and break as atoms move into each other's neighborhood. Surrounding solvent helps hold a protein together. I have not noticed a protein model drift apart even though *Sculpt* does not model solution. If the lack of solution becomes a problem, a possible approximation models a weak gravitational pull on all the atoms towards the center. At least supercomputer processing speed is necessary to model all the interactions with surrounding solution.

Generalizing *Sculpt* to model molecular structures other than proteins should not require much effort. A protein is described in *Sculpt* as a list of points with length and angle functions defined on them; only the hierarchical model used to ease group definitions is protein specific. The current performance on small proteins indicates that *Sculpt* can interactively model the mechanics of drugs and nanomachines.

### 2. Improved performance

The bottleneck in the minimization algorithm is the solution of linear equations. With four processors, approximately fifty percent of the computation time with the Felix model is used in the solution of linear equations. Two approaches can significantly reduce this bottleneck. First, an iterative method, such as Gauss-Seidel or Conjugate Gradient [Luenberger 1973], may converge to a solution much faster than direct Gaussian elimination, since each time-step starts with an excellent initial approximation. Iterative methods can reduce the average computation but cannot reduce the linear complexity. Second, the constrained minimization can use a different Lagrange multiplier estimation that does not require solving a system of equations. A zero-order method estimates the Lagrange multipliers using previous values of the constraints rather than the first-order information contained in the constraint Jacobian [Gill 1981]. A zero-order estimator, in general, does not make as accurate a prediction as the first-order method, so the constrained minimization algorithm requires more iterations. However, the estimation can execute in parallel. A constrained minimizer using a zero-order estimator may find a solution significantly faster on a massively-parallel architecture than does the current implementation.

### 3. Deformable models

Extending the groups in *Sculpt* to deformable (rather than rigid) objects may ease specification of complex modeling operations. For example, consider the necessary operations for twisting the backbone of a helix using the current system. A user must tug each atom in a different direction, tangential to the backbone, by a different amount. The concept of a twist is simple, but the application is cumbersome. If a user applied a pre-defined twist operation to a deformable coil, *Sculpt* could map this to a twist of the atoms. Properly implementing deformable models for this purpose requires research in three areas. First, a simple parameterization of geometric deformations is needed; for example, a method is needed to easily specify a bend or twist of a coil or sheet of paper. Second, a deformation of a geometric object must move atoms without violating constraints and must maintain a local energy minimum; bending a helix should not overlap atoms near the center of the bend. Third, a user-interface is needed for expressing common deformations, such as bend and twist, to different geometric objects.

### 4. User interface and graphical visualization

*Sculpt's* user interface needs major development. Specific areas for enhancements will arise as chemists use the system more. Five needed extensions are the following:

transforming common molecular input format into *Sculpt's* format, inserting springs between two atoms, changing the stiffness of user springs, indicating protein properties with the highest strain energy, and model editing that lets a user insert or remove residues.

The near-neighbor visualization discussed in Chapter 8 is adequate, though not great. Further research into visualization of near-neighbor interactions is needed. Professor Richardson wants *Sculpt* to display more information about neighbor interactions in the region of user tugs and less information in the remainder of the model. This may provide better information about the sidechain interactions and let a chemist better pack the protein interior. A much harder property to visualize is long-distance, electrostatic interaction. These interactions can extend between atoms on opposite sides of a molecule. The shells used in the near-neighbor visualization will not work for long-distance interactions.

## 5. Other applications

The algorithms used in *Sculpt* are applicable to other applications. Some of the benefits discussed in Chapter 9, such as removing the task of model repair after an interactive modeling session, will likely arise in other interactive applications that incorporate physically-based modeling. This section examines one application that has the same properties as protein sculpting and another interactive application that may benefit from incorporating physically-based modeling.

### 5.1. Skeletal figure animation

*Sculpt* should work well for animation of skeletal figures. Under one thousand lines of code is specific to molecules. *Sculpt's* input is a list of points with a set of length and angle functions defined on the points. Two important model assumptions allow linear computational complexity: there are fixed bounds, independent of model size, on the number of (1) constraints that reference any one point and (2) joints in any chain attached to the central backbone. A vertebrate whose backbone has many more joints than does any limb meets these criteria. In a separate paper I discuss the same analysis from Chapter 5 of the constrained minimization algorithm in [Surles 1992] for general skeletal figures.

### 5.2. Architectural layout

Simple changes in a modeling system for architectural models (e.g. blueprints) often require numerous operations. For example, narrowing a corridor requires moving the corridor walls *and* lengthening the walls that connect to it. A large portion of the effort in the Building Walkthrough project [Airey 1990] at the University of North Carolina at

Chapel Hill is spent fixing and maintaining databases of models (these databases contain approximately 4,000 to 30,000 polygons). An automated radiosity calculation followed by viewing uncovers modeling errors, including walls not connected to ceilings and doors outside the plane of their walls. Most of the errors arise from previous database edits that left parts of the model inconsistent.

Applying constrained minimization to this application could reduce these burdens. In the corridor example, constraints can require that moving the corridor wall also moves the connecting walls. Additional cost functions can increase as certain goals are not met such as rooms containing a certain area or being a given distance from an exit.

# Bibliography

Airey, J. M., Rohlf, J. H. and Brooks, F. P., Jr. (1990). "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments," *Symposium on Interactive 3D Graphics* Vol. 24, No. 2, 41-50.

Akeley, K. and Jermoluk, T. (1988). "High-Performance Polygon Rendering," *Computer Graphics* Vol. 22, No. 4, 239-246.

Barr, A. (1984). "Global and Local Deformations of Solid Primitives," *Computer Graphics* Vol. 18, No. 3, 21-30.

Barzel, R. and Barr, A. (1988). "A Modeling System Based On Dynamic Constraints," *Computer Graphics* Vol. 22, No. 4, 179-188.

Bentley, J. L. and Friedman, J. H. (1979). "Data Structures for Range Searching," *Computing Surveys* Vol. 11, No. 4, 397-409.

Biosym (1991). *Insight*. San Diego, CA, Biosym Technologies Inc.

Borning, A. (1979). *THINGLAB--A Constraint Oriented Simulation Laboratory*. Ph.D. Dissertation, Stanford University.

Britton, E. G. (1977). *A Methodology for the Ergonomic Design of Interactive Computer Graphic Systems, and its Application to Crystallography*. Ph. D., University of North Carolina at Chapel Hill.

