Rigid-Body Alignment

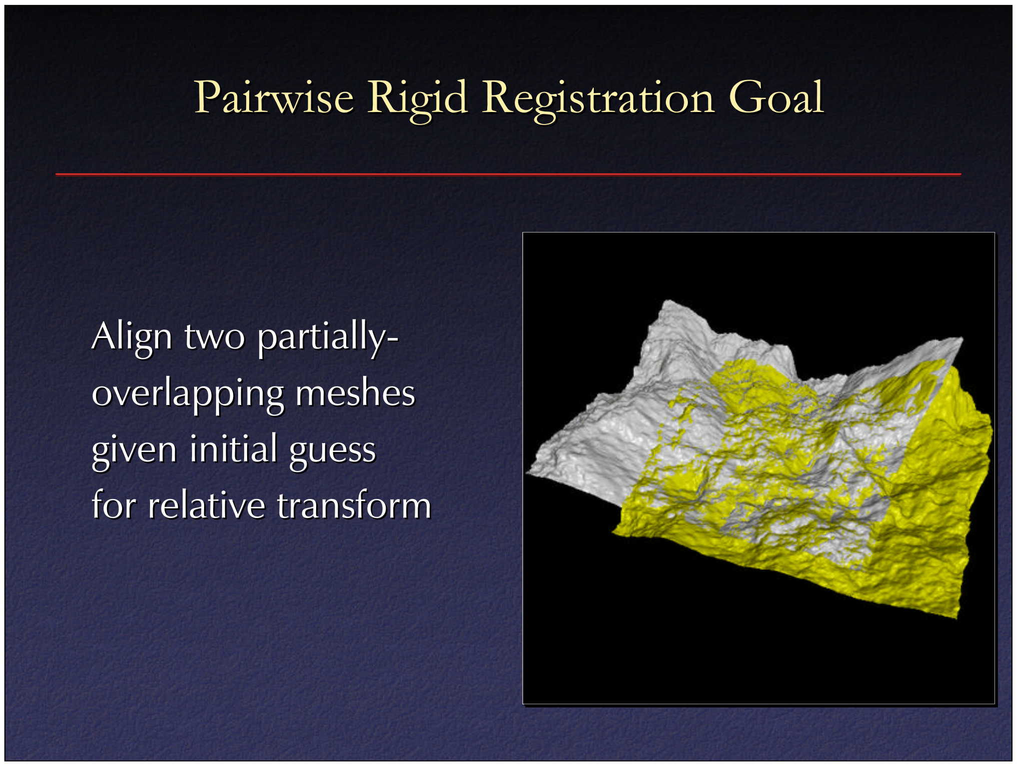
3D Scan Matching and Registration, Part II

ICCV 2005 Short Course

Szymon Rusinkiewicz

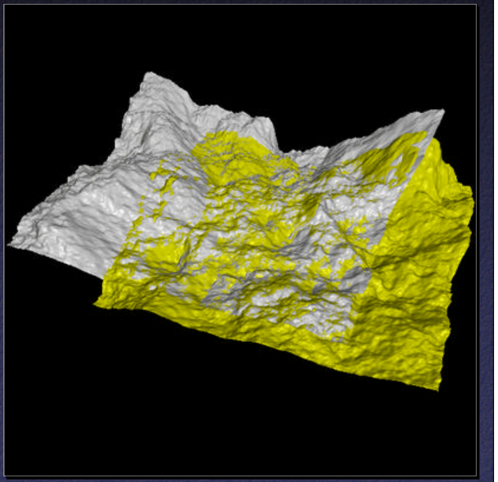
Princeton University

This section of the course covers techniques for pairwise (i.e., scan-to-scan) and "global" (i.e., involving more than 2 scans) alignment, given that the algorithms are constrained to obtain a rigid-body transformation.

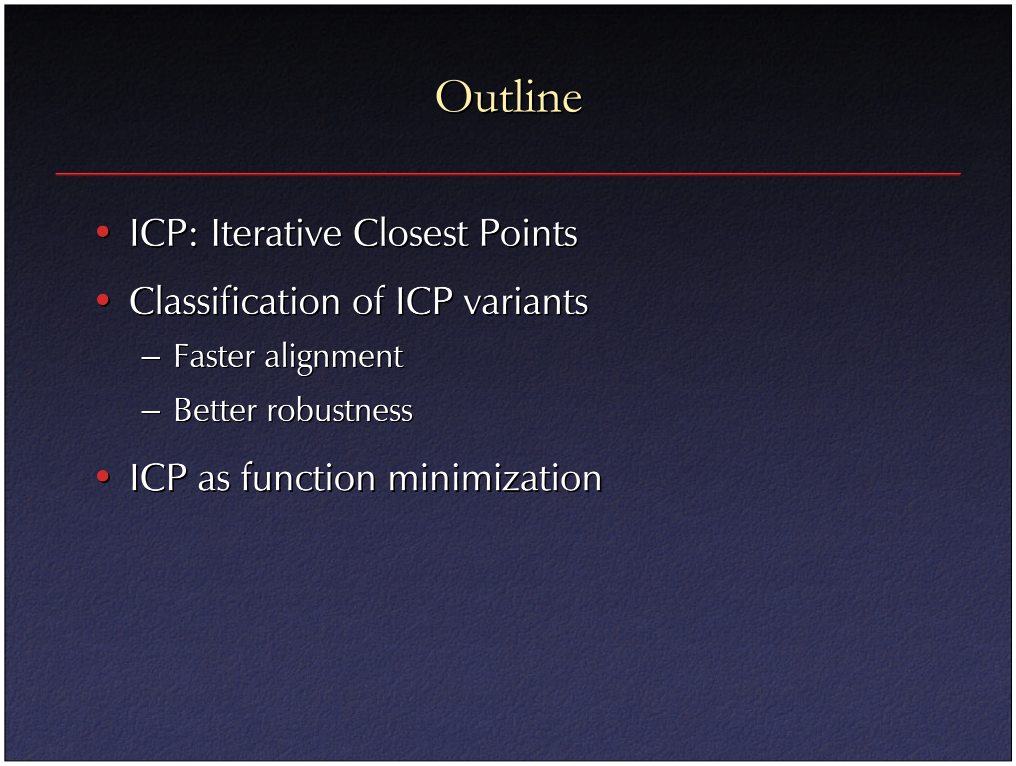


Pairwise Rigid Registration Goal

Align two partially-overlapping meshes given initial guess for relative transform



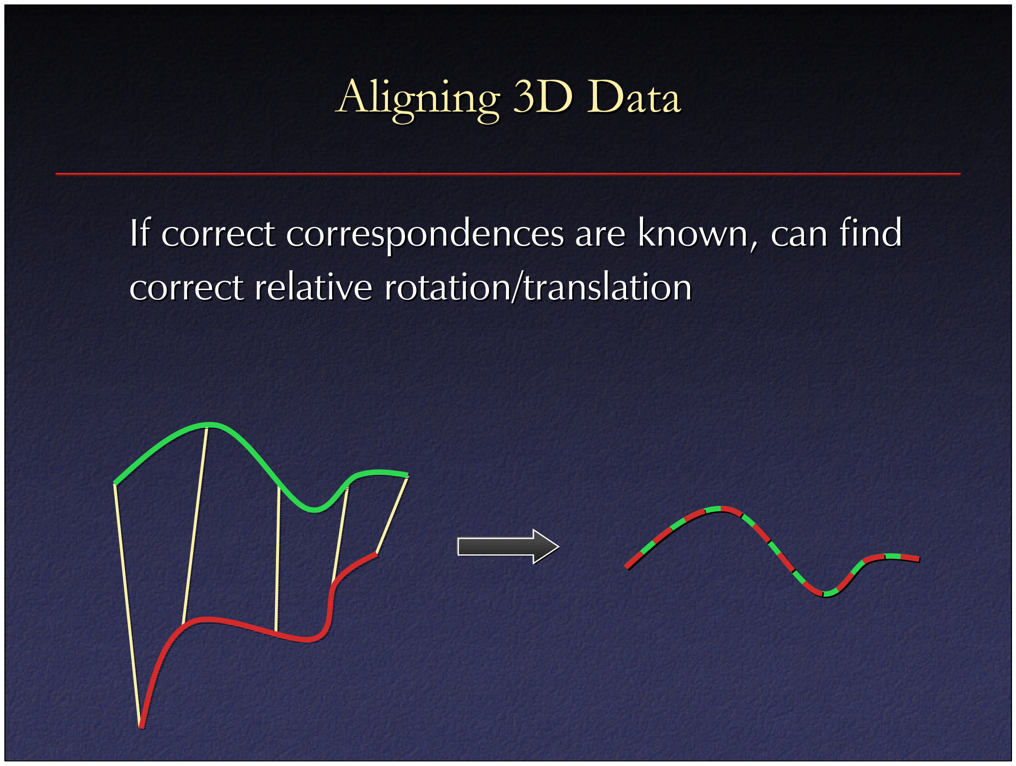
So, let’s formalize the rigid registration problem as solving for a *rigid-body* transform (i.e., translation and rotation, or 6 total degrees of freedom in 3D) that minimizes the distance between two partially-overlapping meshes. We’ll focus on iterative algorithms that converge to a local minimum, so we’ll assume that we have an initial guess already.



Outline

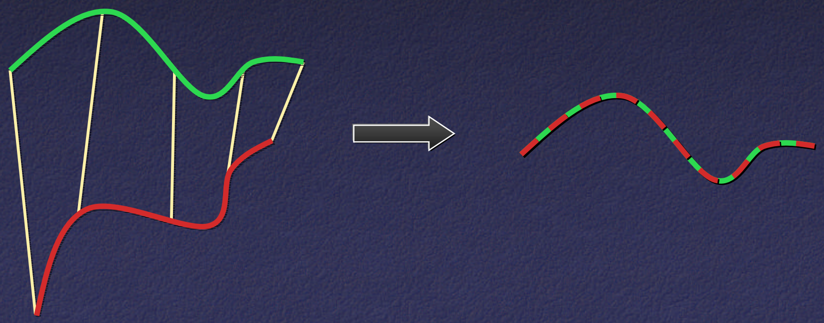
- ICP: Iterative Closest Points
- Classification of ICP variants
  - Faster alignment
  - Better robustness
- ICP as function minimization

We'll be talking about the best-known such algorithm, ICP, together with some variants that make it faster or more robust.

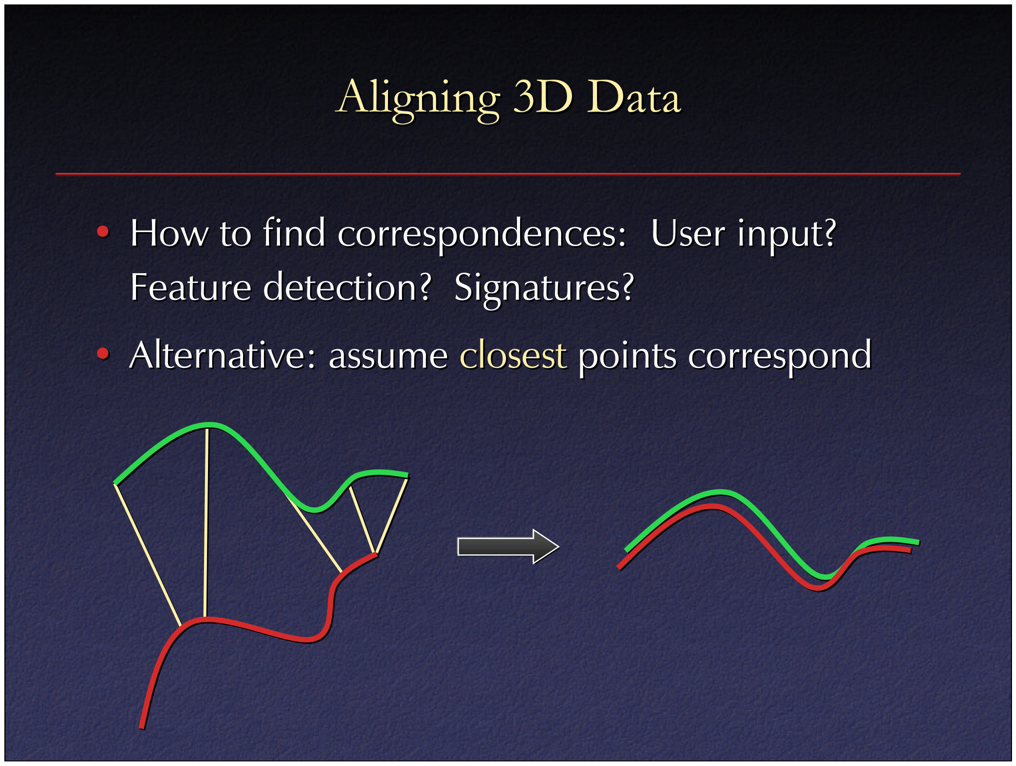


Aligning 3D Data

If correct correspondences are known, can find correct relative rotation/translation



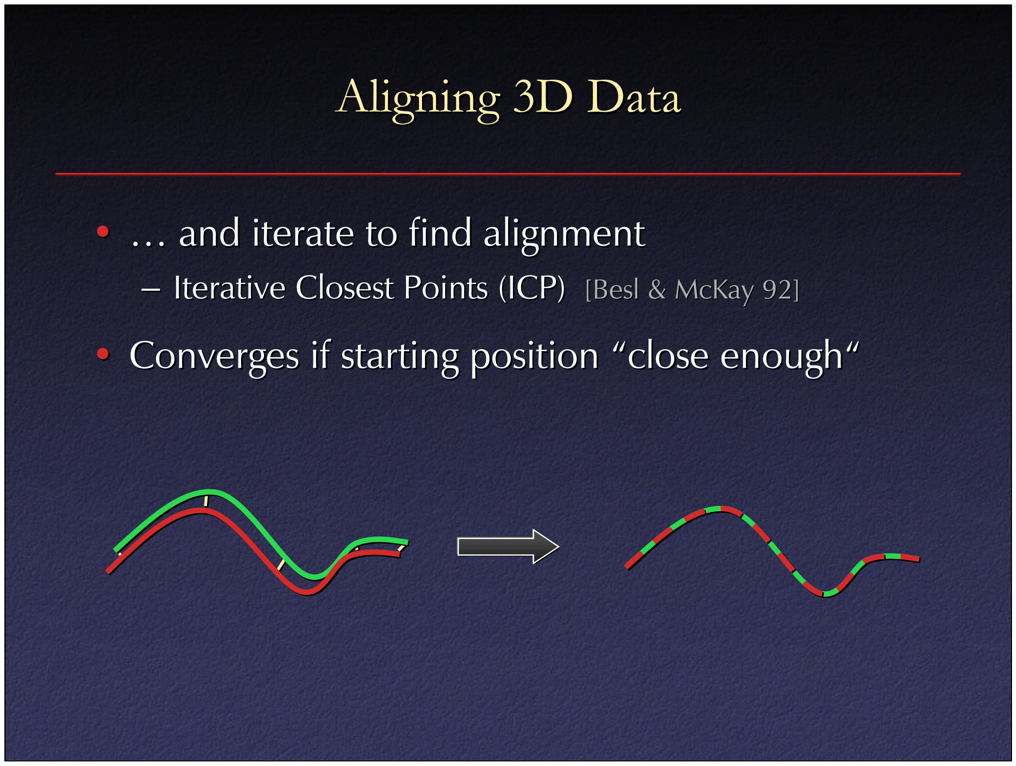
The basic idea behind ICP is that, if we somehow knew correspondences, we could solve for the translation that minimizes pairwise distance.



Aligning 3D Data

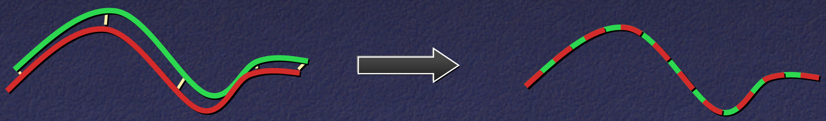
- How to find correspondences: User input? Feature detection? Signatures?
- Alternative: assume closest points correspond

What ICP does is make a seemingly-radical guess: that *closest* points are the correspondences. You take these, run the point pairs through the minimization algorithms, and get… the wrong answer. However, if you started out reasonably close, this process got you closer…

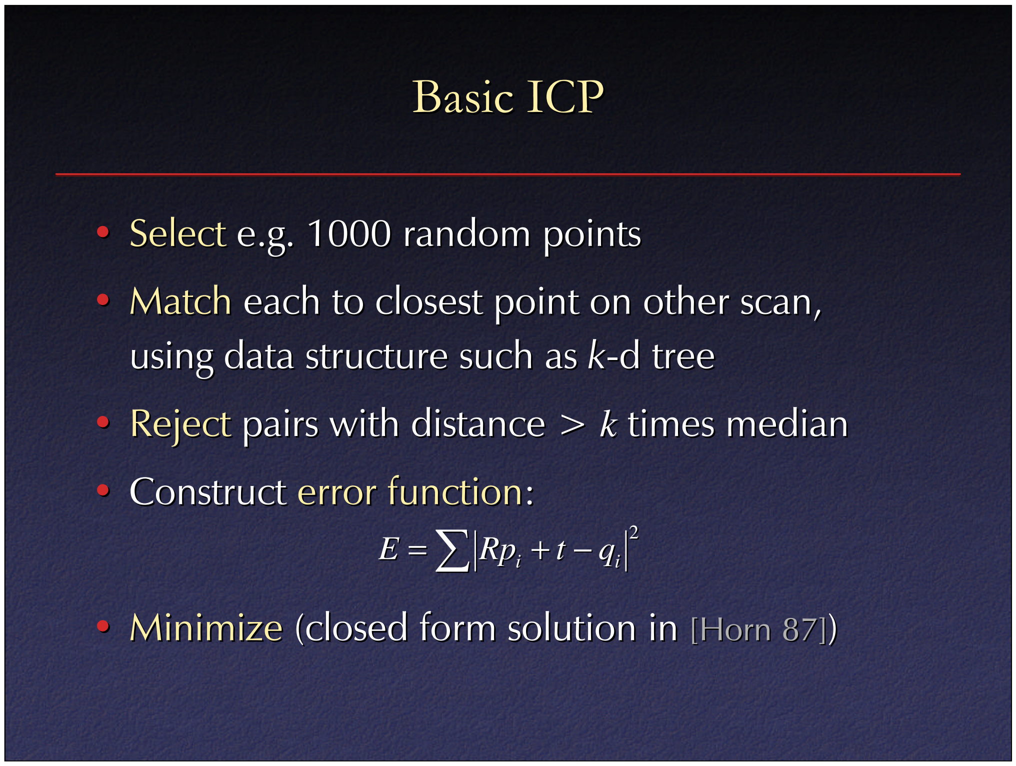


Aligning 3D Data

- … and iterate to find alignment
  - Iterative Closest Points (ICP) [Besl & McKay 92]
- Converges if starting position "close enough"



… which means that you can repeat the process and converge to the right answer.



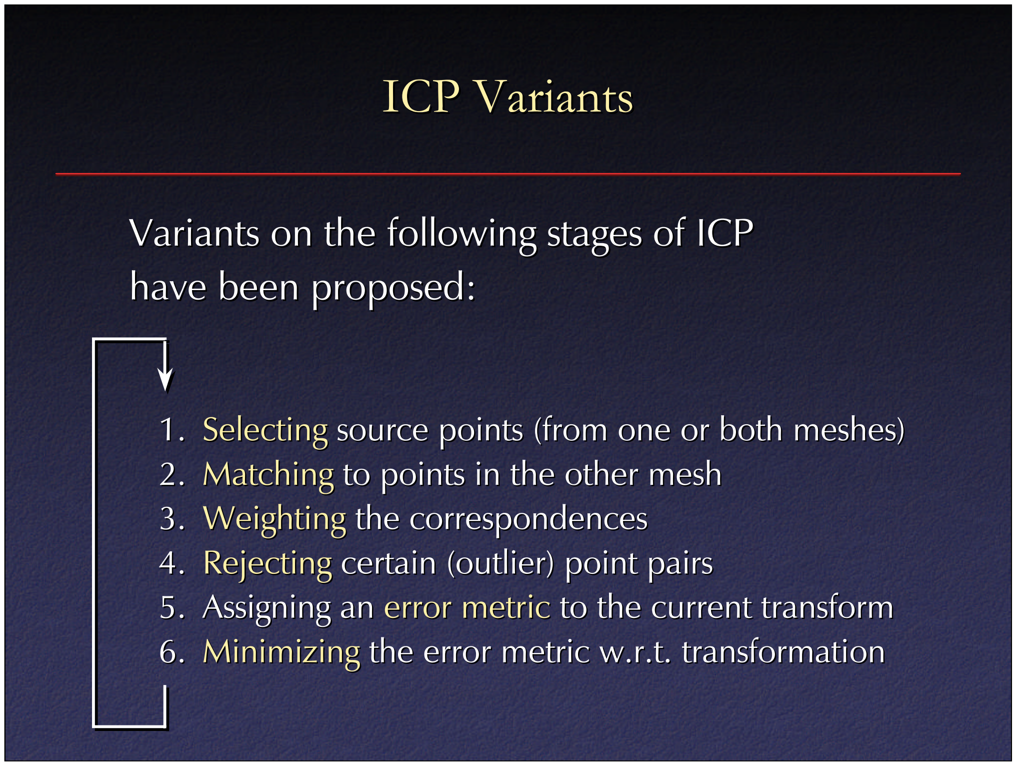
Basic ICP

- Select e.g. 1000 random points
- Match each to closest point on other scan, using data structure such as *k*-d tree
- Reject pairs with distance > *k* times median
- Construct error function:

$$E = \sum \left| Rp_i + t - q_i \right|^2$$

- Minimize (closed form solution in [Horn 87])

So, here's a slightly formalized version of that algorithm.

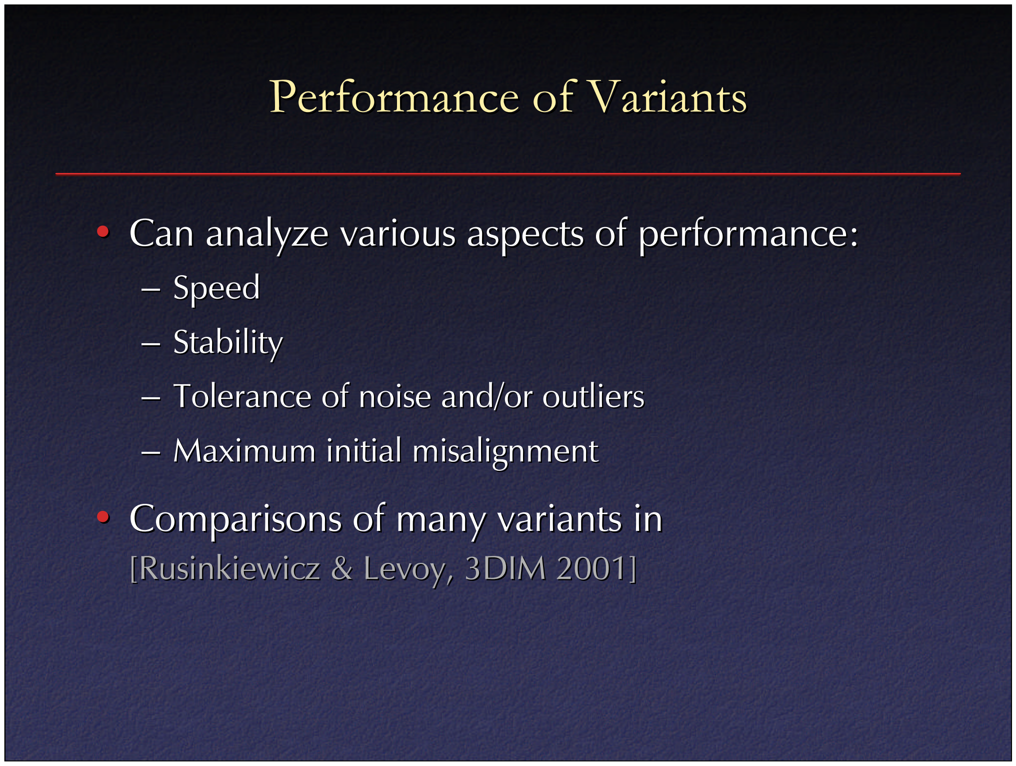


ICP Variants

Variants on the following stages of ICP have been proposed:

1. Selecting source points (from one or both meshes)
2. Matching to points in the other mesh
3. Weighting the correspondences
4. Rejecting certain (outlier) point pairs
5. Assigning an error metric to the current transform
6. Minimizing the error metric w.r.t. transformation

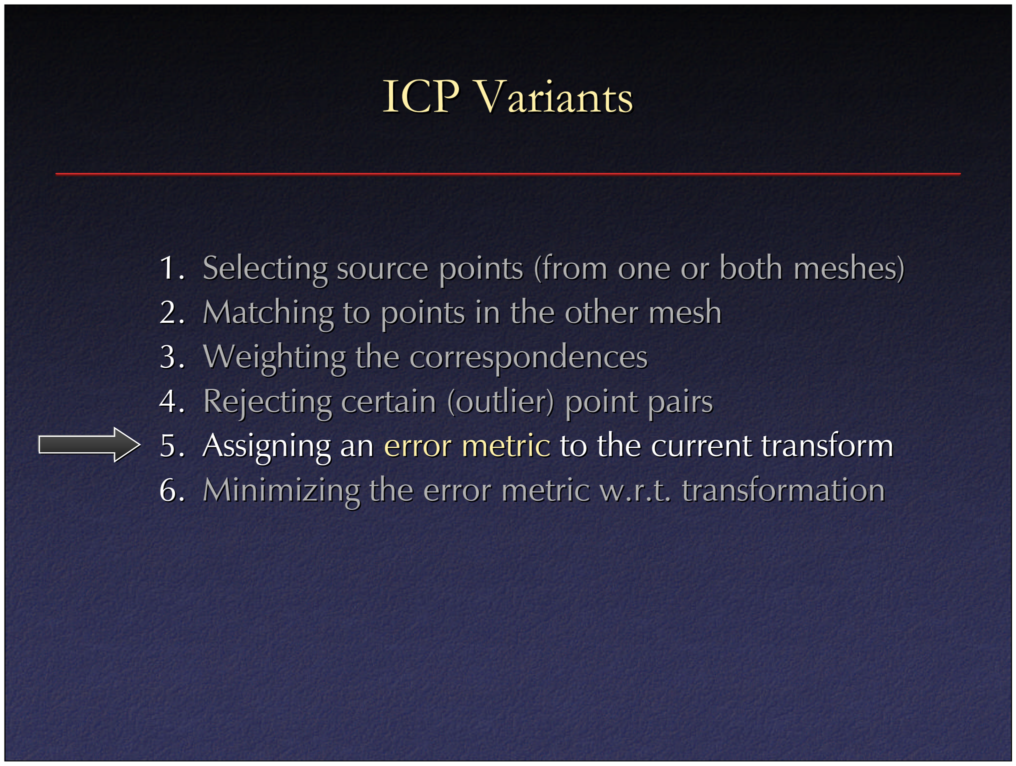
If we look at the preceeding algorithm, we can see that it consists of a bunch of different stages, and there are different choices you can make at each stage. It turns out that there’s been a lot of research on ICP and, to a good approximation, you can classify all the different tweaks to ICP that people have proposed into variants on 6 different stages.



Performance of Variants

- Can analyze various aspects of performance:
  - Speed
  - Stability
  - Tolerance of noise and/or outliers
  - Maximum initial misalignment
- Comparisons of many variants in [Rusinkiewicz & Levoy, 3DIM 2001]

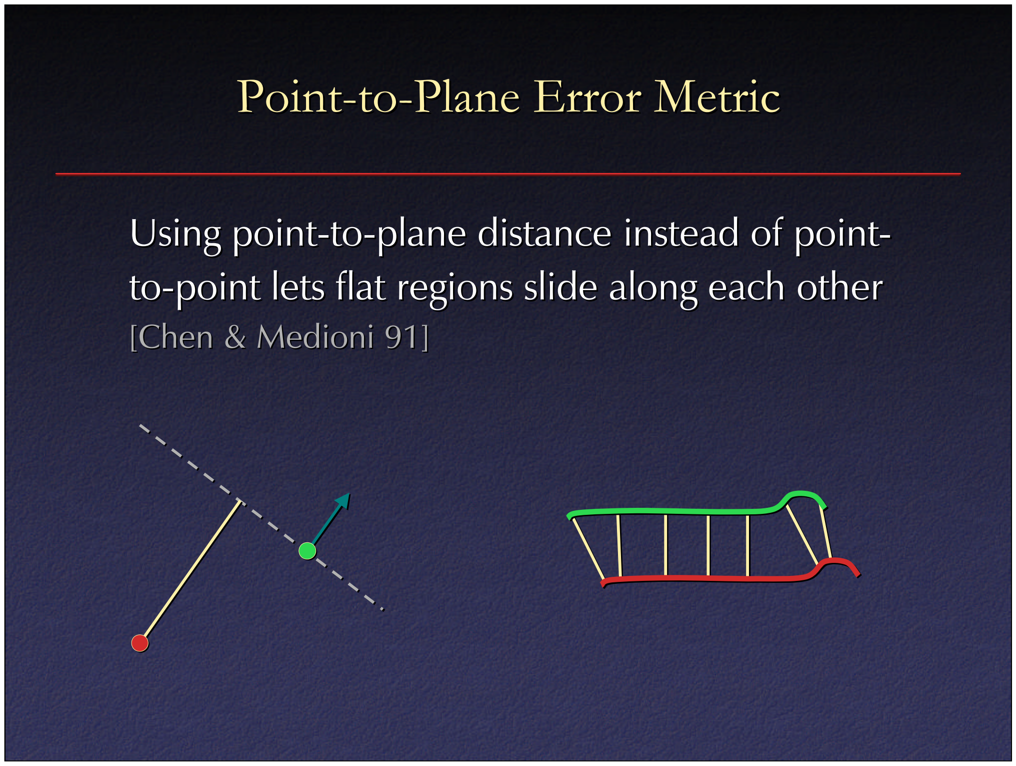
Each variant has some effect on the speed or robustness of the method, and there are a few cocktails of variants that have proven reasonably popular over the years. The next bit of this course is going to be about looking at those.



ICP Variants

1. Selecting source points (from one or both meshes)
2. Matching to points in the other mesh
3. Weighting the correspondences
4. Rejecting certain (outlier) point pairs
5. Assigning an error metric to the current transform
6. Minimizing the error metric w.r.t. transformation

The first thing we'll talk about is a change that makes a huge (an order of magnitude or more) difference in the convergence of ICP. It's a change in the "error metric."

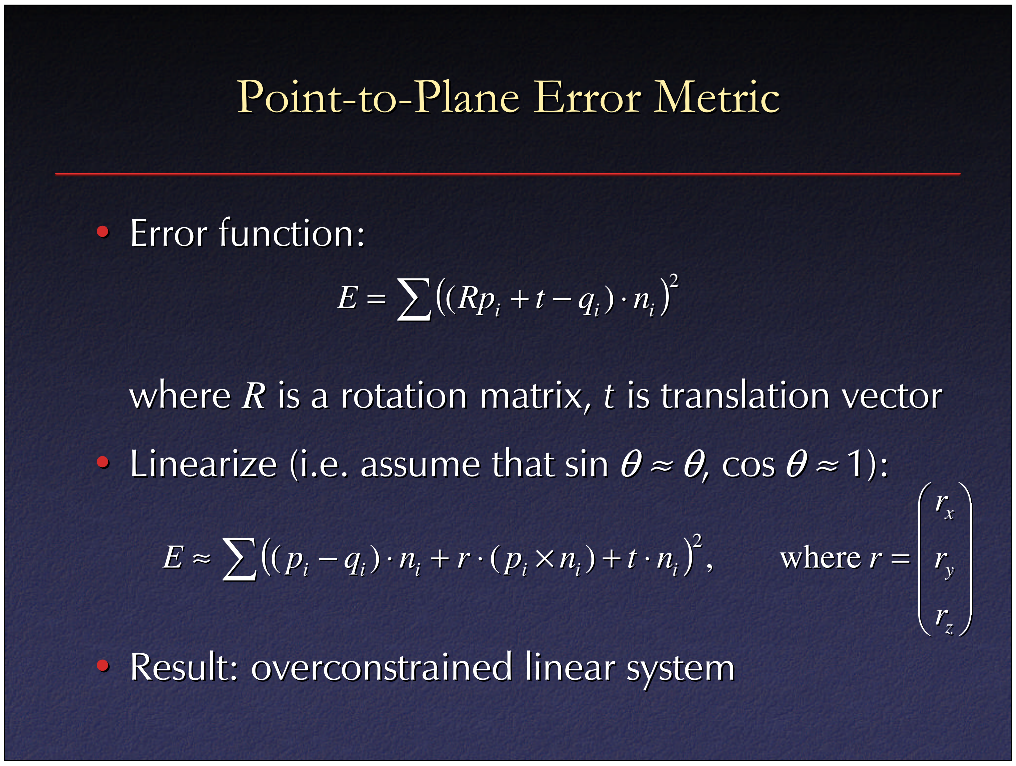


Point-to-Plane Error Metric

Using point-to-plane distance instead of point-to-point lets flat regions slide along each other [Chen & Medioni 91]

The idea is that instead of minimizing the distance between pairs of points, we'll minimize the distance between one of the points and the plane passing through the other and perpendicular to its normal. (This, of course, assumes that you've estimated normals, which is typically done just by fitting planes to small neighborhoods of points, etc.)

The big advantage of this is that it better captures the notion that sliding two planes along each other doesn't increase the distance between them. Consider, for example, the situation shown at right. You have two scans that are mostly planar, with only the little bump constraining their left-and-right motion. Using the original point-to-point distance, however, the pairs of points in the flat region will prevent the scans from "sliding along each other" to reach the correct transformation: think of them as little springs that want to minimize their length. Using the point-to-plane distance in this case will lead to significantly faster convergence.



Point-to-Plane Error Metric

- Error function:

$$E = \sum \left((Rp_i + t - q_i) \cdot n_i \right)^2$$

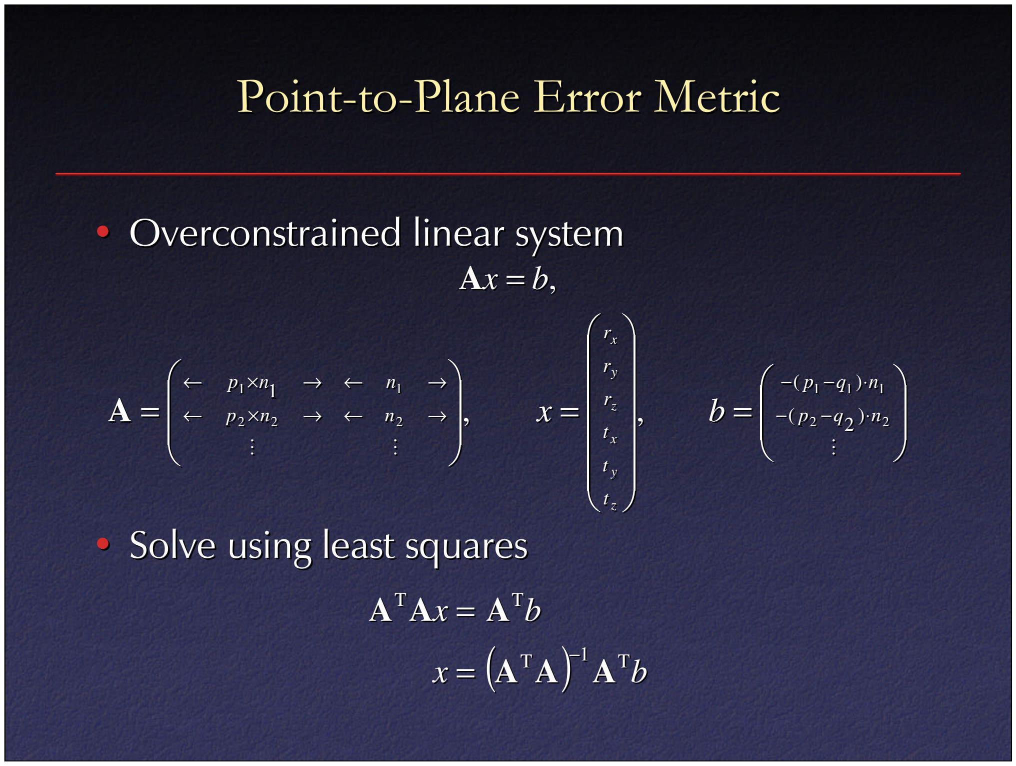
  where R is a rotation matrix, t is translation vector
- Linearize (i.e. assume that $\sin\theta \approx \theta$, $\cos\theta \approx 1$):

$$E \approx \sum \left((p_i - q_i) \cdot n_i + r \cdot (p_i \times n_i) + t \cdot n_i \right)^2, \qquad \text{where } r = \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix}$$

- Result: overconstrained linear system

So, how do we actually implement this? The first thing is to write down the error function: we're taking the transformed version of point p_i, finding the vector to q_i, then taking just the component of that vector along the normal n_i. This is a little nasty, since the matrix R involves some trig functions, etc., and minimizing this error function directly (i.e., solving for the components of R and t) can't be done in closed form. So, we're going to *make* the problem linear by assuming that the angles we need to solve for are small, so we can use the usual first-order approximations to sine and cosine. This lets us rewrite the energy function in a way that's linear in the unknowns, and solve it using standard linear least squares.

In practice, making the assumption that angles are small is not too bad: the angles *will* be small in later iterations, while during early iterations the error of this approximation tends to be dominated by the fact that the correspondences aren't really correct… In short, it works really well in practice.