Brooks, B. R., et al. (1983). "CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations," *Journal of Computational Chemistry* Vol. 4, No. 2, 187-217.

Brooks III, C. L., Karplus, M. and Pettitt, B. M. (1988). *Proteins: A Theoretical Perspective of Dynamics, Structure, and Thermodynamics*. New York, John Wiley & Sons.

Claris (1988). *MacDraw II*. Mountain View, Claris Corporation.

Dickerson, R. E. and Geis, I. (1969). *The Structure and Action of Proteins*. New York, Harper and Row.

Duff, I. S. and Reid, J. K. (1983). "The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations," *ACM Transactions on Mathematical Software* Vol. 9, No. 3, 302-325.

Fasman, G. D., Ed. (1989). *Prediction of Protein Structure and the Principles of Protein Conformation*. New York, Plenum Press.

Fletcher, R. (1987). *Practical Methods of Optimization*. John Wiley and Sons.

Gill, P., Murray, W. and Wright, M. (1981). *Practical Optimization*. Academic Press.

Grant, E. (1991). *Constraint-based Design by Cost Function Optimization*. Ph.D. Dissertation, University of North Carolina at Chapel Hill.

Harwell (1988). *Harwell Sparse Matrix Library*. Harwell Laboratory of the UKAEA, Numerical Analysis Group.

Hecht, M. H., et al. (1990). "De Novo Design, Expression, and Characterization of Felix: A Four-Helix Bundle Protein of Native-Like Sequence," *Science* Vol. **249**, No. 4964, 884-891.

Hendrickson, W. A. and Konnert, J. H. (1980). "Incorporation Of Stereochemical Information Into Crystallographic Refinement," *Computing in Crystallography,* Bangalore, India, The Indian Academy of Sciences, 13.01-13.25.

Hermans, J. (April 20, 1992). Personal communication at North Carolina Supercomputer Center.

Hermans, J. and Carson, M. (1989). *CEDAR--Crystal Energy Dynamics and Refinement*. University of North Carolina at Chapel Hill.

Hestenes, M. R. (1975). *Optimization Theory, The Finite Dimensional Case*. New York, John Wiley and Sons.

Kirkpatrick, S., Gelatt, C. D., Jr. and Vecchi, M. P. (1983). "Optimization by Simulated Annealing," *Science* Vol. **220**, No. 4598, 671-679.

Knuth, D. E. (1973). *Fundamental Algorithms, The Art of Computer Programming*. Addison-Wesley.

Leler, W. (1987). *Specification and Generation of Constraint Satisfaction Systems*. Ph.D. Dissertation, University of North Carolina at Chapel Hill.

Levinthal, C. (1966). "Molecular Model-building by Computer," *Scientific American* Vol. **214**, No. 6,

Luenberger, D. G. (1973). *Introduction to Linear and Nonlinear Programming*. Addison-Wesley.

Mc Cammon, J. A., Gelin, B. R. and Karplus, M. (1977). "Dynamics of folded proteins," *Nature* Vol. **267**, No. 16, 585-590.

Miller, G. (1988). "The Motion Dynamics of Snakes and Worms," *Computer Graphics* Vol. **22**, No. 4, 169-178.

Pentland, A. and Williams, J. (1989). "Good Vibrations: Modal Dynamics for Graphics and Animation," *Computer Graphics* Vol. **23**, No. 3, 215-222.

Platt, J. (1989). *Constraint Methods for Neural Networks and Computer Graphics*. Ph.D. Dissertation, California Institute of Technology.

Platt, J. and Barr, A. (1988). "Constraint Methods for Flexible Models," *Computer Graphics* Vol. **22**, No. 4, 279-288.

Polygen (1991). *Quanta*. Waltham, MA, Polygen Corporation.

Porter, T. and Duff, T. (1984). "Compositing Digital Images," *Computer Graphics* Vol. **18**, No. 3, 253-259.

Richardson, J. S. (1992). *Felix,* Department of Biochemistry at Duke University.

Richardson, J. S. (1992). *Flavodoxin,* Department of Biochemistry at Duke University.

Schulz, G. E. and Schirmer, R. H. (1979). *Principles of Protein Structure*. New York, Springer-Verlag.

Sederberg, T. and Parry, S. (1986). "Free-Form Deformation of Solid Geometric Models," *Computer Graphics* Vol. **20**, No. 4, 151-160.

Stroustrup, B. (1986). *The C++ Programming Language*. Addison-Wesley.

Surles, M. (1992). "An Algorithm With Linear Complexity For Interactive, Physically-based Modeling of Large Proteins," *Computer Graphics* Vol. **26**, No. 4, 8.

Surles, M. (1992). "Interactive Modeling Enhanced with Constraints and Physics—With Applications in Molecular Modeling," *Symposium on Interactive 3D Graphics* Vol. **26**, No. 2, 8.

Sutherland, I. E. (1963). *SKETCHPAD: A Man-Machine Graphical communication system*. Ph.D. Dissertation, MIT.

Terzopoulos, D., et al. (1987). "Elastically Deformable Models," *Computer Graphics* Vol. **21**, No. 4, 205-214.

Tripos (1988). *Sybyl.* St. Louis, MO, Tripos Associates.

Tuffery, P., et al. (1991). "FORME: An interactive package for protein backbone deformation," *Journal of Molecular Graphics* Vol. **9**, No. 3, 175-181.

van Gunsteren, W. F. and Berendsen, H. J. C. (1977). "Algorithms for macromolecular dynamics and constraint dynamics," *Molecular Physics* Vol. **34**, No. 5, 1311-1327.

Weiner, S. J., et al. (1984). "A New Force Field for Molecular Mechanical Simulation of Nucleic Acids and Proteins," *Journal of the American Chemical Society* Vol. 106, No. 765-784.

Weiner, S. J., et al. (1986). "An All Atom Force Field for Simulations of Proteins and Nucleic Acids," *Journal of Computational Chemistry* Vol. 7, No. 2, 230-252.

Witkin, A., Fleischer, K. and Barr, A. (1987). "Energy constraints on parameterized models," *Computer Graphics* Vol. 21, No. 4, 225-232.

Witkin, A., Gleicher, M. and Welch, W. (1990). "Interactive Dynamics," *Symposium on Interactive 3D Graphics* Vol. 24, No. 2, 11-21.

Witkin, A. and Kass, M. (1988). "Spacetime Constraints," *Computer Graphics* Vol. 22, No. 4, 159-168.