Point-to-Plane Error Metric

- Overconstrained linear system

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

$$\mathbf{A} = \begin{pmatrix} \leftarrow & p_1 \times n_1 & \rightarrow & \leftarrow & n_1 & \rightarrow \\ \leftarrow & p_2 \times n_2 & \rightarrow & \leftarrow & n_2 & \rightarrow \\ & \vdots & & & \vdots & \end{pmatrix}, \quad x = \begin{pmatrix} r_x \\ r_y \\ r_z \\ t_x \\ t_y \\ t_z \end{pmatrix}, \quad b = \begin{pmatrix} -(p_1 - q_1) \cdot n_1 \\ -(p_2 - q_2) \cdot n_2 \\ \vdots \end{pmatrix}$$

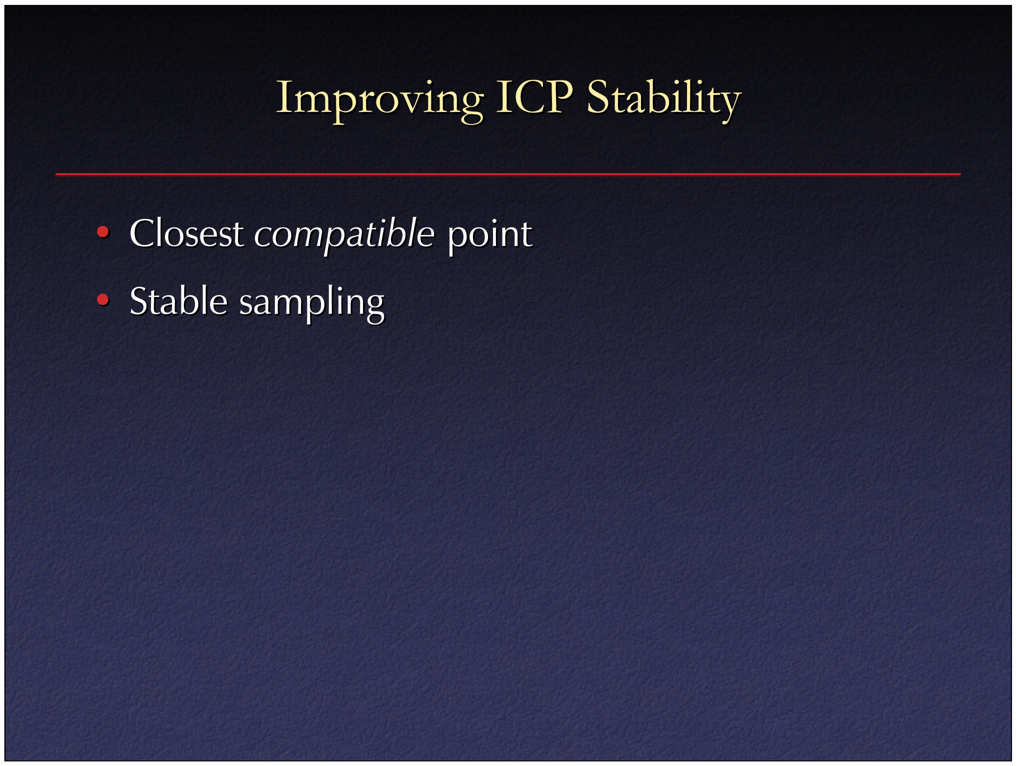
- Solve using least squares

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}x = \mathbf{A}^{\mathrm{T}}b$$

$$x = \left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\mathbf{A}^{\mathrm{T}}b$$

So, how do we actually implement this? The first thing is to write down the error function: we're taking the transformed version of point p_i, finding the vector to q_i, then taking just the component of that vector along the normal n_i. This is a little nasty, since the matrix R involves some trig functions, etc., and minimizing this error function directly (i.e., solving for the components of R and t) can't be done in closed form. So, we're going to *make* the problem linear by assuming that the angles we need to solve for are small, so we can use the usual first-order approximations to sine and cosine. This lets us rewrite the energy function in a way that's linear in the unknowns, and solve it using standard linear least squares.

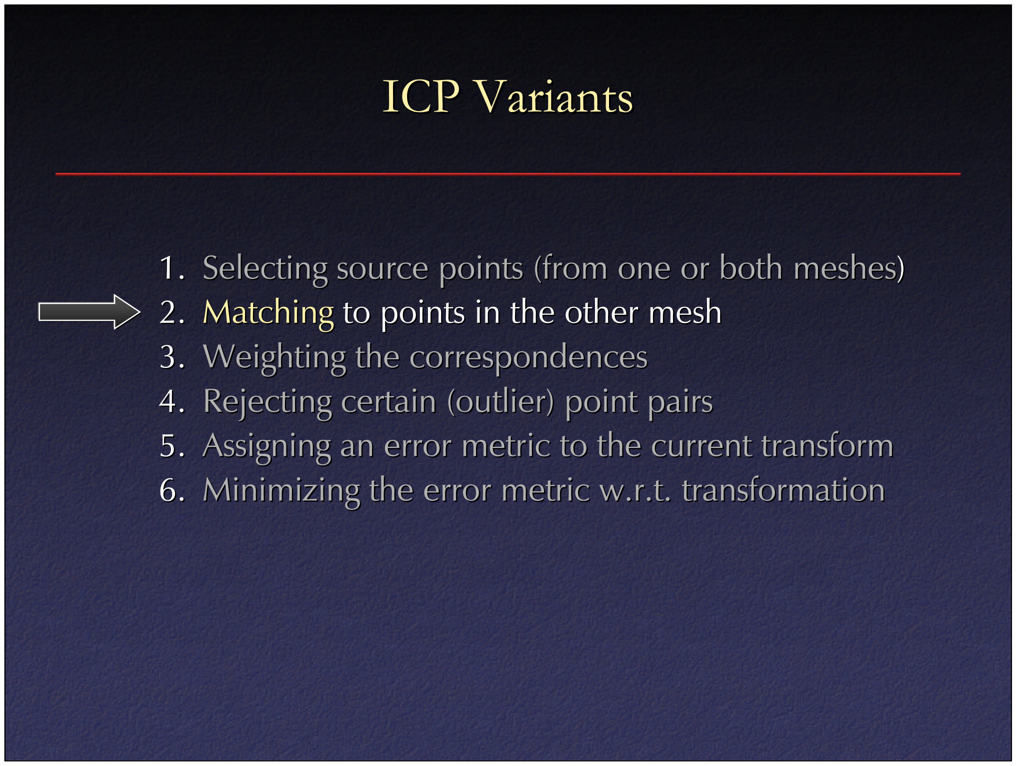
In practice, making the assumption that angles are small is not too bad: the angles *will* be small in later iterations, while during early iterations the error of this approximation tends to be dominated by the fact that the correspondences aren't really correct… In short, it works really well in practice.



Improving ICP Stability

- Closest *compatible* point
- Stable sampling

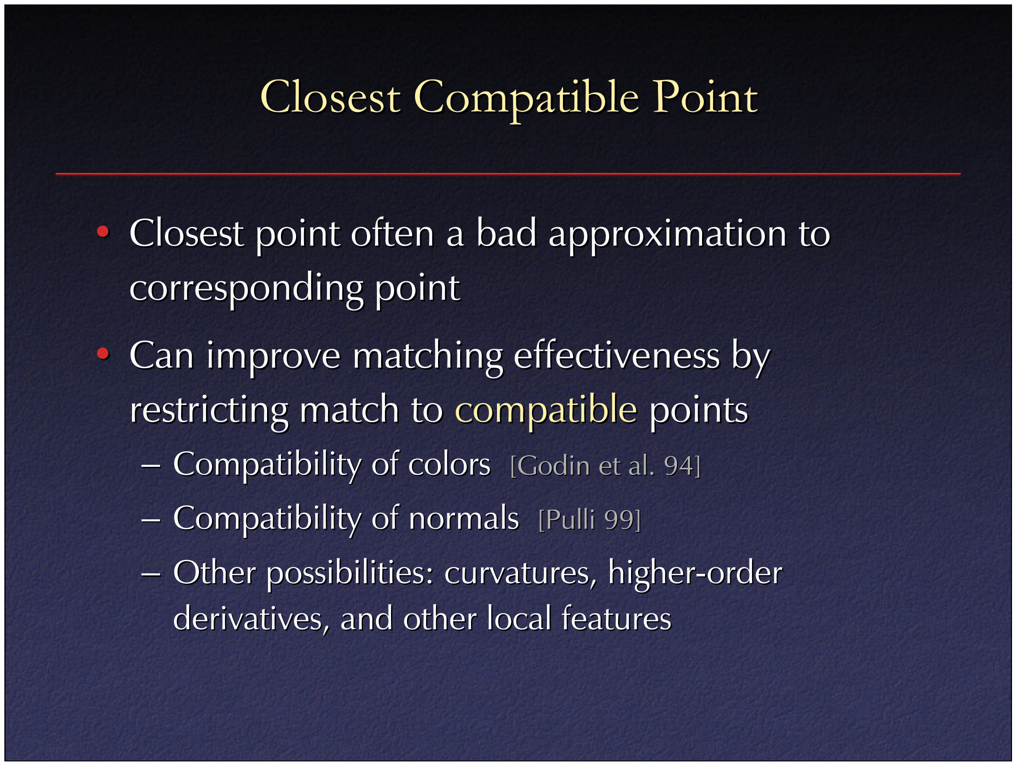
The next set of ICP variants we'll talk about concern making ICP more stable.



ICP Variants

1. Selecting source points (from one or both meshes)
2. Matching to points in the other mesh
3. Weighting the correspondences
4. Rejecting certain (outlier) point pairs
5. Assigning an error metric to the current transform
6. Minimizing the error metric w.r.t. transformation

The first of these is a change to the matching stage of ICP.

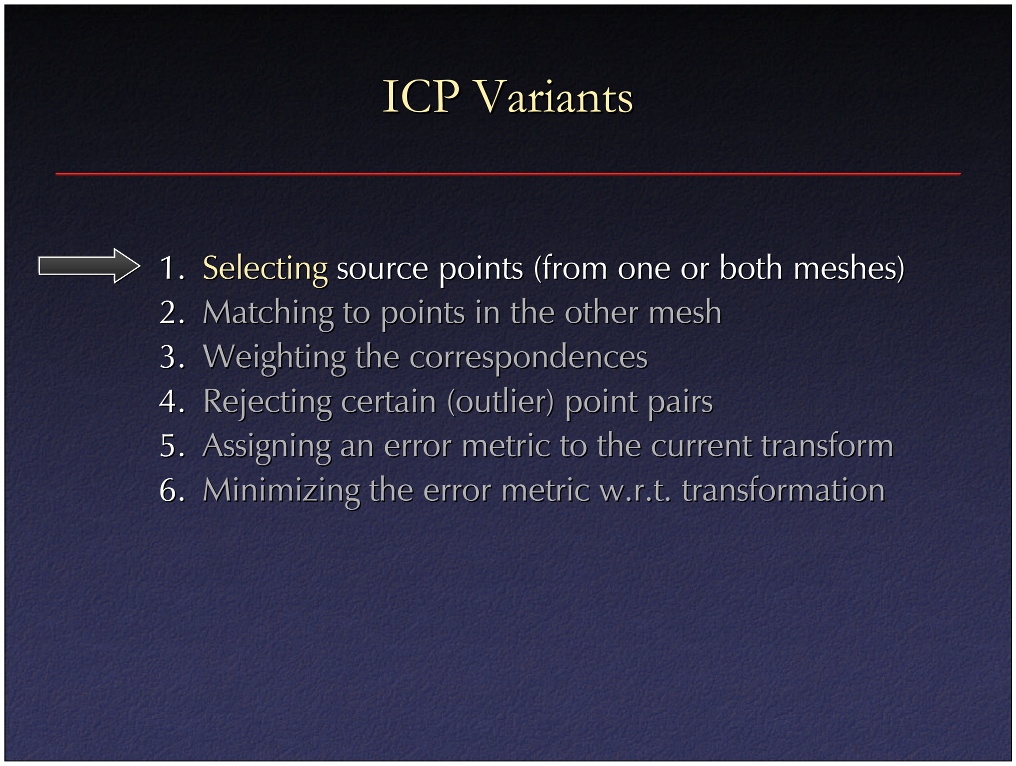


Closest Compatible Point

- Closest point often a bad approximation to corresponding point
- Can improve matching effectiveness by restricting match to compatible points
  - Compatibility of colors [Godin et al. 94]
  - Compatibility of normals [Pulli 99]
  - Other possibilities: curvatures, higher-order derivatives, and other local features

The idea is that instead of just blindly choosing the closest point as your correspondence, you instead try to find better ones. One way of phrasing this is as selecting the closest point that is *compatible* with the source point, by some metric(s). This can be done based on color, normals, or any of the features covered earlier.

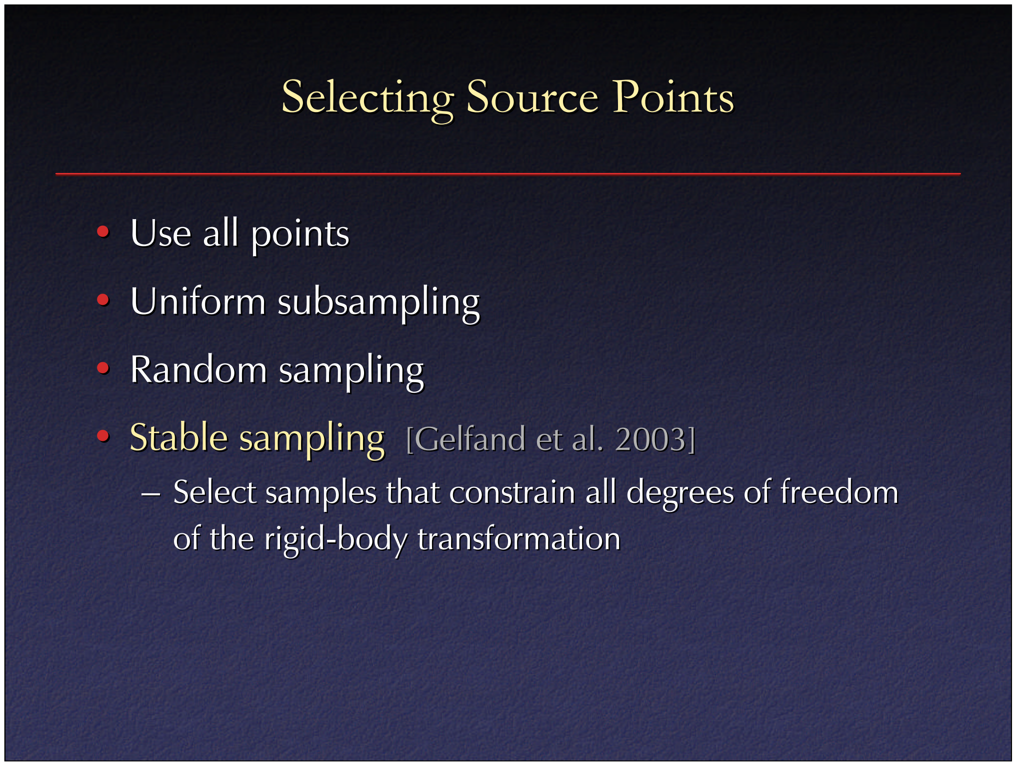
Note that there's a different way of incorporating more information into the matching process, which is to directly do the matching in a higher-dimensional space, where 3 of the dimensions are just Euclidean coordinates, while the remaining dimensions correspond to your feature space. This certainly can be made to work, but you sometimes encounter difficulties relating to how the different dimensions are scaled. That is, there's no natural way of saying e.g. how a distance of 3mm in space should be weighted relative to a distance of 10 in a color coordinate. In practice, you need to use something like a Mahalanobis distance (i.e., give the dimensions a weight computed using variances (and covariances) of the data itself).



ICP Variants

1. Selecting source points (from one or both meshes)
2. Matching to points in the other mesh
3. Weighting the correspondences
4. Rejecting certain (outlier) point pairs
5. Assigning an error metric to the current transform
6. Minimizing the error metric w.r.t. transformation

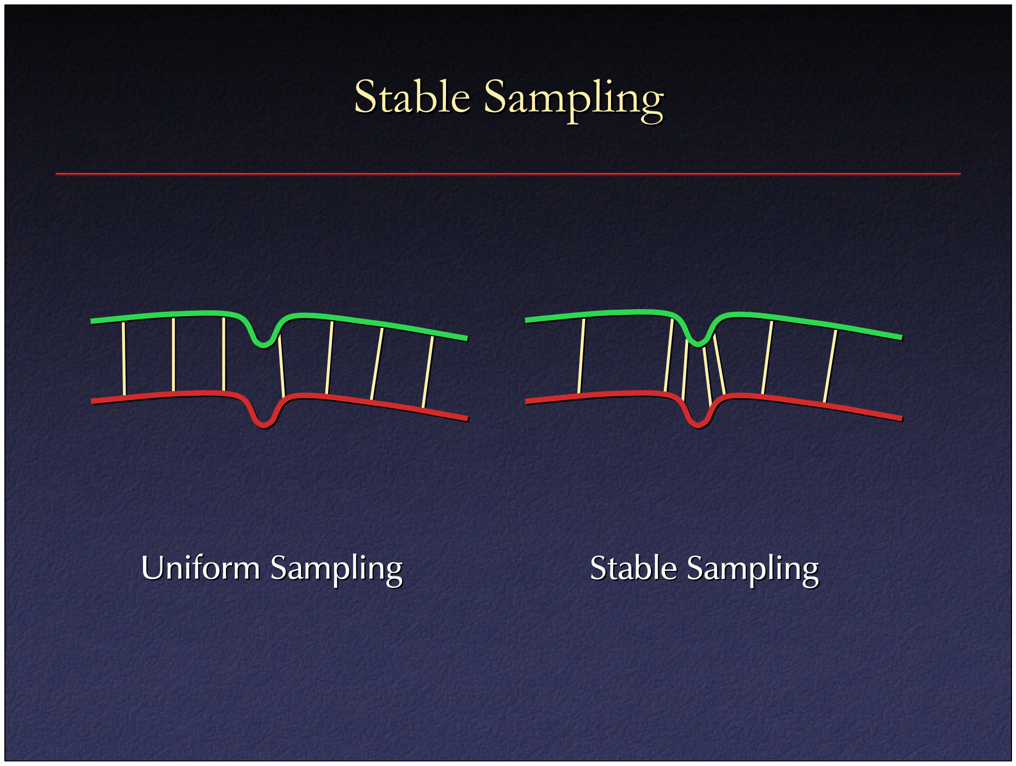
Let’s move on to a tweak you can make to the “selection” stage of ICP to give you better performance in difficult cases.



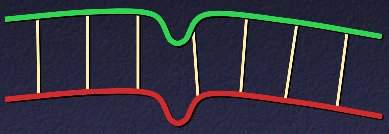
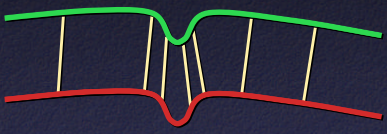
Selecting Source Points

- Use all points
- Uniform subsampling
- Random sampling
- Stable sampling [Gelfand et al. 2003]
  - Select samples that constrain all degrees of freedom of the rigid-body transformation

The idea is to select samples on the scan that are somehow the most relevant to constraining all the different degrees of freedom of the transformation you are solving for.



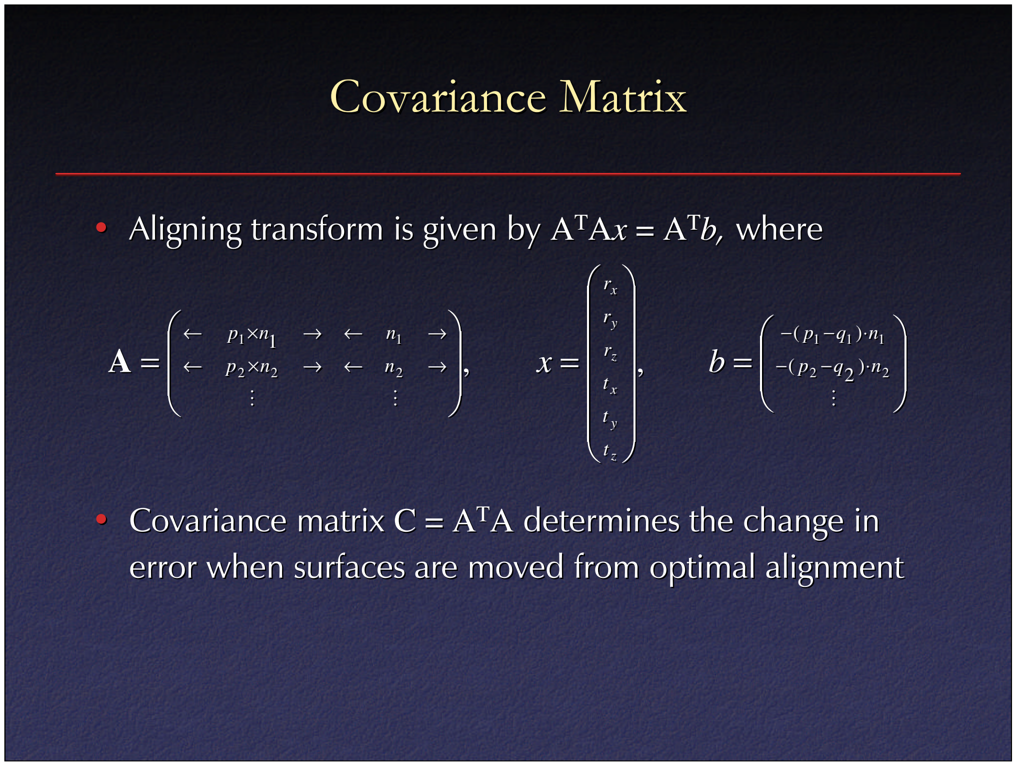
Stable Sampling

Uniform Sampling

Stable Sampling

So, if you had this situation, you'd want to recognize that the up-and-down component of the motion is well-constrained by almost any point you choose. In contrast, the left-and-right component of the transformation is really only established by points on the little bump in the center. Random sampling might put few (or no) points on this bump, and hence be unstable. So, we need some method for putting more points on this feature.



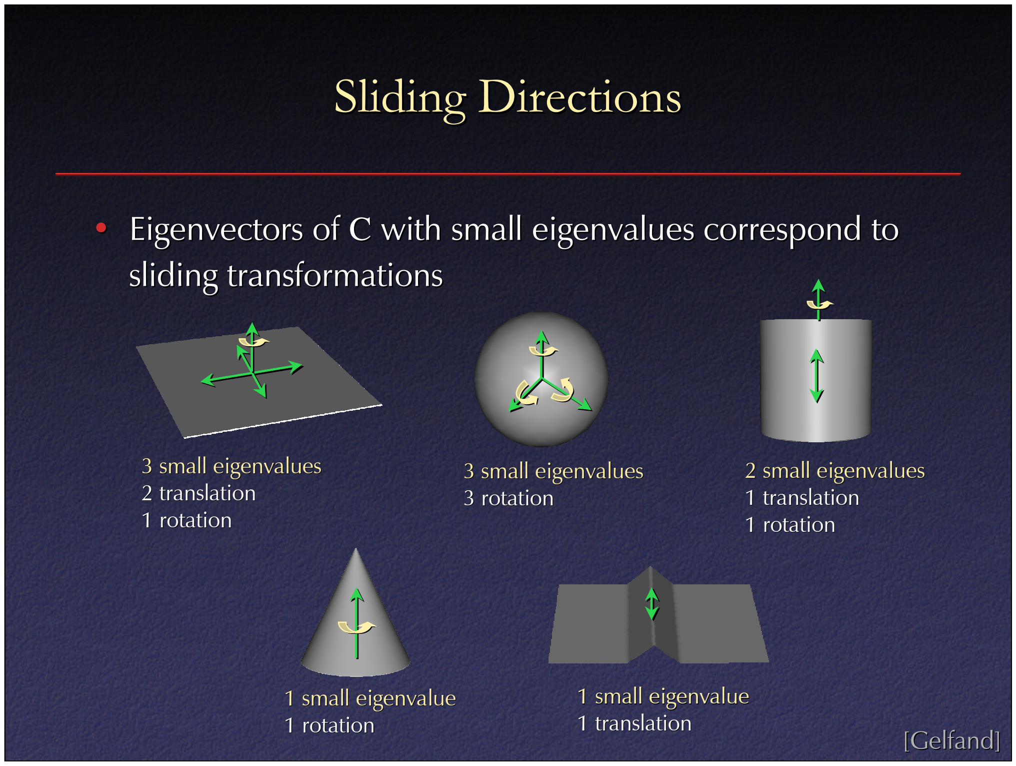
Covariance Matrix

- Aligning transform is given by $A^T A x = A^T b$, where

$$\mathbf{A} = \begin{pmatrix} \leftarrow & p_1 \times n_1 & \rightarrow & \leftarrow & n_1 & \rightarrow \\ \leftarrow & p_2 \times n_2 & \rightarrow & \leftarrow & n_2 & \rightarrow \\ & \vdots & & & \vdots & \end{pmatrix}, \quad x = \begin{pmatrix} r_x \\ r_y \\ r_z \\ t_x \\ t_y \\ t_z \end{pmatrix}, \quad b = \begin{pmatrix} -(p_1 - q_1) \cdot n_1 \\ -(p_2 - q_2) \cdot n_2 \\ \vdots \end{pmatrix}$$

- Covariance matrix $\mathbf{C} = A^T A$ determines the change in error when surfaces are moved from optimal alignment

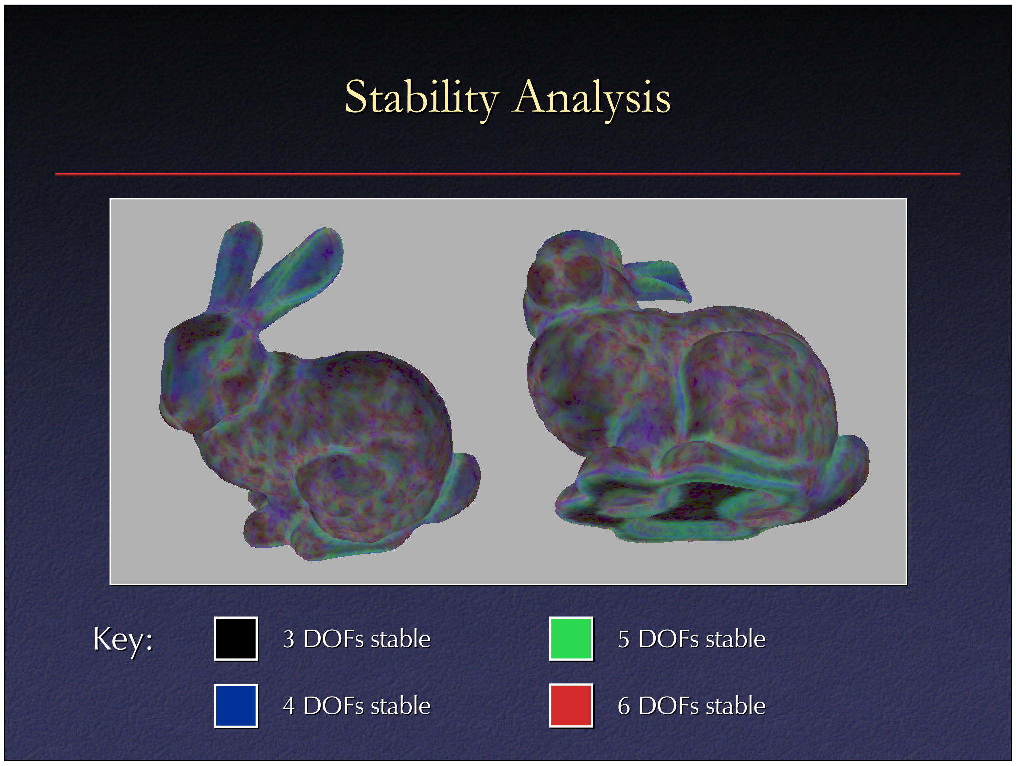
To do this, we need to go back and look at what happens when you solve for the transformation using the linearized point-to-plane framework we saw earlier. Since this is a linear least squares problem, the crucial thing is the "covariance" matrix of the data, which encodes the shape of the error landscape around the minimum.



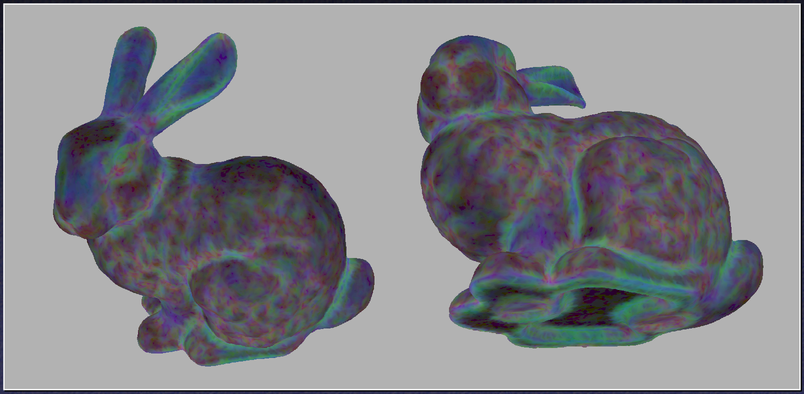
Sliding Directions

- Eigenvectors of C with small eigenvalues correspond to sliding transformations

The eigenvalues of *C*, together with their corresponding eigenvectors, encode how stable particular transformations will be: eigenvectors with *large* eigenvalues correspond to transformations that are *stable*. Here are a few examples of shapes that lead to *small* eigenvalues of *C*. In each case, the number of small eigenvalues corresponds to the number of degrees of freedom of motion that are not well-constrained by the geometry: these are ways in which the object can "slip" along itself.



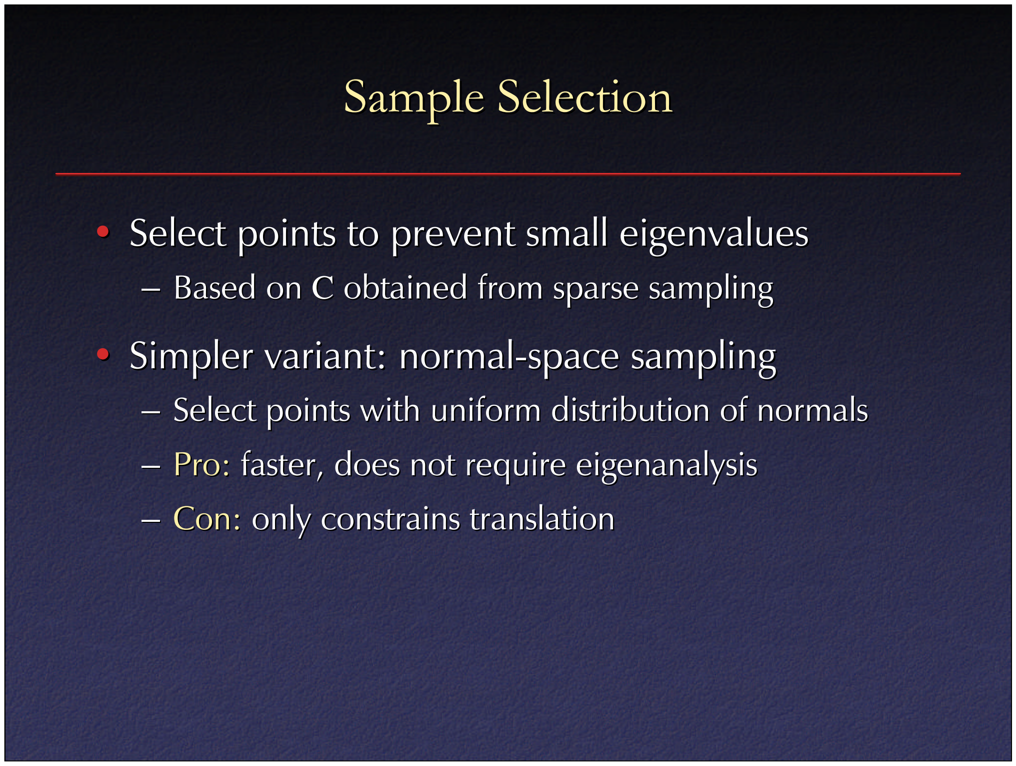
Stability Analysis



Key: ⬛ 3 DOFs stable 🟩 5 DOFs stable

🟦 4 DOFs stable 🟥 6 DOFs stable

Here's a color-coded visualization of this on a larger mesh, just looking at the stability of small neighborhoods of points. Notice that it makes a lot of sense: locally-cylindrical areas are unstable along 1 degree of freedom, etc. Of course, since we're aligning whole scans instead of small neighborhoods this sort of local analysis doesn't exactly tell the story of what will and will not be stable, but it turns out that we'll encounter this local stability again when we talk about nonrigid alignment.



Sample Selection

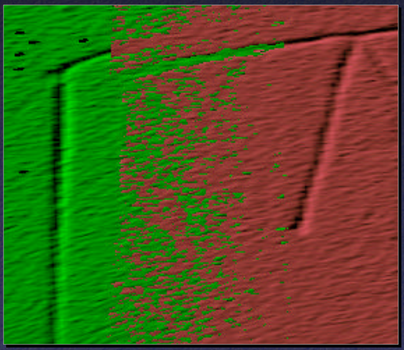
- Select points to prevent small eigenvalues
  - Based on C obtained from sparse sampling
- Simpler variant: normal-space sampling
  - Select points with uniform distribution of normals
  - Pro: faster, does not require eigenanalysis
  - Con: only constrains translation

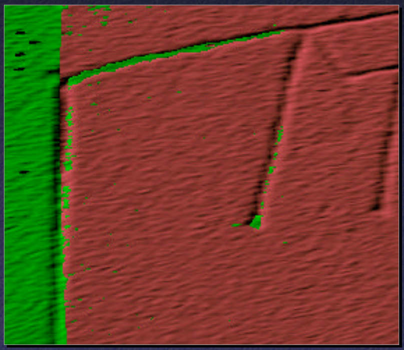
So, now that we can analyze stability we can incorporate it into the sampling stage of ICP. We want to constrain *C* to have no small eigenvalues, so we proceed as follows:

1. Figure out how bad the situation would be if we just used random sampling. To do this, we estimate what *C* would be using just a small uniform sampling of points.
2. For each point, we figure out how much stability it is contributing to the transformation corresponding to each eigenvector of *C*. We form sorted lists for each eigenvector, based on the points that constrain it the most.
3. Now, to do the “real” sampling, we draw points from the different lists, keeping track of how the different eigenvectors have been constrained. When we notice that one eigenvector is underconstrained, we draw points from its list.

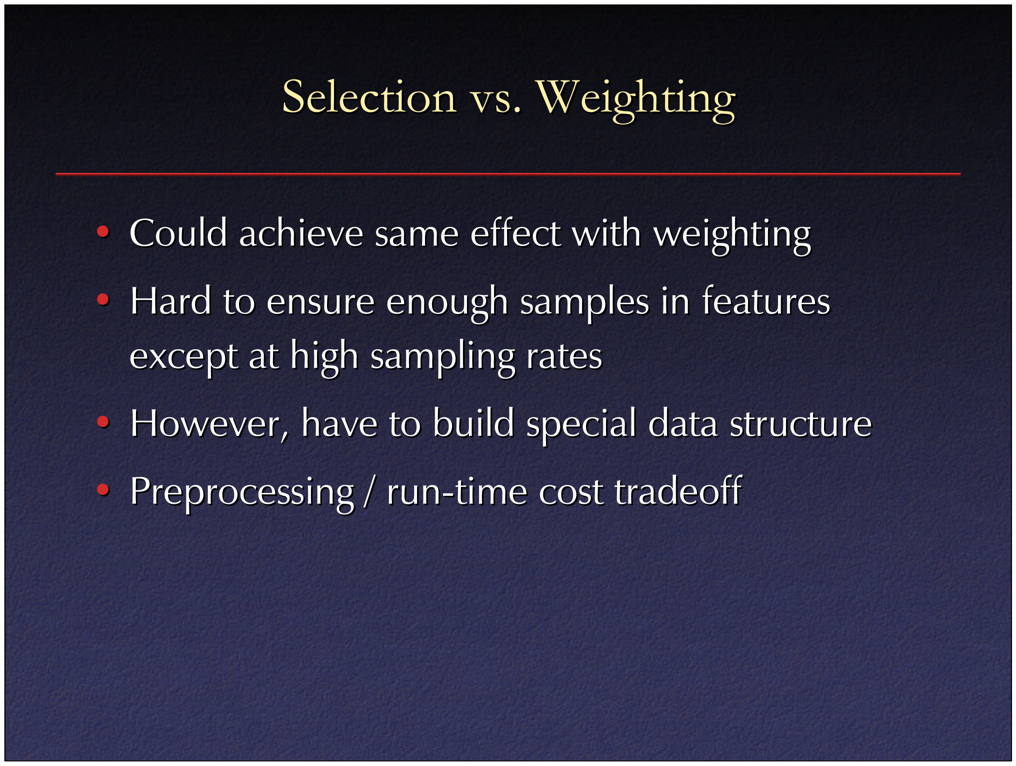
Result

Stability-based or normal-space sampling important for smooth areas with small features


Random sampling


Normal-space sampling

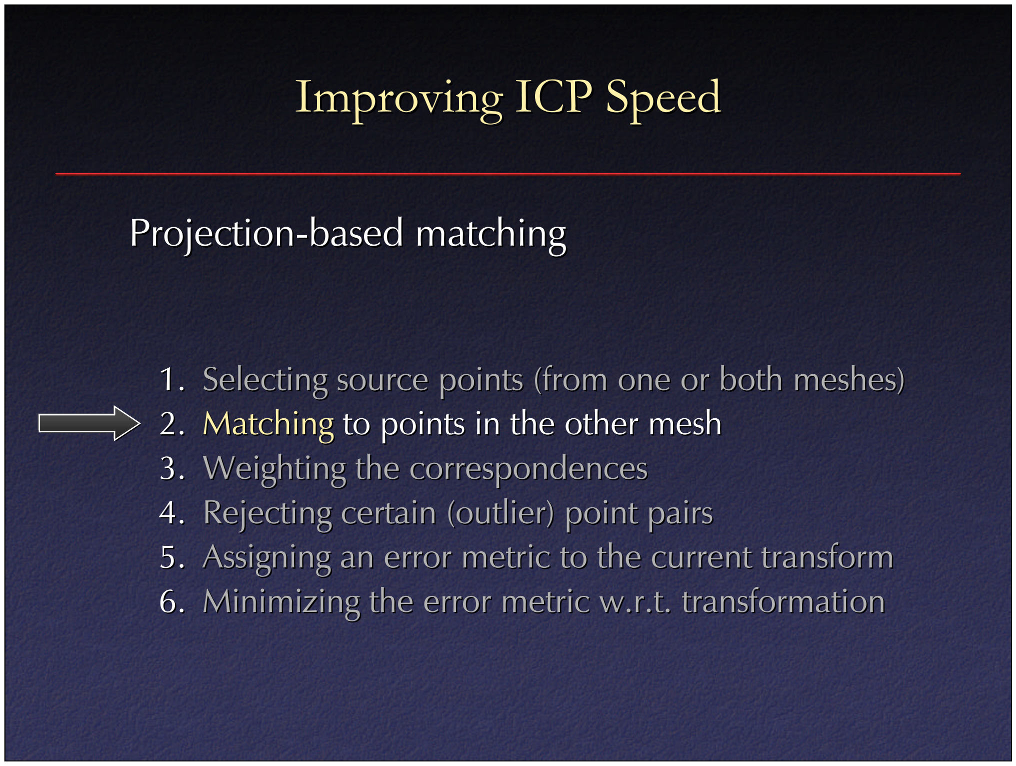
Here's a result of the normal-space sampling on real data. This was a slab with engraved grooves on the front, and a combination of noise and warp (due to miscalibration) caused regular ICP to converge to the wrong transformation. In contrast, the normal-space sampling got the grooves to line up between scans.