# Appendix

The following paper discusses two modeling problems solved using *Sculpt*. The paper is reprinted with permission of the Association of Computing Machinery, Inc. A full citation of the paper follows.

Surles, M. (1992). "Interactive Modeling Enhanced with Constraints and Physics—With Applications in Molecular Modeling," *Symposium on Interactive 3D Graphics* Vol. **26**, No. 2, pp. 175-182.

# Interactive Modeling Enhanced with Constraints and Physics— With Applications in Molecular Modeling

**Mark C. Surles**
**Department of Computer Science**
**University of North Carolina at Chapel Hill**
**Chapel Hill, NC 27599-3175**

Appears in the 1992 Symposium on Interactive 3D Graphics

## Abstract

Interactive modeling systems that continually maintain a physically-realistic representation of an object combine advantages of interactive graphics and batch simulations. In this paper I address two advantages of incorporating physics into *Sculpt*, an interactive protein modeling system. First, time-consuming model correction is avoided by maintaining a physically-valid model throughout a modeling session. Second, additional cues about model properties can arise when a chemist interactively guides a simulation rather than views a cine loop from a pre-computed simulation. I argue these benefits with examples from sessions with *Sculpt*. A chemist can interactively move atoms while *Sculpt* automatically maintains proper bond topology and atom separations. *Sculpt* models bonded and non-bonded atom interactions for medium-size proteins (800 atoms) at 0.6 updates per second on a Silicon Graphics 240 using a constrained energy minimization method.

**CR Categories and Subject Descriptors:** I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling; I.3.6 [**Computer Graphics**]: Methodology and Techniques; I.J.2 [**Computer Applications**]: Physical Sciences.

**Additional Keywords and Phrases:** Physically-based modeling, interactive modeling, constraint systems, scientific visualization.

## 1. Introduction

Within the last ten years a trend in computer graphics has been to increase scene realism by using physically-based models. Animators use physically-based modeling to create realistic detailed behavior. Most animations generated with physically-based modeling, to date, required minutes to hours of computation for each frame. This large computation time has kept physically-based modeling out of interactive graphics systems except with small, simple models. However, increased computer speeds now permit adding physically-based models to interactive systems. I have picked a large modeling problem with simple properties to study issues that arise in modeling physical properties in an interactive graphics system.

Protein modeling systems represent molecules containing one hundred to several thousand atoms. The systems can be classified as interactive or batch (though some interactive systems have batch processing). Most interactive systems maintain bonded properties such as fixed bond lengths and angles by restricting operations to rotation of segments about particular bonds. The performance of interactive systems is only limited by the display capability of the graphics system since the modeling operations are only rotations. Batch simulations model variance in bond lengths and angles *and* interactions among non-bonded atoms over relatively near and far distances. Accurately modeling all these properties requires batch computation, even for small proteins.

Today an interactive, physically-based modeling system, called *Sculpt*, models non-bonded atom interactions for medium-size proteins (800 atoms) on a Silicon Graphics 240 at 0.6 updates per second. *Sculpt* lets a chemist interactively move atoms while automatically keeping correct bonded properties and non-bonded atom separations using a constrained energy minimizer. Compared to many other physically-based modeling systems in computer graphics, *Sculpt* models simpler properties (e.g. angles versus volumes) and minimizes static strain energies rather than functions of object dynamics. However, system performance now allows investigation into issues that arise when physically-based modeling is applied to complex real applications.

Chemists that collaborate on the project believe interactive, physically-based modeling will relieve many manual modeling tasks, allowing more work in less time, and provide additional cues about protein behavior. In this paper I present two improvements the system provides that result from modeling physical properties interactively. First, the system removes the often laborious task of fixing a physically-invalid model after a modeling session. Though interactive systems such as Sybyl [15] maintain fixed bond lengths and angles, they make the chemist keep non-bonded atoms at appropriate separations. Second, the system provides a new medium for exploring protein properties by allowing interactive, guided simulation. This should combine benefits of interactive graphics and batch simulations.

## 2. Related work

Physically-based modeling frequently aids computer animations by automating detailed motion planning and complex object interactions. Miller generates realistic snake motions by modeling muscle contractions with springs and friction against surfaces [9]. Witkin models the energy and momentum of a Luxo lamp jumping hurdles and ski jumps [18]. Terzopoulos models energy in elastically deformable objects such as cloth to create animations of flags [14]. These examples simulate the motion of objects by first stating application-specific conditions about the objects and scene and then solving Newton's equations of motion.

Similar applications use *constraints* to restrict the allowable states of objects and express dependencies among objects. Barzel uses constraints in animation to specify paths for objects [4]. Witkin uses geometric constraints to assemble models [16], and he describes a system that lets a user interactively connect and manipulate objects such as a mechanical assembly or tinker-toy [17]. Constraints maintain constant volume in incompressible solids [12] and restrict penetration when a ball strikes a trampoline [11].

## 3. Driving problem – protein modeling

A protein, to a first approximation, contains fixed bond lengths, fixed bond angles, and some planar segments. Figure 1-A shows three sequential segments in a protein with vectors representing bonds between atoms and gray areas denoting planar regions. The only degrees of freedom in the figure are rotations about the N-C and C-C bonds that enter and leave each planar segment. A linear sequence of the segments comprise the protein *backbone*. Attached to the atom between each segment (C) are *sidechains* (not shown) with additional fixed length and angle properties. Superimposed onto this geometric model are non-bonded attractions and repulsions. Attractions hold nearby atoms together, while repulsions maintain a minimal separation between all atom pairs.

Chemists often use brass models (Kendrew models) to study geometric properties and relationships in a protein. Brass models contain segments shown in Figure 1-A connected with rotational joints about the N-C and C-C bonds. Manipulating such a model with one's hands aids understanding of relationships. However, the models have two drawbacks. First, the model's size becomes difficult to hold and manipulate when dealing with large molecules (e.g. an 800-atom brass model of the protein in Color Plate 1 is 80 centimeters wide when 2 cm of brass represents 1 Ångstrom, a typical bond). Second, brass models do not represent attractive and repulsive interactions among non-bonded atoms.

Chemists use computers to model large proteins and non-bonded atom interactions. Interactive modeling systems resemble brass models by allowing only rotations about particular bonds. The limiting factor in interactive systems is display rate of the graphics machine. Batch simulations model non-bonded atom interactions and more accurately model bond lengths and angles (these do vary, though by only a few percent).

Protein modeling provides a good driving problem for research in interactive physically-based modeling. First, the benefits of interactive graphics and batch simulations are each well established. Second, real users want such a system and will provide valuable assistance in its development. Third, the size of useful models requires improved algorithms for interactive modeling on current machines. Fourth, many aspects of protein modeling are similar to other problems. For example, the inherent three-dimensional structure requires addressing mechanical modeling issues similar to those encountered in articulated-figure motion and computer-aided design. Fifth, understanding the interplay of properties in proteins during the modeling requires good visualization paradigms.

## 4. *Sculpt's* interface and performance

*Sculpt* continually maintains realistic protein properties as a chemist moves an atom. *Sculpt* lets a chemist move an atom by first attaching a spring between the atom and the cursor and then dragging the cursor in a desired direction. Throughout the dragging process, *Sculpt* polls the cursor position and adds the strain energy of that spring to the energy in the protein. *Sculpt* then finds a local minimum of the total energy that also maintains rigid bond lengths, angles, and planar segments. *Sculpt* also lets a chemist insert a spring that continually pulls an atom towaros a given three-dimensional position.

The color plates show photographs of *Sculpt* sessions. Depth-cued vectors represent bonds between atoms; cyan denotes the central backbone, and tan denotes sidechains connected to the backbone. Gold coils show springs attached by a chemist to pull atoms toward positions denoted by the gold thumbtacks. Color Plate 1 shows a model containing 760 atoms of a medium-sized protein called Felix [8]. The model contains 2205 constraints (bond length, angle, and others) and approximately 8005 energy functions (attraction, repulsion, and others). The backbone in Color Plate 1 winds through four helices (purple cylinders highlight the two on the left). Color Plate 2 shows a model composed of the two helices highlighted in Color Plate 1. The model contains 355 atoms, 1027 constraints, and approximately 3450 energy functions. The text in Color Plate 2 names several of the sidechains.

*Sculpt* maintains approximately 0.7 updates per second with the model in Color Plate 1 and 1.5 updates per second with the model in Color Plate 2, on a Silicon Graphics 240-GTX [2]. An update includes the following steps: evaluate protein properties (bond lengths, angles, attractions, and repulsions) and their derivatives, minimize the energy and satisfy the constraints, update atom positions, and display the results. Though this performance is twenty times too slow for smooth interaction, our chemist collaborators believe the performance already provides enough interactivity on medium-size proteins that new, useful research can be accomplished that could not previously be undertaken. The system is described in greater detail in [13].

## 5. Maintaining a consistent model

Users of interactive modeling systems (not only molecular) often unintentionally move objects into a configuration that violates required properties of the application—producing an invalid model database. For example: moving an endpoint so that an originally constrained line is no longer horizontal; moving a wall without adjusting those adjoining it; leaving cables dangling in a car engine after moving the alternator; moving atoms closer than electron shells allow.

Changing a computer object so that it mimics the properties of its physical counterpart can be arbitrarily complex. Most modeling applications leave this task to the user. For example, moving a wall in an architectural model requires that a user rejoin all the adjacent walls and then ensure those changes did not invalidate the model. Some molecular modeling systems let a user invoke a batch energy minimizer to move atoms into a valid arrangement. However, such automated post-processing methods can change the model differently than the user intends.

An interactive modeling system that maintains a physically-valid model throughout user modifications eliminates the model re-idealization task. This section presents two protein-modeling examples to illustrate complexities that can arise in manual and automated methods for repairing the invalid models.

## 5.1. A simple edit requiring complex repairs

A common operation in molecular modeling requires flipping a planar segment (peptide) in the backbone, surrounded closely by neighboring atoms, by 180 degrees. Figure 1 shows two stages of the flip operation. Figure 1-A shows the center segment and its neighbors before a flip. Lines represent bonds between atoms and hashed areas represent rigid planar segments. Each atom contains an electron shell that (to a first approximation) cannot intersect other electron shells. Figure 1 represents the shells with circles (notice the circles do not intersect in Figure 1-A). Most systems only allow rotations about the C–C and C–N bonds so that bond lengths, angles and planar groups do not change. This makes the flip difficult by itself since one rotation affects all the atoms further along the chain. Figure 1-B shows the center segment flipped 180 degrees after an appropriate sequence of rotations. The model now requires repairs because the circles overlap.

**Manual correction.** A chemist can manually adjust the atom positions to remove the intersections in Figure 1-B. Moving an atom requires that a chemist choose appropriate combinations of rotations so that other segments do not move. Moving one atom usually causes interference with another, which then requires additional repairs. Correctly fitting the flipped segment often causes small changes that propagate through the entire protein. In practice this problem is much harder because a chemist fits spheres rather than circles and approximates non-bonded atom interactions by getting the spheres to touch. Professor Jane Richardson, a collaborator from Duke University's Biochemistry Department, usually adjusts models manually after operations such as this flip. This example takes on the order of fifteen minutes.

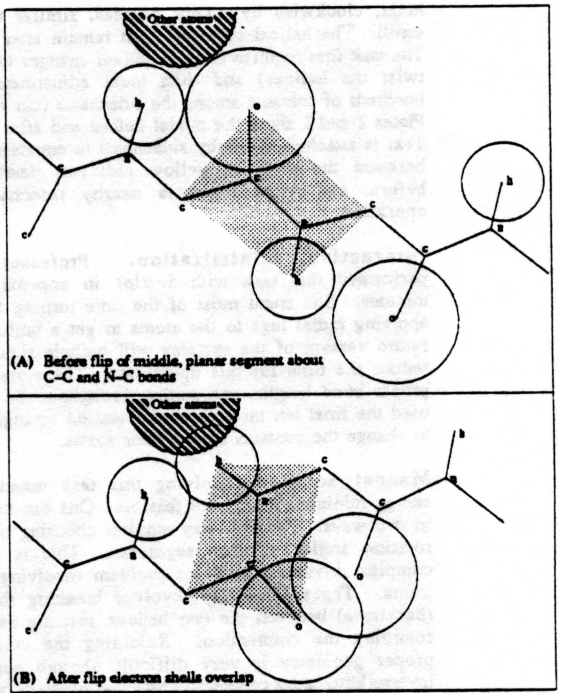

Other atoms
c
c
c
c
c
h
(A) Before flip of middle, planar segment about C–C and N–C bonds
Other atoms
h
c
c
c
c
c
(B) After flip electron shells overlap

**Figure 1: Modeling errors introduced by flipping a rigid planar segment.**

**Batch minimization.** A chemist can also use a batch minimization package to remove the intersections. Such packages find a local minimum of the ensemble energy associated with the overlapping shells. These work well if the atom shells only slightly overlap. Overlaps greater than, say, twenty percent contain very large strain energy that cause minimization packages to make large changes to the model. Batch routines often resolve such interactions by moving atoms the chemist did not intend to change. Professor Richardson interleaves some manual intervention with energy minimization to avoid these undesirable changes.