Selection vs. Weighting

- Could achieve same effect with weighting
- Hard to ensure enough samples in features except at high sampling rates
- However, have to build special data structure
- Preprocessing / run-time cost tradeoff

Note that if we had just taken more samples (to ensure that we got enough samples where we needed them), then we could have gotten the same effect by fudging the “weighting” stage of ICP rather than the point selection. This represents a pretty classic preprocessing / run time tradeoff.

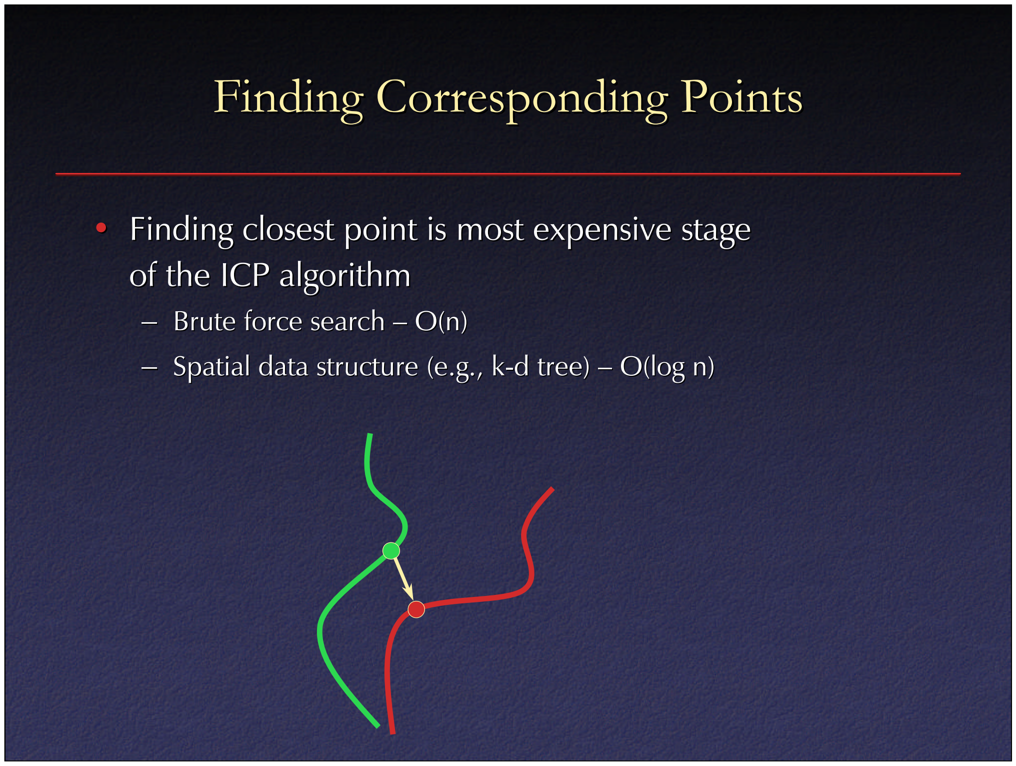


Improving ICP Speed

Projection-based matching

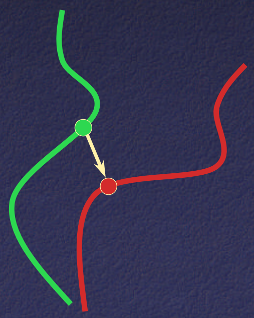
1. Selecting source points (from one or both meshes)
2. Matching to points in the other mesh
3. Weighting the correspondences
4. Rejecting certain (outlier) point pairs
5. Assigning an error metric to the current transform
6. Minimizing the error metric w.r.t. transformation

Now let's look at an ICP variant that's all about speed. It changes the matching stage.

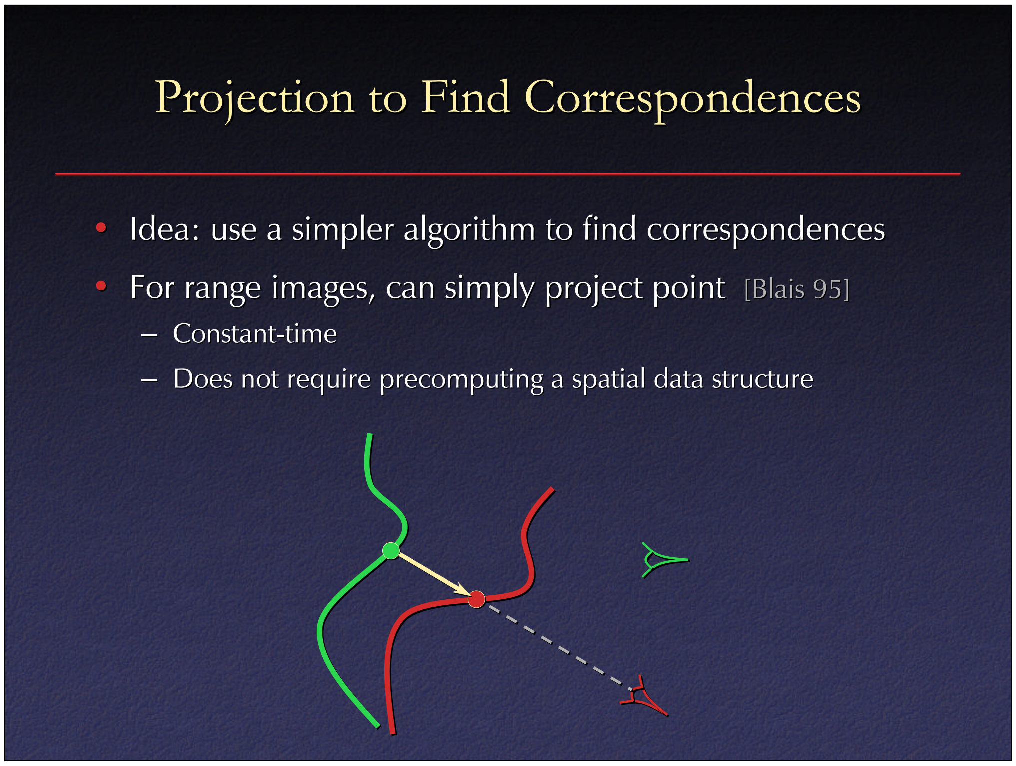


Finding Corresponding Points

- Finding closest point is most expensive stage of the ICP algorithm
  - Brute force search – O(n)
  - Spatial data structure (e.g., k-d tree) – O(log n)

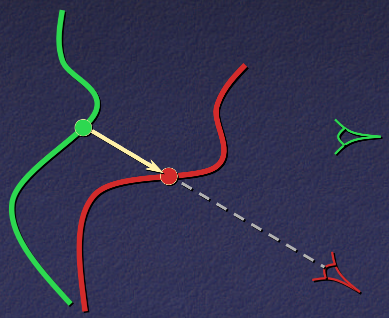


This variant is based on the observation that the slowest stage of most ICP algorithms is actually finding the closest point. Typically a data structure such as a k-d tree is used to accelerate the search, but it’s still not that fast and of course there is the preprocessing time required to build that data structure in the first place.

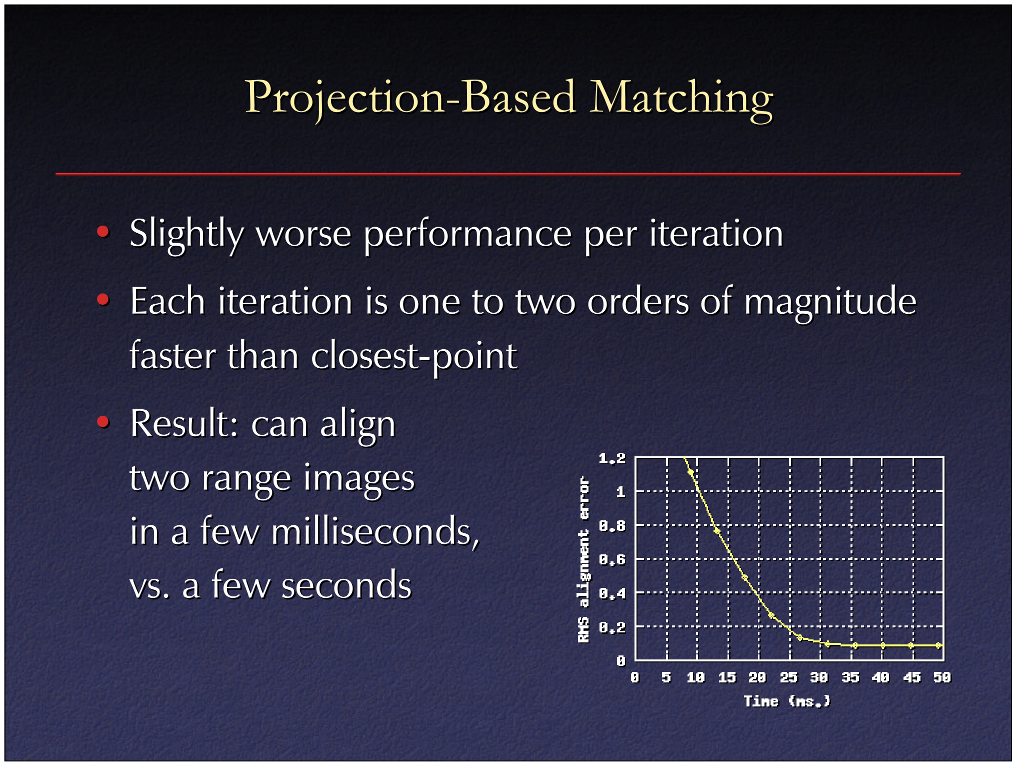


Projection to Find Correspondences

- Idea: use a simpler algorithm to find correspondences
- For range images, can simply project point [Blais 95]
  - Constant-time
  - Does not require precomputing a spatial data structure

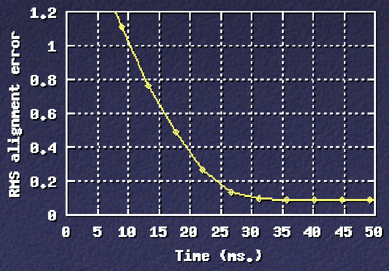


However, we can make the observation that using the closest point is wrong anyway. So, why not be a little more wrong as long as it's fast? For range images (i.e., scans that represent each point as a "height" or "distance" on a regularly-sampled grid) it is possible to just use the result of projecting a point onto the grid as the "correspondence". This is very fast (a bit of math and a single array lookup), but of course can be pretty far from the point you really wanted, depending on how far away you are from the right transformation. So, this type of algorithm is most useful in cases where you are starting pretty close to the right answer already.

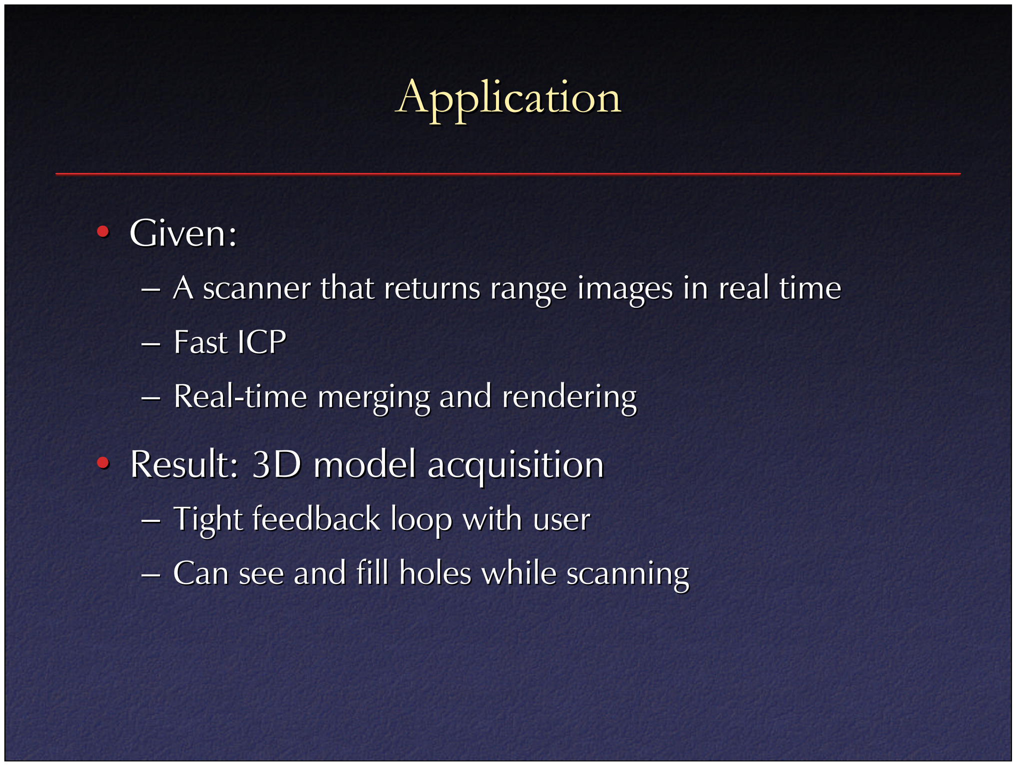


Projection-Based Matching

- Slightly worse performance per iteration
- Each iteration is one to two orders of magnitude faster than closest-point
- Result: can align two range images in a few milliseconds, vs. a few seconds



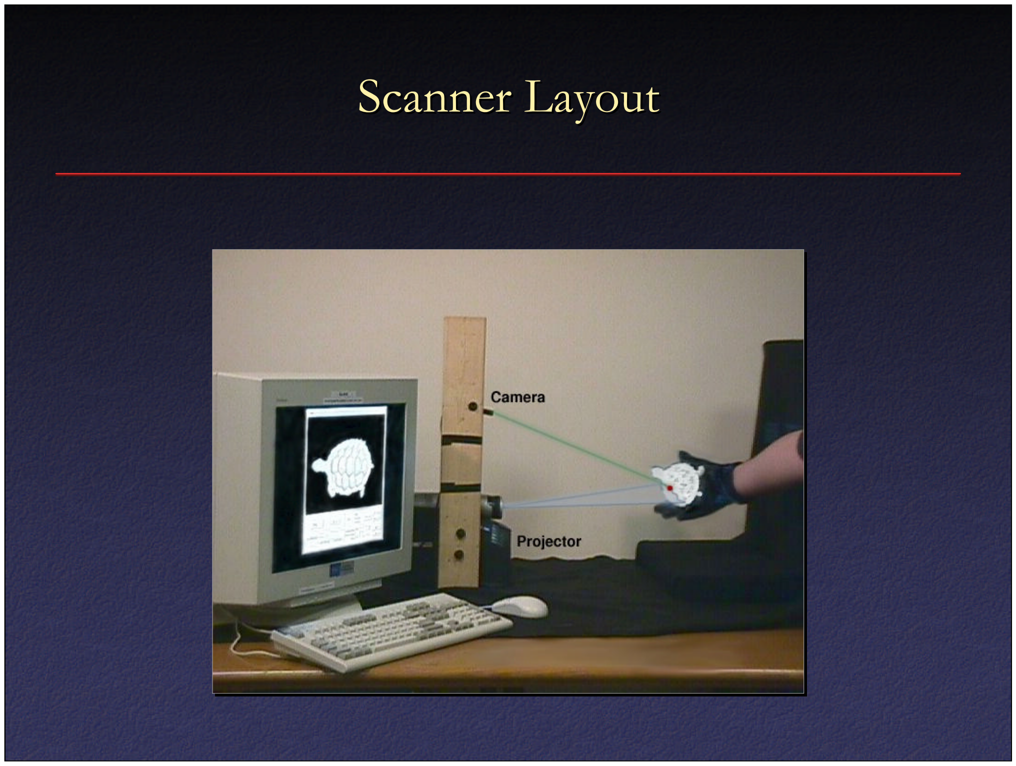
The net result is that alignment can be done in milliseconds per pair of meshes (these timings, by the way, are on 5-year-old hardware).



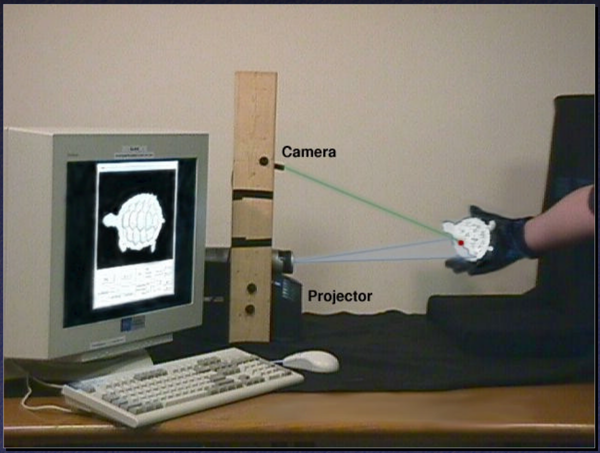
Application

- Given:
  - A scanner that returns range images in real time
  - Fast ICP
  - Real-time merging and rendering
- Result: 3D model acquisition
  - Tight feedback loop with user
  - Can see and fill holes while scanning

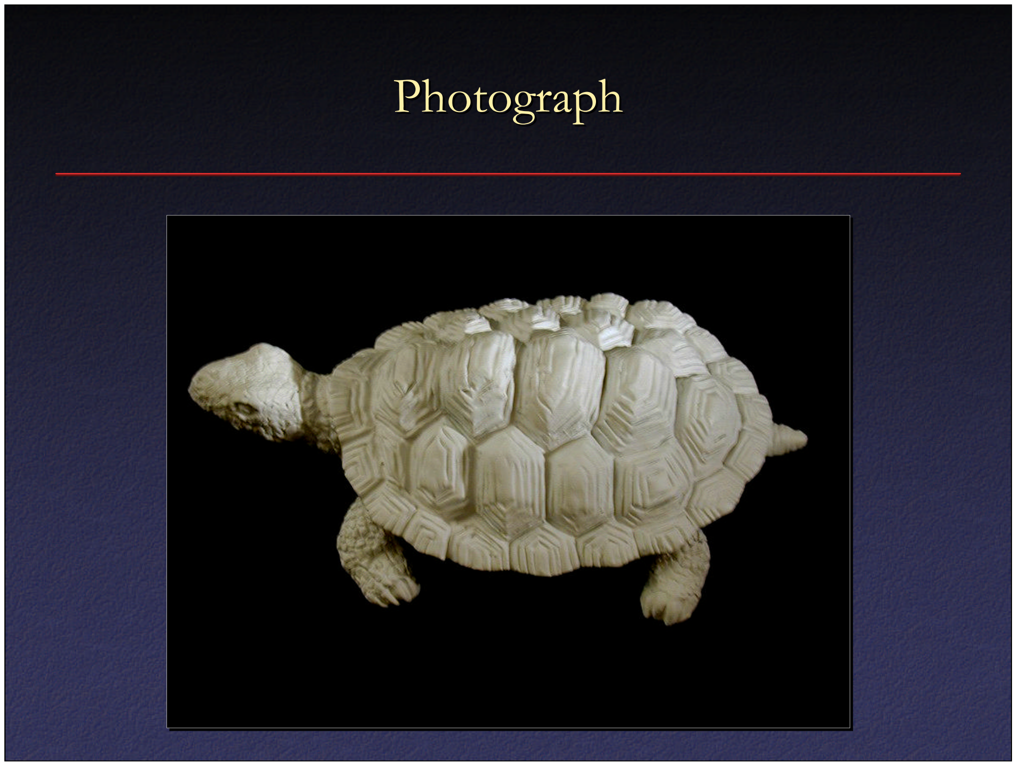
A good application of this type of fast ICP algorithm is a real-time scanner that returns range images at many frames per second. The ICP can then align the images in real time and, because you typically start very close to the correct alignment already, there is little danger of incorrect convergence. [Rusinkiewicz et al. 2002]



Scanner Layout



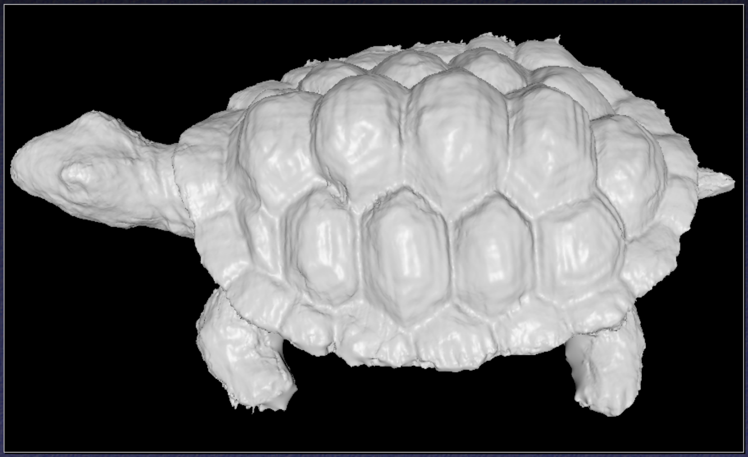
Here's what the scanner looks like: it uses a camera and projector in a triangulated-structured-light system, and provides real-time feedback to the user. The thing displayed on the monitor is NOT an image, but rather a rendering of the model acquired so far. The user can see and immediately fill holes in the model.



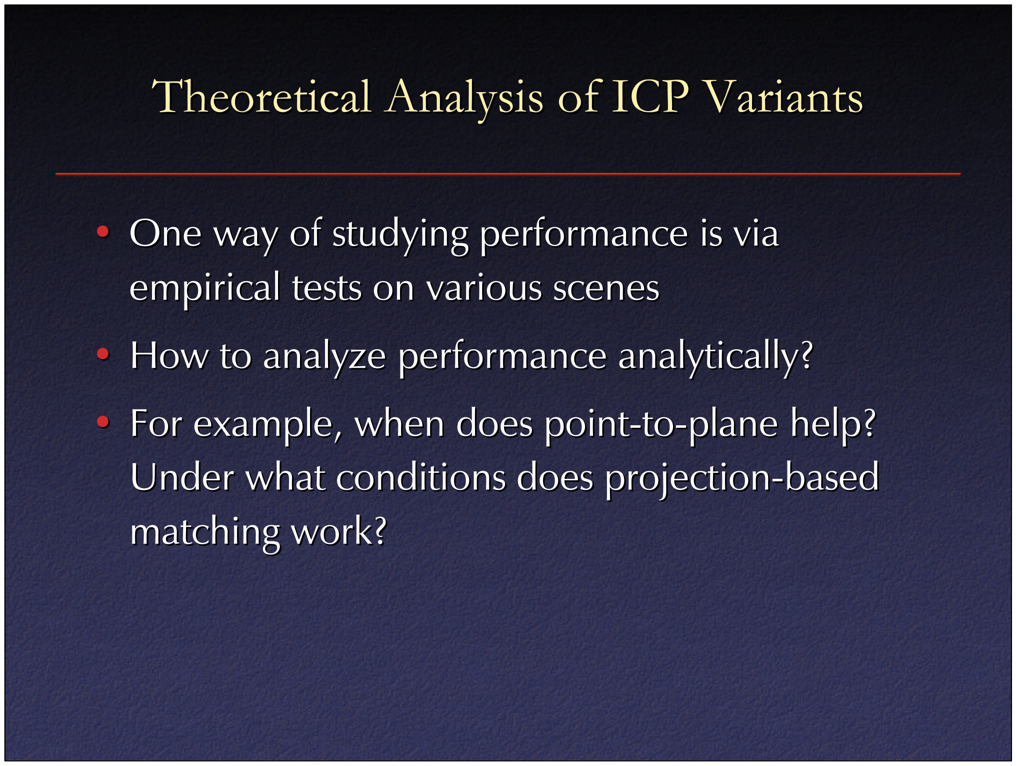
Photograph

Here's an object…

Real-Time Result



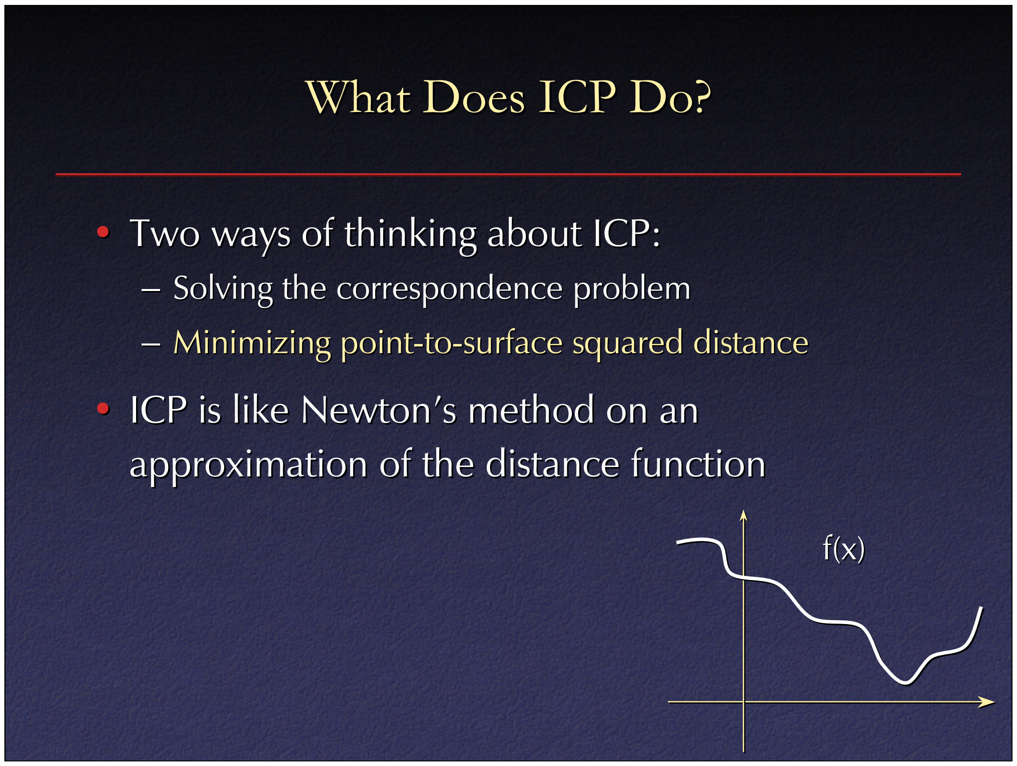
and here's the result of this real-time 3D model acquisition system.



Theoretical Analysis of ICP Variants

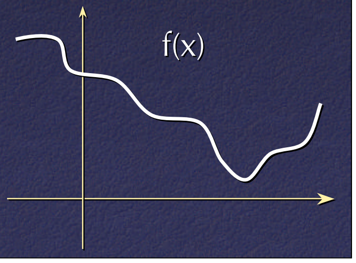
- One way of studying performance is via empirical tests on various scenes
- How to analyze performance analytically?
- For example, when does point-to-plane help? Under what conditions does projection-based matching work?

Now that we've looked at a few different ICP variants, it's time to step back for a bit and think about what ICP is doing.

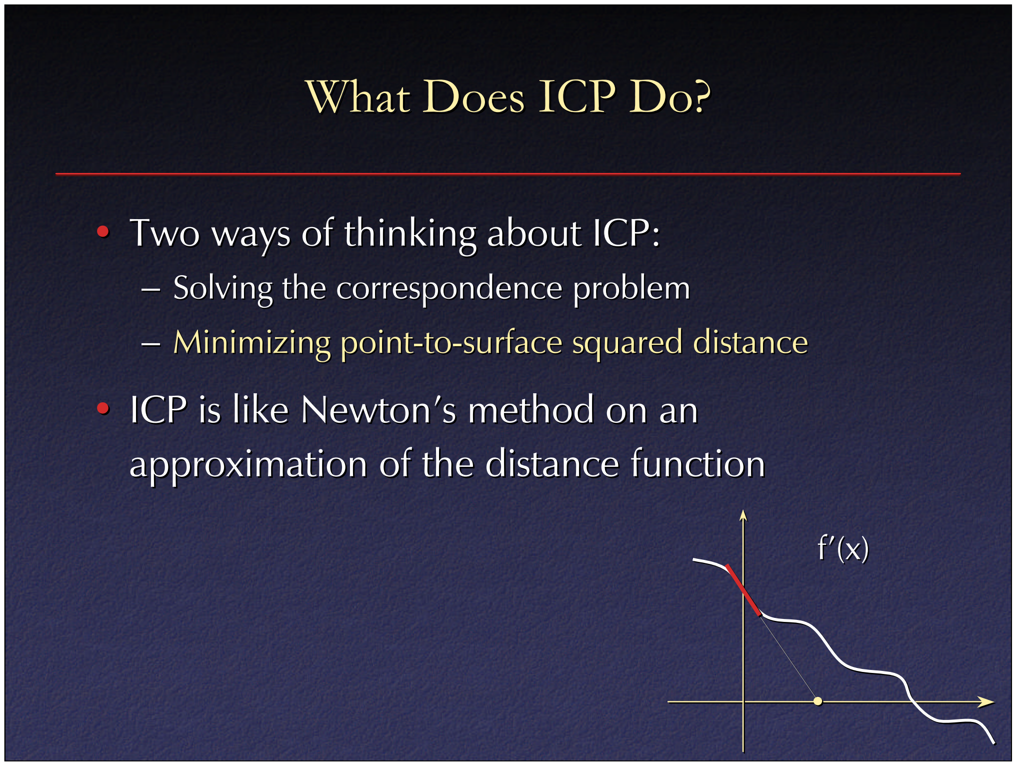


What Does ICP Do?

- Two ways of thinking about ICP:
  - Solving the correspondence problem
  - Minimizing point-to-surface squared distance
- ICP is like Newton's method on an approximation of the distance function

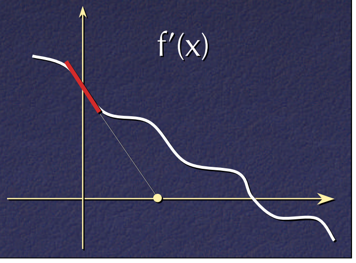


One way of thinking about it is that it's solving the correspondence problem. A different way, though, is that it's minimizing a function: the distance from one scan to the other. Equivalently, it's trying to minimize the (squared) *distance field* of one scan, at locations on the surface of the other. It does so using essentially a Newton-like iteration, on *some approximation* of the distance field.

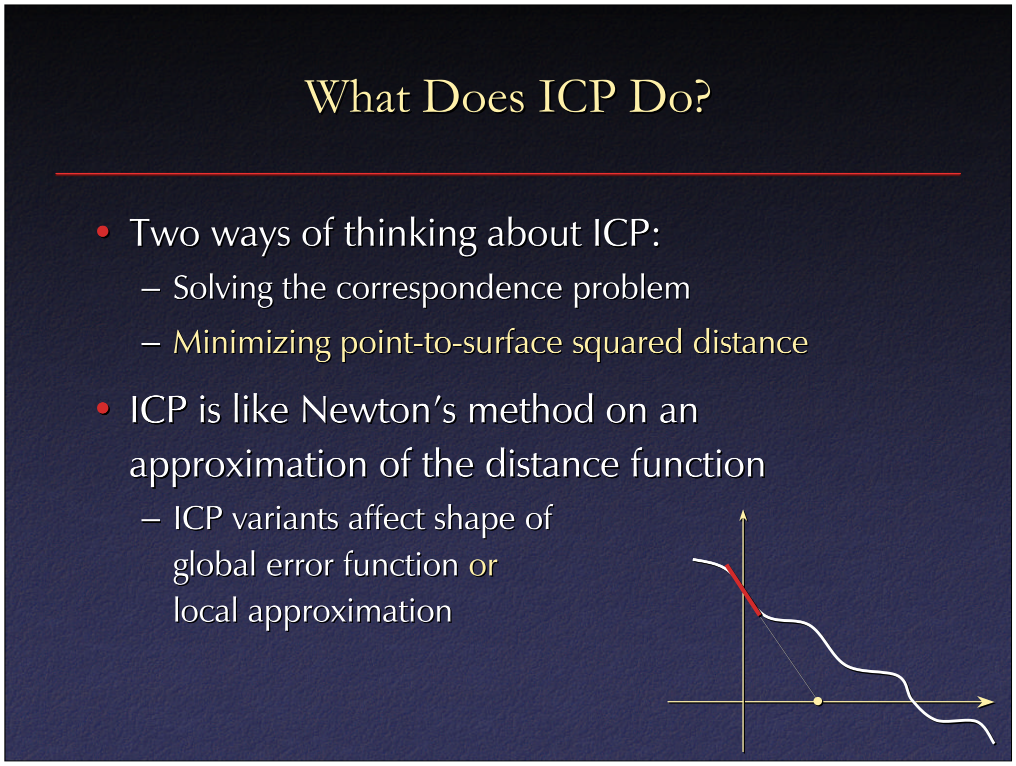


What Does ICP Do?

- Two ways of thinking about ICP:
  - Solving the correspondence problem
  - Minimizing point-to-surface squared distance
- ICP is like Newton's method on an approximation of the distance function

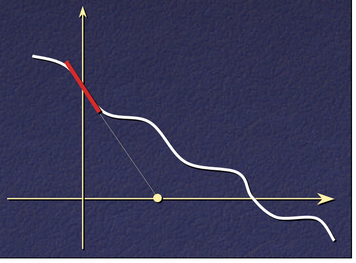


The iteration is basically finding the derivative of the function, extrapolating to zero, etc.

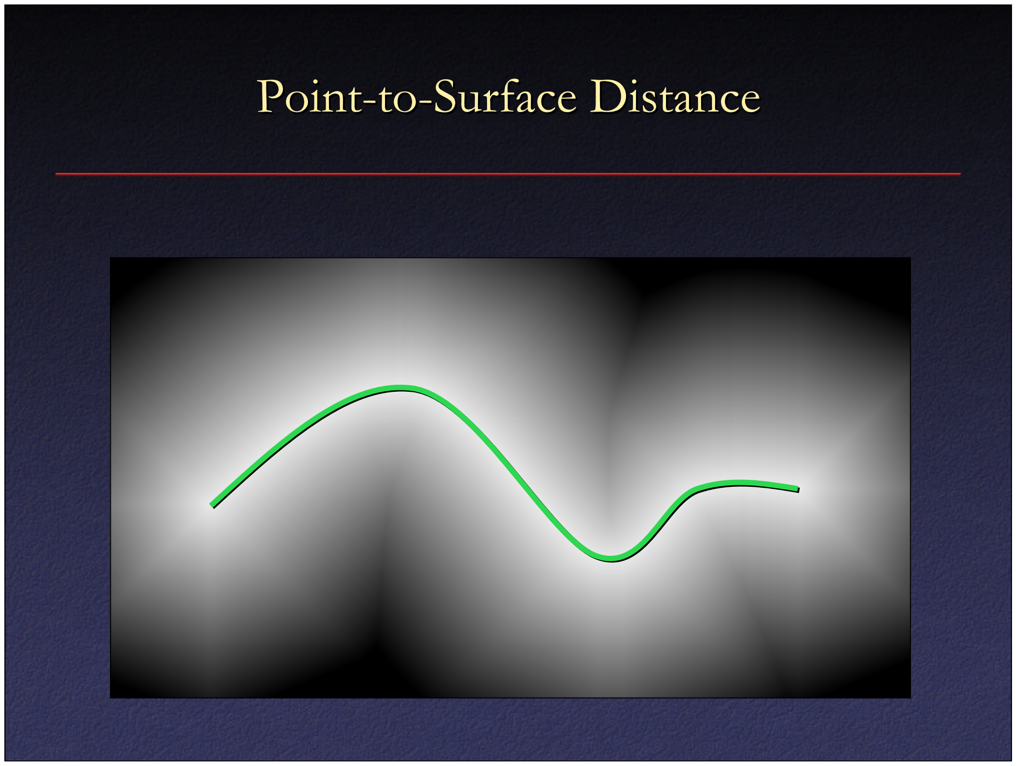


What Does ICP Do?

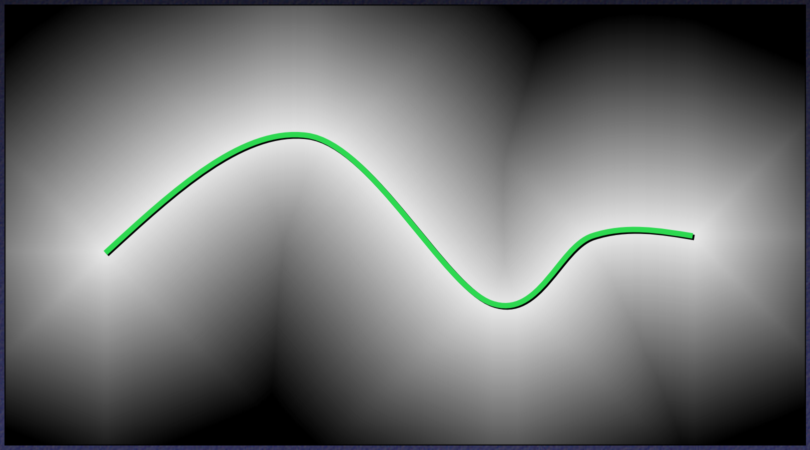
- Two ways of thinking about ICP:
  - Solving the correspondence problem
  - Minimizing point-to-surface squared distance
- ICP is like Newton's method on an approximation of the distance function
  - ICP variants affect shape of global error function or local approximation



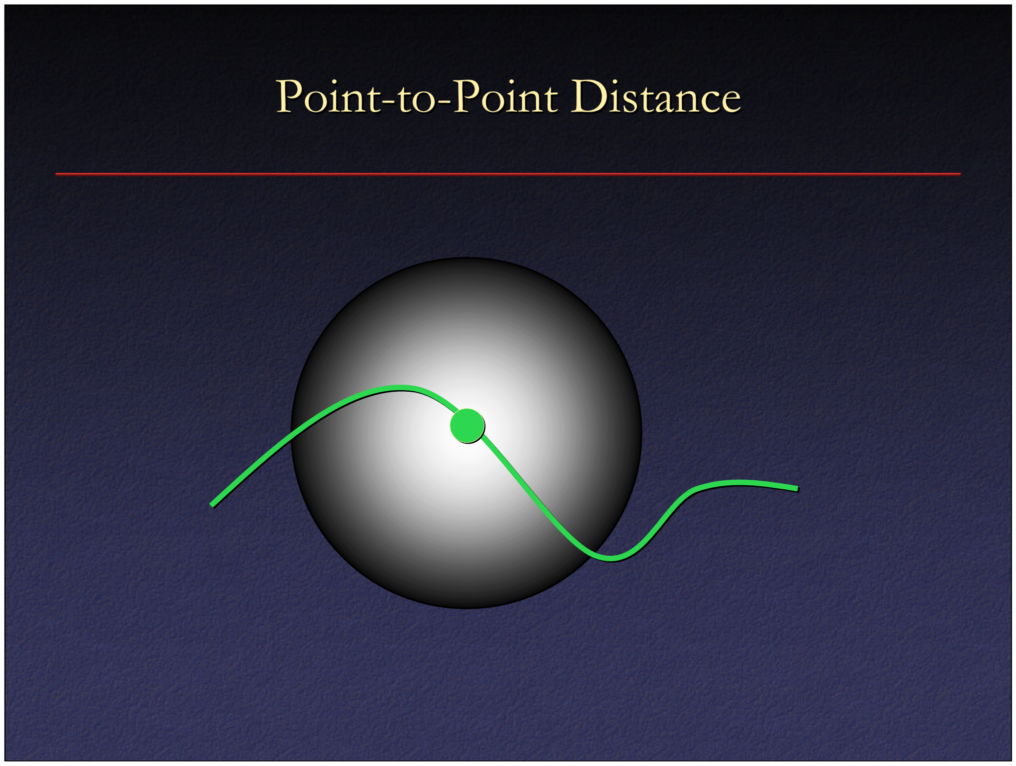
Now, the different ICP variants can do two things. Some of them affect the shape of the function that's being minimized, while others affect how the distance field is locally approximated during the iterations.