**Interactive minimization.** Performing this operation in *Sculpt* requires approximately thirty seconds (depending on the size of the protein). A chemist tugs the atoms from one orientation to another while *Sculpt* continuously adjusts segments along the chain to accommodate the change. Throughout the operation, *Sculpt* maintains a valid protein model. *Sculpt* does nothing here that batch minimization systems cannot perform. The difference is the small minimization time in *Sculpt* allows the system to continuously minimize the energy rather than do it once after the user interaction.

## 5.2. A complex task requiring exorbitant re-idealization

This example requires changing the orientation of two helices between Color Plate 2 and 3 by unwinding the lower helix, counter-clockwise by ninety degrees, and winding the upper helix, clockwise by ninety degrees, similar to unrolling a scroll. The helical structure must remain after the operation. The task first requires large structural changes to the model (to twist the helices) and then local adjustments to remove hundreds of contacts among the sidechains (tan vectors). Color Plates 2 and 3 show the model before and after the operation. Text is attached to nearby sidechains to emphasize the change between the pictures; yellow indicates nearby sidechains before, and white indicates nearby sidechains after the operation.

**Interactive minimization.** Professor Richardson performed this task with *Sculpt* in approximately thirty minutes. She spent most of the time turning the helices by applying radial tugs to the atoms to get a uniform twist. (A future version of the systems will include rigid segments to reduce the time for this operation.) The system maintained proper bond lengths and angles throughout the session. She used the final ten minutes of the session arranging sidechains to change the contacts among their atoms.

**Manual solution.** Solving this task manually, without energy minimization, is not feasible. One can turn the helices in two ways. The first way requires choosing the appropriate rotation angles between segments. This is an extremely complex, inverse-kinematics problem involving hundreds of joints. The second way involves breaking the connection (backbone) between the two helices, rotating each helix, and rejoining the connection. Rejoining the connection with proper geometry is very difficult, though easier than the inverse-kinematics problem. Once the two helices are turned, a chemist must resolve hundreds of contacts between sidechain atoms. Professor Richardson attempted to solve this task manually but quit after several frustrating days and was never fully satisfied with the results.

**Batch minimization.** A chemist could specify target positions for some atoms (if such end positions are known) and invoke an energy minimization package. The minimizer chooses a path to move the atoms along towards their targets. Certain paths can tear the model apart in order to reach the target (e.g. through the middle of a structure). Instead of solving the problem with one minimization, a chemist may choose subgoals along a path to the target and run minimizations for each subgoal. This approach works better than the manual solution, but the turnaround time between subgoal minimization limits the number of steps picked along the path. Continuously running a minimization as a chemist moves atoms to targets is the same as choosing an infinite sequence of subgoals and running batch minimizations on each.

## 6. Interactive, guided simulation

Interactive modeling of physical properties is essentially a form of interactive, guided simulation. Placing a user in the computation-loop of a simulation that once required hours or days we hope will provide greater insights to properties and relationships in a model. This section discusses benefits of interactive simulations compared to batch simulation and interactive graphics without simulations and discusses complications of scientific visualization in interactive simulations.

### 6.1. Simulation

Simulations can illustrate molecular properties not easily incorporated into brass models such as attractions and repulsions between non-bonded atoms. Though a chemist understands individual attractions and repulsions between two atoms, comprehension of hundreds of simultaneous interactions becomes very difficult. Simulations are typically used to examine specific atom interactions in a molecule. A simulation requires that a chemist choose model parameters, run the simulation, and view the results in a cine loop. If the results do not show the specific interaction, the steps are repeated with new parameters. Simulations have uncovered important molecular properties, but long turnaround times have kept this from being a common exploration tool for most researchers.

*Sculpt* lets a chemist explore non-bonded interaction while interactively moving atoms. Professor Richardson believes interactively exploring protein models with non-bonded interactions will improve perception of subtle relationships within proteins. In several sessions Professor Richardson has seen unexpected reactions that, upon closer examination, resulted from non-bonded interactions competing against other properties such as bond rotations.

Interactive modeling of physical properties augments benefits from batch simulations with features from interactive graphics. Today chemists use interactive graphics to study a static structure or series of structures from pre-computed simulations. Interactively controlling the view and display parameters provides more cues about a molecule's structure and nature than does viewing multiple, static images. Guiding an interactive simulation while immediately viewing the results lets the user remain continually engaged in the modeling process. I believe this provides greater situational awareness of complex relationships within a model than viewing cine loops of simulations Guiding an interactive simulation lets a user stumble upon unexpected reactions in the model that may go unnoticed in batch simulations (the Ahah! phenomenon). Also more users will experiment with the models as turnaround time is shortened.

One advantage batch simulations, viewed with cine loops, have over interactive simulation is the ability to replay the simulation. Since a cine loop is a sequence of frames, a user can easily move backwards in the sequence to study a particular property. Unless a system saves all user actions during an interactive simulation, a user cannot readily return to a previous state. Like an on-going laboratory experiment, an event cannot be repeated without re-running the experiment from the beginning with the same steps.

### 6.2. Visualization of non-bonded forces

Near-neighbor interactions among non-bonded atoms play an important role in protein conformations by holding those atoms together at fixed distances. A protein modeling system should convey these interactions to help a chemist tightly pack the protein's interior. These interactions, unfortunately, are not as simple to display as a bond (vector connecting atoms). Figure 2 plots the potential energy of the van der Waal interaction between two atoms as a function of their separation (1 Ångstrom = $10^{-10}$ meters). The plot shows a maximum attractive (negative) energy, $E_m$, at a separation of $R_m$. The energy decreases nonlinearly as the separation increases from $E_m$. The energy becomes repulsive, increasing at a different nonlinear rate, as the separation decreases from $E_m$. Each atom in a protein, on average, interacts with ten atoms within a six-Ångstrom radius (the model in Color Plate 1 contains 7,577 van der Waal interactions). A useful visualization of a non-bonded interaction should convey the type (attractive or repulsive), magnitude, and ideal separation, $R_m$.

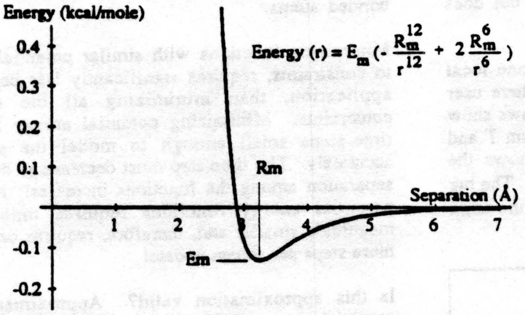


**Figure 2: Van der Waal potential energy between two atoms.**

*Sculpt* displays van der Waal interactions that have an energy magnitude greater than a user-defined threshold. A partial spherical shell is placed around both of the interacting atoms and aligned along a vector between them (see Color Plate 4). Currently a shell with a solid angle of $0.4\pi$ steradians (ten percent coverage) represents the weakest interaction. Solid angle increases with the magnitude of the interaction. Weak interactions are represented by dot spheres, and strong interactions are represented by wireframe spheres. A dot-sphere indicates that an interaction exists without distracting the user and consuming as much screen space as the wireframe sphere. Blue denotes attraction, and red denotes repulsion.

Color Plate 4 illustrates this visualization on a small model. The photograph shows a spring attached to a planar ring (highlighted with a purple tube) that pulls one atom into another. Notice the wireframe shells around the two atoms labeled with text. The shells bend rather than intersect so that the vector in the two shells do not interfere visually. Intersecting wireframe shells are difficult to associate with their respective atoms.

## 7. Adding physical modeling to interactive graphics systems

The physically-based modeling module in *Sculpt* is inserted into the control flow of an interactive graphics systems with minor modifications. The white boxes in Figure 3 list the sequence of actions in the interactive graphics system: the system receives a user action (e.g. mouse movement), interprets it (move an atom by one Ångstrom in a given direction), applies the change to the model database (change the coordinates of atom), and displays the next frame. The shaded box shows the additional step that modifies the user action according to properties of the application (e.g. also adjust distances to neighboring atoms).

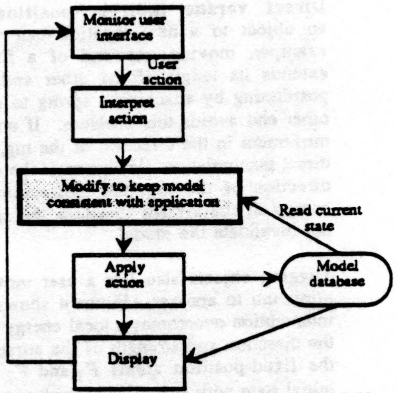


**Figure 3: Steps in an interactive modeling system for processing a user action.**

The control-flow presented in the white boxes is similar to the event loop of many graphics systems [7]. The remainder of this section discusses some implementation issues addressed in *Sculpt* that may be useful to others wishing to incorporate physically-based modeling into interactive graphics systems.

### 7.1. Constrained minimization

*Sculpt* implements the shaded box in Figure 3 with a constrained minimizer in the following manner. *Sculpt* converts a user action into a potential energy function (e.g. a spring to pull atoms). *Sculpt* then finds a local minimum of the total system energy (from protein and user) that also satisfies the set of bond length and angle constraints. Mathematically, the minimizer solves the following problem:

**Given:**

| | |
|---|---|
| **x** | **model state (e.g. vector of atom positions)** |
| **Energy(x)** | **sum of potential energies in model** |
| **Constraint(x)** | **vector of constraint functions** |

**Solve:**

| | |
|---|---|
| **minimize** | **Energy(x)** |
| **such that** | **Constraint(x) = 0.** |

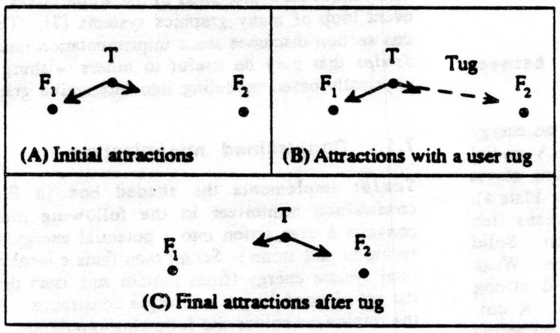
The minimizer finds the solution using a method of Lagrange multipliers as discussed in [6], [17] and [13]. The minimizer determines changes in atom positions. The changes are sent to the next module in Figure 3 (Apply action) which then updates the model database.

Other constrained-minimization approaches fit within the framework of Figure 3. Witkin minimizes a potential energy function [16] associated with the physical state of elastic models. Amburn minimizes costs associated with design goals [3]. Phillips uses kinematic constraints to reduce allowable joint movements in an articulated figure while minimizing costs associated the positioning goals [10].

## 7.2. Positioning

**Direct versus indirect positioning.** Directly moving an object to a new location can violate constraints. For example, moving one end of a fixed-length line segment extends its length if the other end cannot move. Indirect positioning by attaching a spring to an object and tugging the other end avoids this problem. If no opposing force prevents movement in the direction of the tug, the result is the same as direct manipulation. However, if the object cannot move in the direction of the tug, the indirection increases the potential energy in the system (because the spring stretches) but does not invalidate the model.

Tugging objects also lets a user move atoms from one local minimum to another. Figure 4 shows an example where user intervention overcomes a local energy minimum. Arrows show the direction and strength of the attractions among atom *T* and the fixed-position atoms $F_1$ and $F_2$. Figure 4-A shows the initial state with atom *T* attracked more by $F_1$ than $F_2$. The tug in Figure 4-B (indicated with the dashed arrow) pulls the atom towards $F_2$. Figure 4-C shows the final result.



(A) Initial attractions

(B) Attractions with a user tug

(C) Final attractions after tug

**Figure 4: A user spring pulls atom *T* between energy minima.**

**Which physics? Dynamics or statics.** Physically-based modeling, as it has most often been used in computer graphics, aims to determine physically-realistic motions and trajectories of objects with specific physical properties (e.g. blowing flags [14] and jumping Luxo [18]). The approach solves Newton's second law, **F=ma**, which gives the ***acceleration*** of objects, and combines this with an initial position and velocity to determine motions.