Point-to-Surface Distance

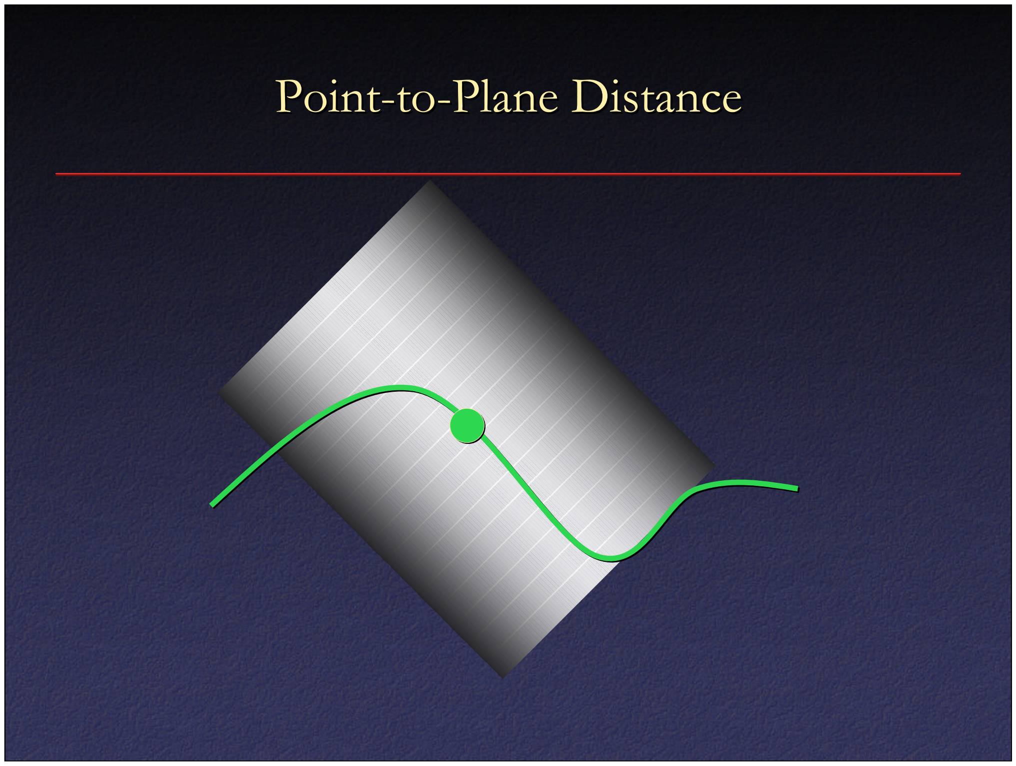


Here's a visualization of the actual function we'd like to use: this is the distance from any point in space to a scan. Registration means minimizing the value of this function as evaluated at points on the other scan.



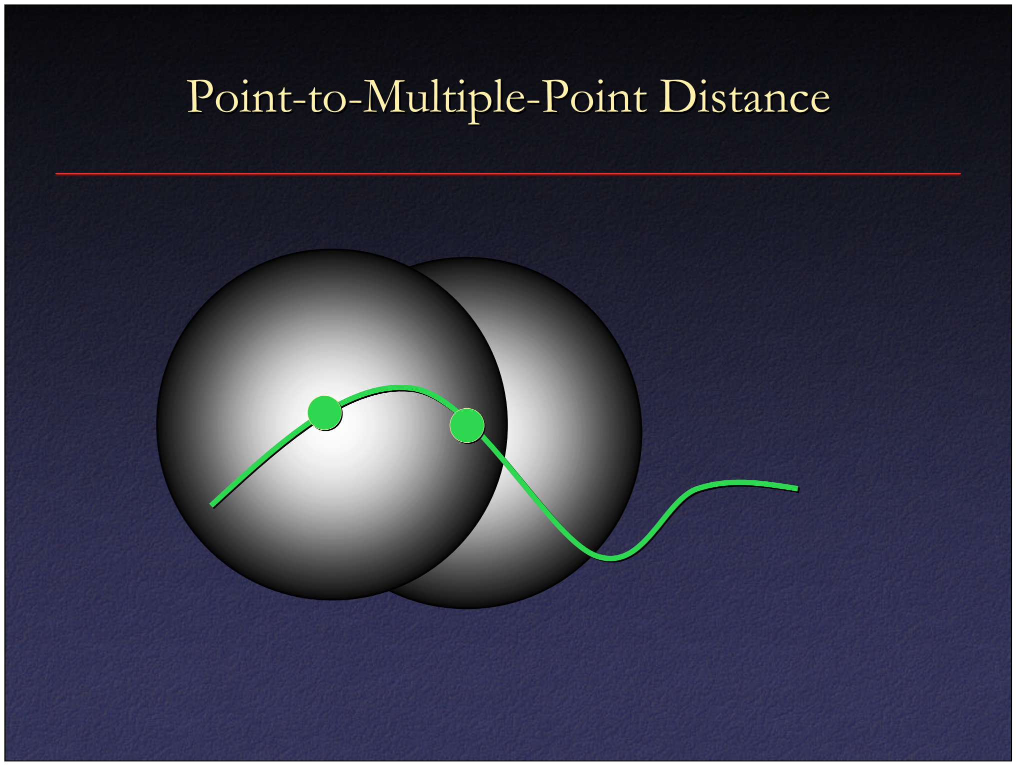
Point-to-Point Distance

Point-to-point (classic) ICP approximates the value of this function by the value of a spherically-symmetric function centered at the point closest to a point on the other scan. How the scan is “pulled” depends on the first and second derivatives of this function.



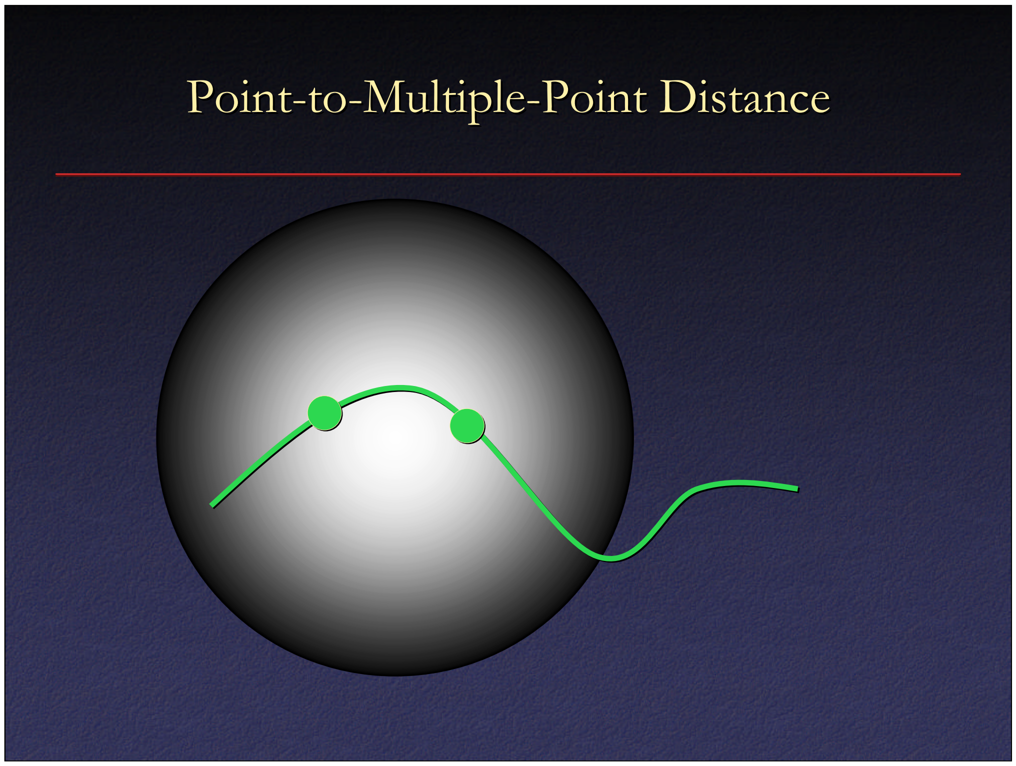
Point-to-Plane Distance

Point-to-plane distance (Chen-Medioni ICP) approximates the function differently. Using this function tends to provide a locally-better approximation to the true distance field than the point-to-point one.



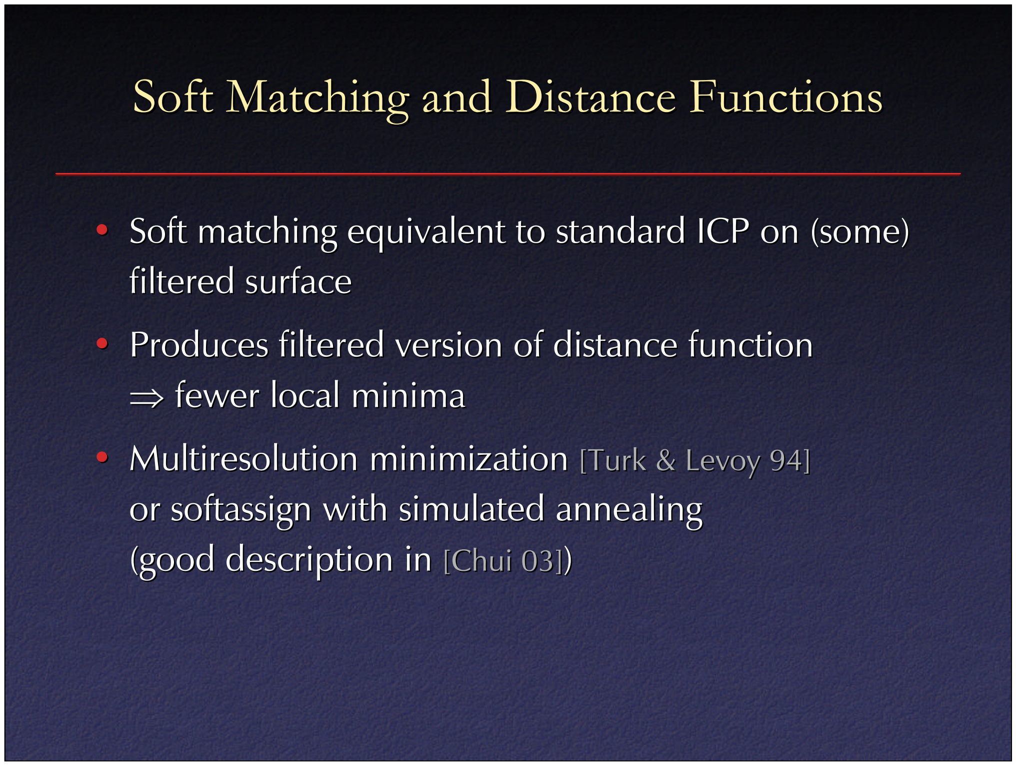
Point-to-Multiple-Point Distance

One thing that some people have tried is an ICP variant that minimizes the distance from one point to several nearby points. This results in a function that’s just the sum of these two spherical functions in space…



Point-to-Multiple-Point Distance

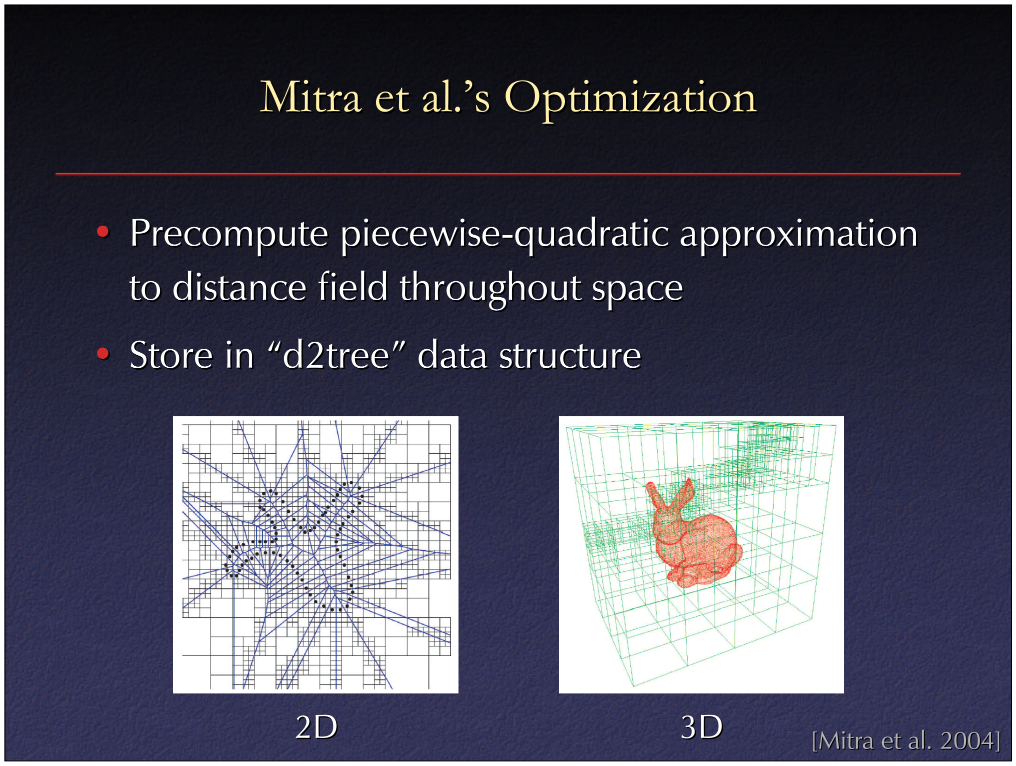
… which works out to be again spherically symmetric.



Soft Matching and Distance Functions

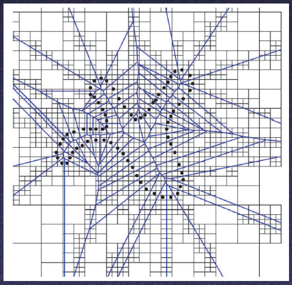
- Soft matching equivalent to standard ICP on (some) filtered surface
- Produces filtered version of distance function $\Rightarrow$ fewer local minima
- Multiresolution minimization [Turk & Levoy 94] or softassign with simulated annealing (good description in [Chui 03])

This means that this "soft" matching is equivalent to just doing standard point-to-point ICP on a smoothed version of the surface. This is good, since it tends to help in avoiding local minima, but it often does not converge as fast as just doing point-to-plane.

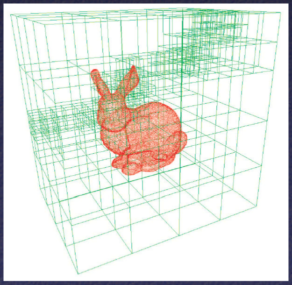


Mitra et al.'s Optimization

- Precompute piecewise-quadratic approximation to distance field throughout space
- Store in "d2tree" data structure



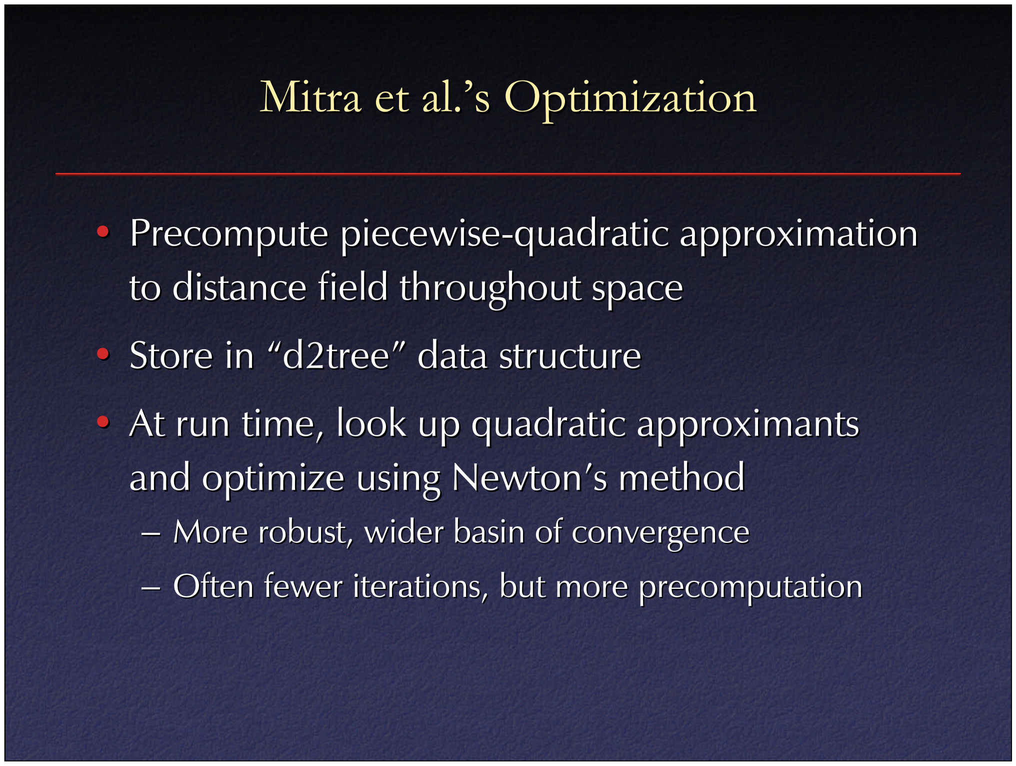
2D



3D

[Mitra et al. 2004]

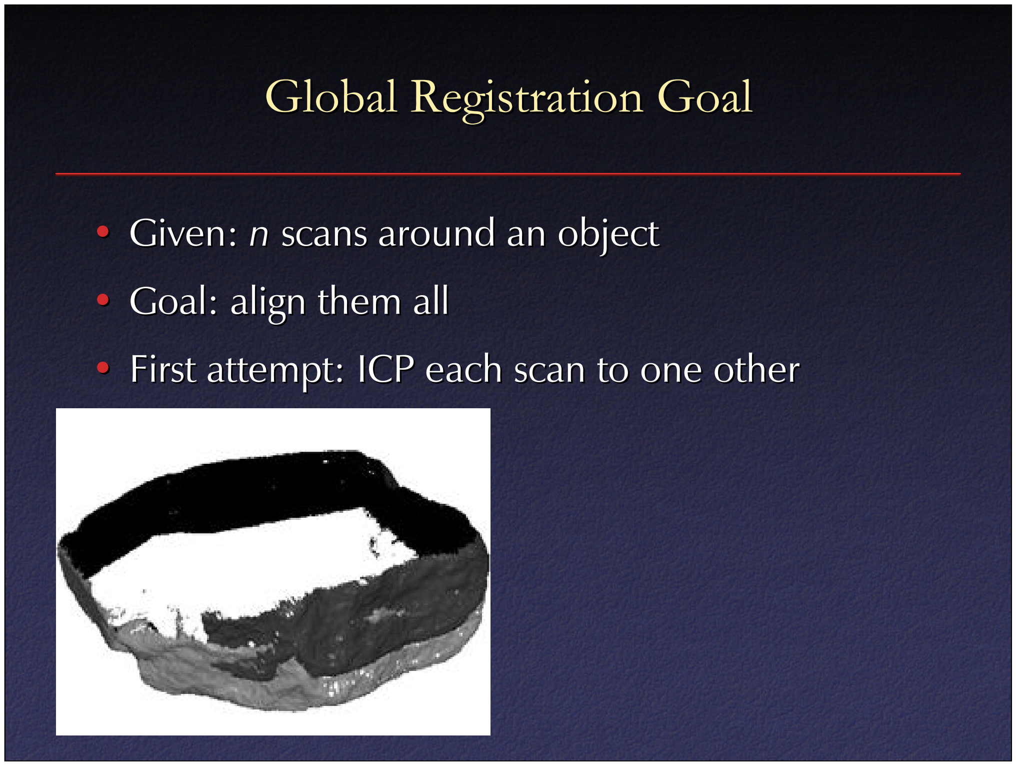
Mitra et al. propose an algorithm that does something different: they precompute and directly store the distance field throughout space. They use a "d2tree" data structure, which is basically a decomposition of space (like a k-d tree) where each region stores the coefficients of a piecewise-quadric approximation to the distance field.