In this work, stable conformations, not the trajectories of reaching them, are the concern. *Sculpt* achieves this by modeling potential energy rather than forces in a model. This gives ***strains*** between objects that the system minimizes. The objects never contain velocity information. Minimizing the potential energy (strain) moves the objects but does not induce momentum. This technique provides greater control over object positions and takes less computation.

## 7.3. Approximating stiff model components with constraints

*Sculpt* makes an approximation that dramatically improves performance without appreciably decreasing accuracy. Properties whose deformation requires very large strain energies relative to others in a model are replaced by rigid constraints. For example, a bond length is constrained to its ideal value since the potential energy increase for extending a bond is ***five*** orders-of-magnitude larger than that associated with a comparable increase in distance between two non-bonded atoms.

Minimizing functions with similar potential energies, subject to constraints, requires significantly less computation, in this application, than minimizing all the energies without constraints. Minimizing potential energy functions requires time-steps small enough to model the stiffest properties accurately. The time-step must decrease as the potential energy separation among the functions increases. Minimizing all the potential energy functions requires time-steps orders-of-magnitude smaller and, therefore, requires orders-of-magnitude more steps per screen update!

Is this approximation valid? Approximating bond-length, potential energy functions with rigid constraints reduces the accuracy of the physical model. However, the large potential energy signifies that bond-length variability is orders-of-magnitude smaller than the variability of other properties. Since the bond lengths hardly change, constraining them for increased performance is justified. *Sculpt* lets a chemist trade performance for accuracy when desired, by modeling lengths with potential energy functions.

An important principle influences this approximation—only compute what is significant. *Sculpt* follows this by only accurately modeling properties that can vary significantly and constraining the others. This approach can prove useful in other applications with wide variability in energy magnitudes.

## 8. Other applications

Removing model re-idealization and enhancing understanding of model properties will most likely arise in other interactive applications that incorporate physically-based modeling. The particular benefits and implementations are specific to the applications. However, similarity between the control-flow in *Sculpt* and other applications suggests that a generic, physically-based modeling module may eventually be developed. For now, the system development effort may be overkill for simple modeling applications and only justified for complex modeling applications. I conclude with two example applications that can benefit from adding physically-based modeling.

### 8.1. Architectural layout

Simple changes in a modeling system for architectural models (e.g. blueprints) often require numerous operations. For example, narrowing a corridor requires moving the corridor walls *and* lengthening the walls that connect to it. A large portion of the effort in the Building Walkthrough project [1] at the University of North Carolina at Chapel Hill is spent fixing and maintaining databases of models (these databases contain approximately 4,000 to 30,000 polygons). An automated radiosity calculation followed by viewing uncovers modeling errors, including walls not connected to ceilings and doors outside the plane of their walls. Most of the errors arise from previous database edits that left parts of the model inconsistent.

Applying constrained minimization to this application reduces these burdens. In the corridor example, constraints can require that moving the corridor wall also moves the connecting walls. Additional cost functions can increase as certain goals are not met such as rooms containing a certain area or being a given distance from an exit.

### 8.2. Drafting

Most interactive drafting and drawing systems ignore application-specific properties to reduce computation and broaden product applicability. They base operations (e.g. move, stretch) on individual, geometric primitives (polygons, lines, control points, etc.). Information regarding an object's construction is usually discarded. For example, MacDraw II lets a user construct a line constrained to the horizontal, but discards the horizontal requirement after construction [5]. The package does not restrict the line to the horizontal if a user later moves one of its endpoints. Keeping information about an object's structure and properties allows a system to maintain a consistent model throughout model editing.

## 9. Future work

The immediate goal for future work is to place *Sculpt* in a chemistry lab and gather results about its usefulness for solving daily protein-modeling problems. This should offer direction for future enhancements to the modeling and visualization components of the system.

The visualization issues offer large scope for future research. The near-neighbor visualization discussed in this paper is adequate, though not great. I will continue to examine the near-neighbor interactions. A much harder property to visualize is long-distance, electrostatic interaction. These interactions can extend between atoms on opposite sides of a molecule. Visualizing these interactions will be hard.

Finally, I plan to apply the techniques described in this paper to other applications. The input to the *Sculpt* system is a list of points with a set of length and angle functions defined on the points. Under one thousand lines of modeling code (out of ten thousand) is specific to molecules. With this framework I hope to examine interactive manipulation of skeletal figures without significant system development.

## Acknowledgements

I thank Professors David and Jane Richardson of Duke University for continued guidance and encouragement. Professor Frederick Brooks provided many useful critiques about the research and the manuscript. Jane Richardson conceived and Jim Begley implemented the shells used in the visualization of non-bonded interactions. This work is supported by NIH National Center for Research Resources grant RR-02170.

## References

1. Airey, J. M., Rohlf, J. H. and Brooks, F. P., Jr. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. *Symposium on Interactive 3D Graphics. 24*, 2 (1990), 41-50.

2. Akeley, K. and Jermoluk, T. High-Performance Polygon Rendering. *Computer Graphics. 22*, 4 (1988), 239-246.

3. Amburn, P., Grant, E. and Whitted, T. Managing Geometric Complexity with Enhanced Procedural Models. *Computer Graphics. 20*, 4 (1986), 189-195.

4. Barzel, R. and Barr, A. A Modeling System Based On Dynamic Constraints. *Computer Graphics. 22*, 4 (1988), 179-188.

5. *MacDraw II*. Claris Corporation, 1988.

6. Fletcher, R. *Practical Methods of Optimization*. John Wiley and Sons, 1987.

7. Foley, J. D., van Dam, A., Feiner, S. K. and Hughes, J. F. *Computer Graphics—Principles and Practice*. Addison-Wesley, New York, 1990.

8. Hecht, M. H., Richardson, J. S., Richardson, D. C. and Ogden, R. C. De Novo Design, Expression, and Characterization of Felix: A Four-Helix Bundle Protein of Native-Like Sequence. *Science. 249*, 4964 (1990), 884-891.

9. Miller, G. The Motion Dynamics of Snakes and Worms. *Computer Graphics. 22*, 4 (1988), 169-178.

10. Phillips, C., Zhao, J. and Badler, N. Interactive Real-time Articulated Figure Manipulation Using Multiple Kinematic Constraints. *Symposium on Interactive 3D Graphics. 24*, 2 (1990), 245-250.

11. Platt, J. *Constraint Methods for Neural Networks and Computer Graphics*. Ph.D. Dissertation, California Institute of Technology, 1989.

12. Platt, J. and Barr, A. Constraint Methods for Flexible Models. *Computer Graphics. 22*, 4 (1988), 279-288.

13. Surles, M. *Techniques For Interactive Protein Manipulation*. Ph. D. Dissertation Manuscript, University of North Carolina at Chapel Hill, 1992.

14. Terzopoulos, D., Platt, J., Barr, A. and Fleisher, K. Elastically Deformable Models. *Computer Graphics. 21*, 4 (1987), 205-214.

15. *Sybyl*. Tripos Associates, 1988.

16. Witkin, A., Fleischer, K. and Barr, A. Energy constraints on parameterized models. *Computer Graphics. 21*, 4 (1987), 225-232.

17. Witkin, A., Gleicher, M. and Welch, W. Interactive Dynamics. *Symposium on Interactive 3D Graphics.* *24*, 2 (1990), 11-21.

18. Witkin, A. and Kass, M. Spacetime Constraints. *Computer Graphics.* *22*, 4 (1988), 159-168.

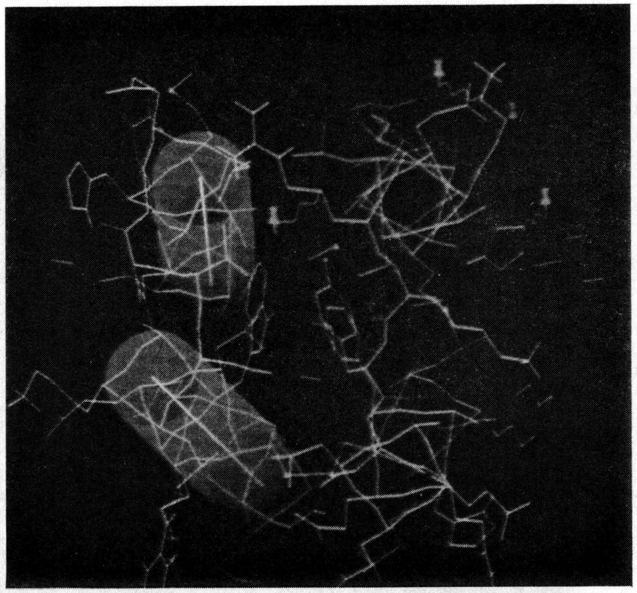


Plate 1: Protein with 760 atoms.

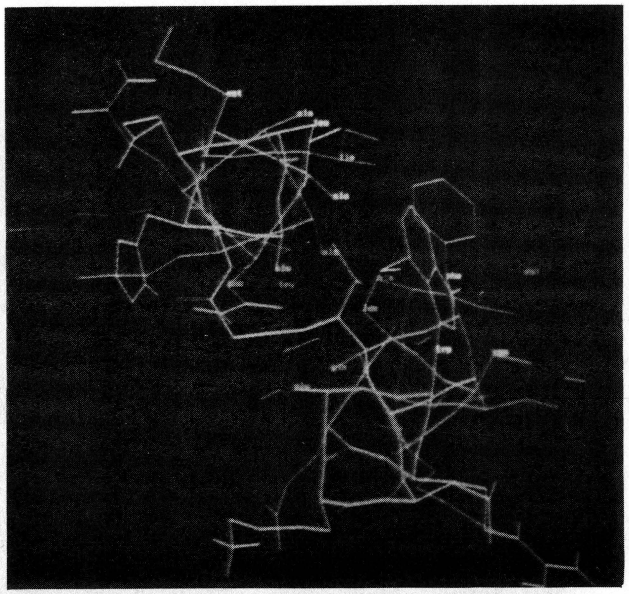


Plate 2: Two helices before rotations.

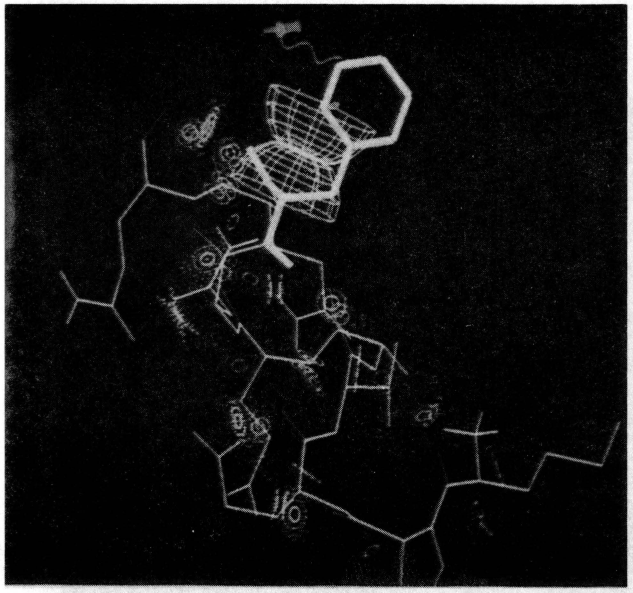


Plate 4: Non-bonded interactions represented with partial wireframe spheres.

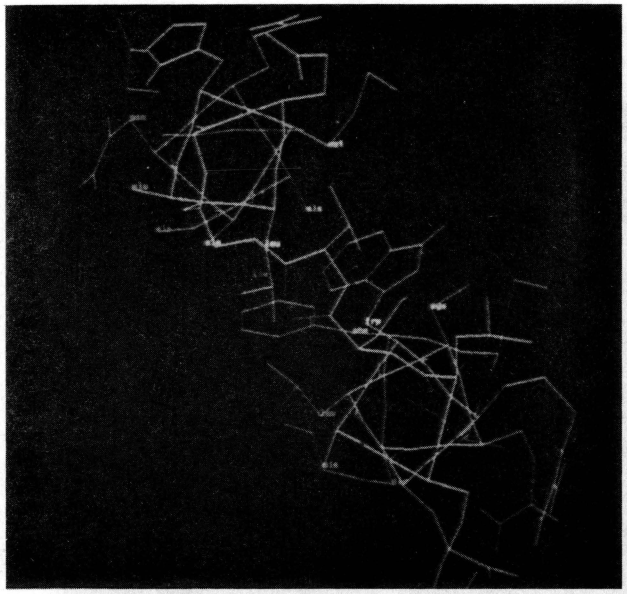


Plate 3: Two helices after rotations.