Mitra et al.'s Optimization

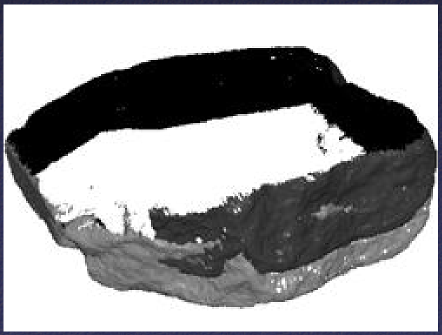
- Precompute piecewise-quadratic approximation to distance field throughout space
- Store in "d2tree" data structure
- At run time, look up quadratic approximants and optimize using Newton's method
  - More robust, wider basin of convergence
  - Often fewer iterations, but more precomputation

At run time, the appropriate derivatives of the distance field are looked up in the data structure, and the optimization directly uses Newton's method. They show that this makes the algorithm much more robust and increases the "basin of convergence" (of course, at the cost of the precomputation necessary to create the d2tree data structure.

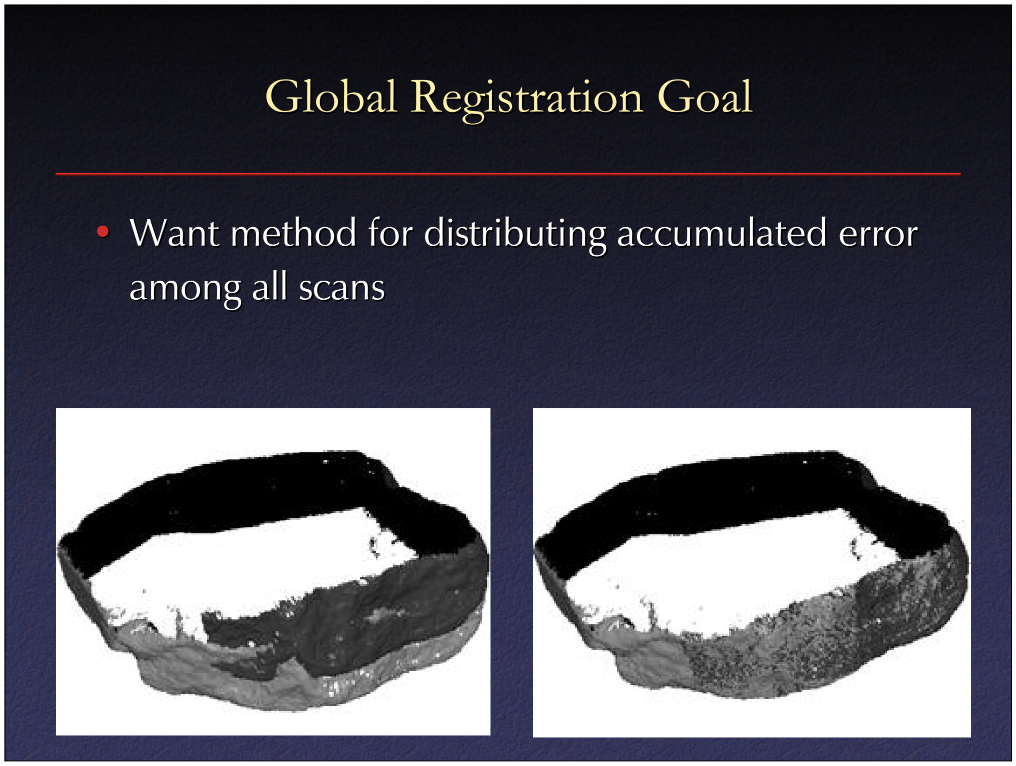


Global Registration Goal

- Given: *n* scans around an object
- Goal: align them all
- First attempt: ICP each scan to one other

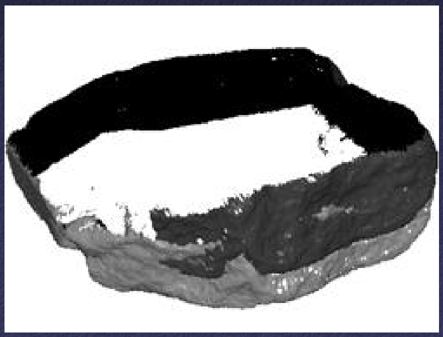


Now let's turn to global registration. Of course, the problem we're trying to solve is aligning more than just 2 scans. As hinted earlier, just aligning scans pairwise can lead to disappointing results.

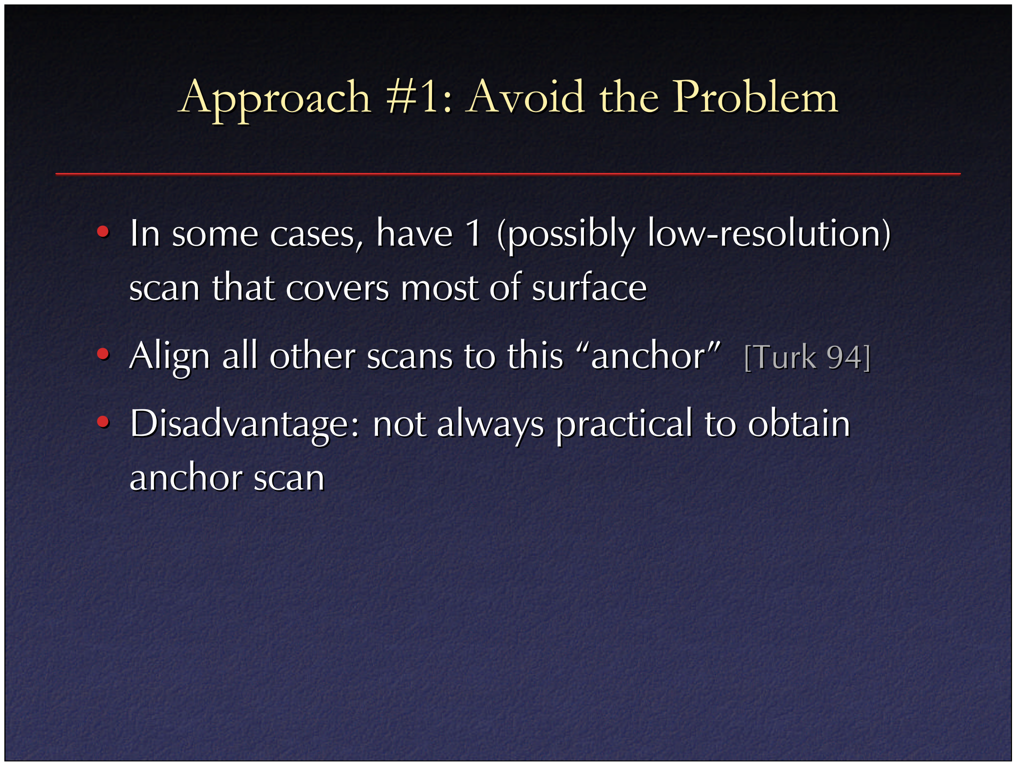


Global Registration Goal

- Want method for distributing accumulated error among all scans



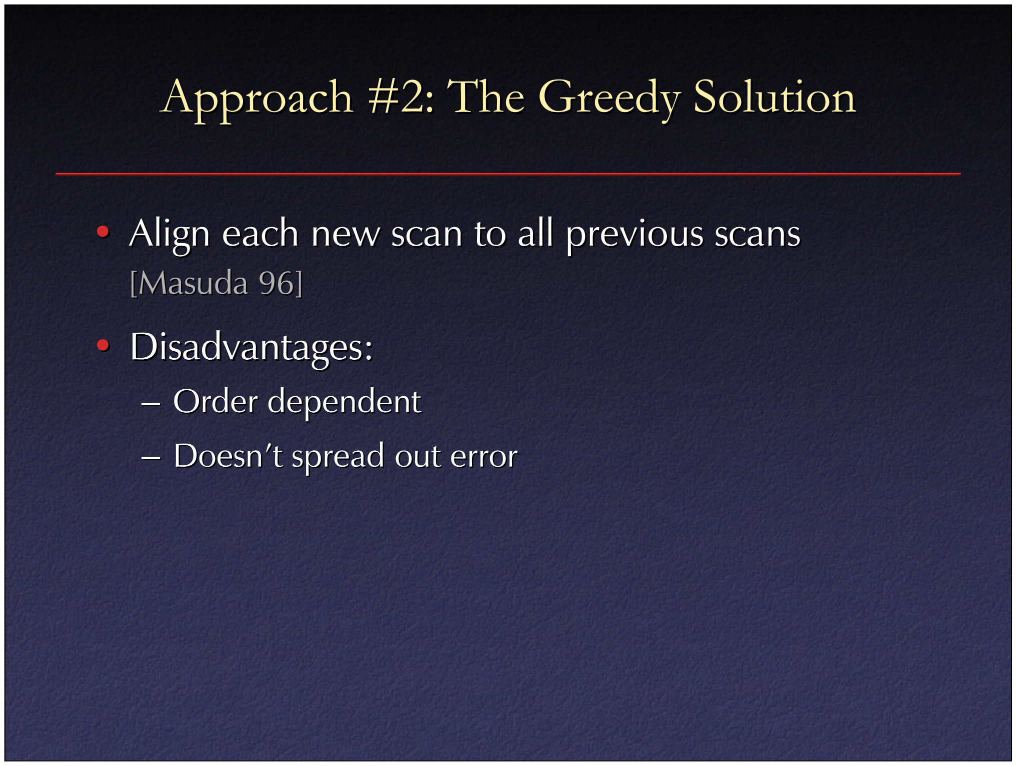
We really want to optimize the pairwise distances of *all* pairs of overlapping scans.



Approach #1: Avoid the Problem

- In some cases, have 1 (possibly low-resolution) scan that covers most of surface
- Align all other scans to this "anchor" [Turk 94]
- Disadvantage: not always practical to obtain anchor scan

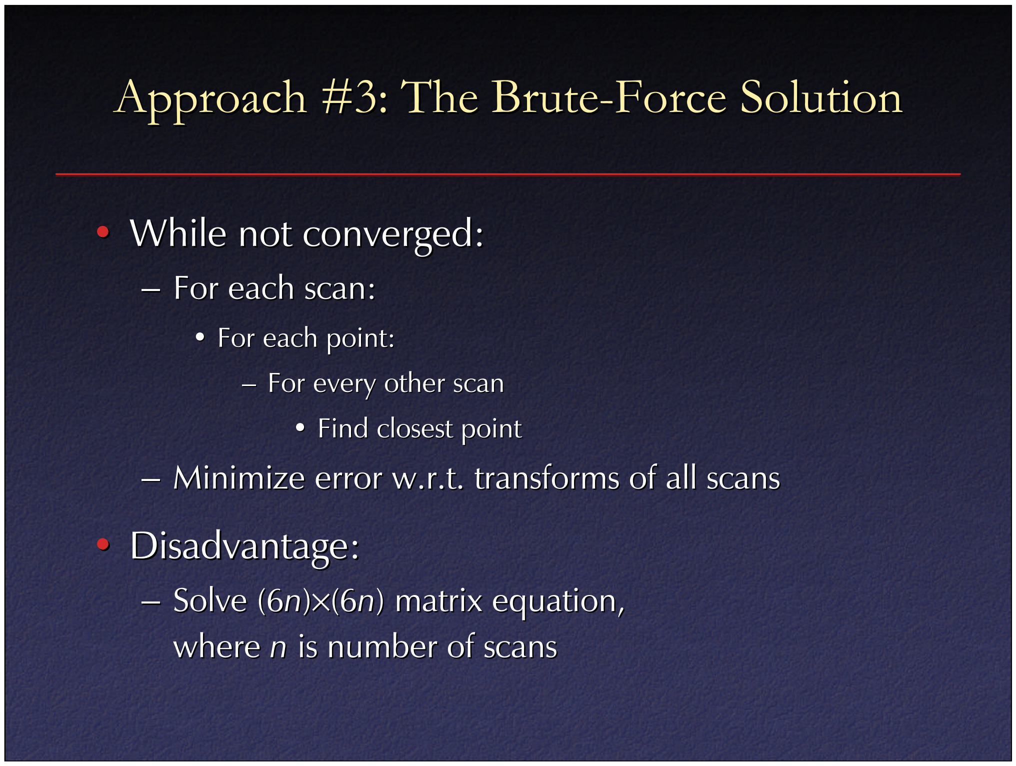
So, how do we do this? One approach that's sometimes used is to take a special scan that covers a large part of the surface. Sometimes this is a "cylindrical scan" that goes all around the object.



Approach #2: The Greedy Solution

- Align each new scan to all previous scans [Masuda 96]
- Disadvantages:
  - Order dependent
  - Doesn't spread out error

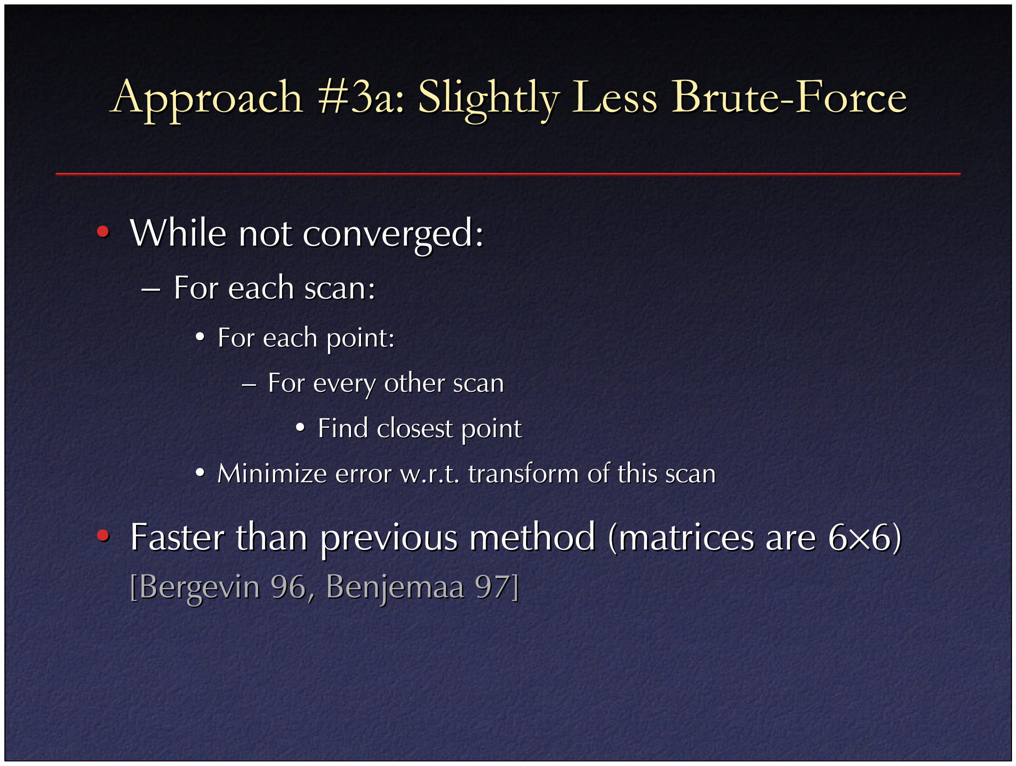
If we can't do that, then at least we can take each new scan as it comes in, and ICP it to the *union* of all previous scans. This sometimes avoids catastrophic accumulation of error, but really isn't guaranteed to do anything.



Approach #3: The Brute-Force Solution

- While not converged:
  - For each scan:
    - For each point:
      - For every other scan
        - Find closest point
  - Minimize error w.r.t. transforms of all scans
- Disadvantage:
  - Solve $(6n)\times(6n)$ matrix equation, where n is number of scans

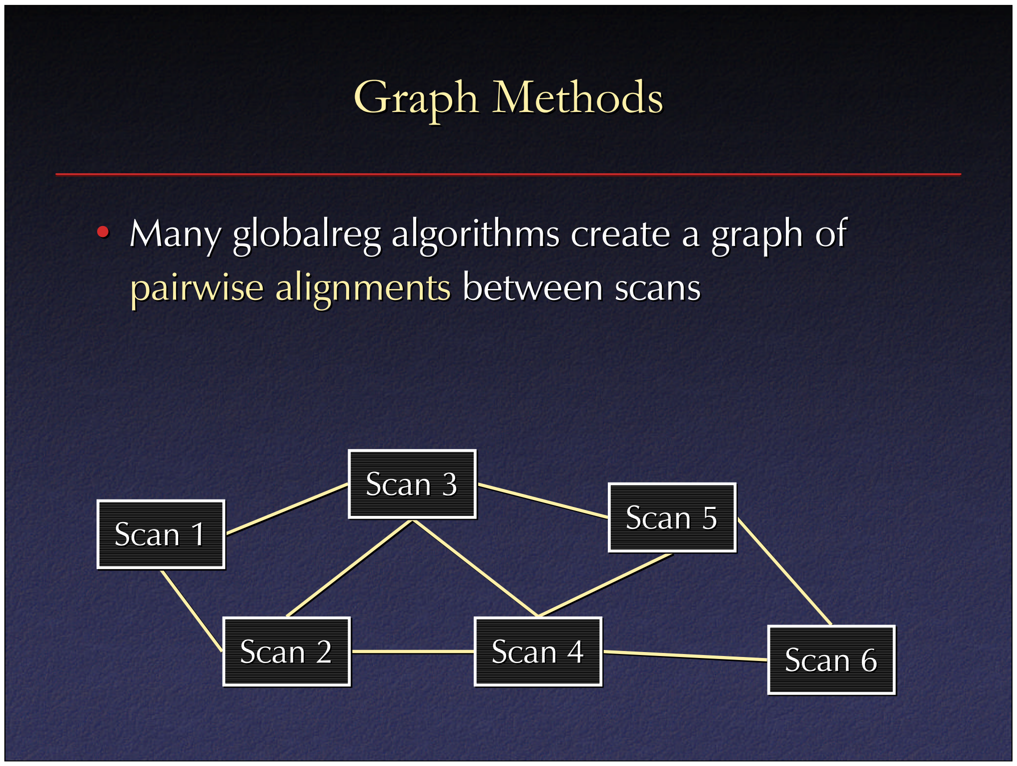
The next step is the brute-force solution: bring all scans into the ICP iteration loop. This works, but requires solving some *awfully* large systems of equations.



Approach #3a: Slightly Less Brute-Force

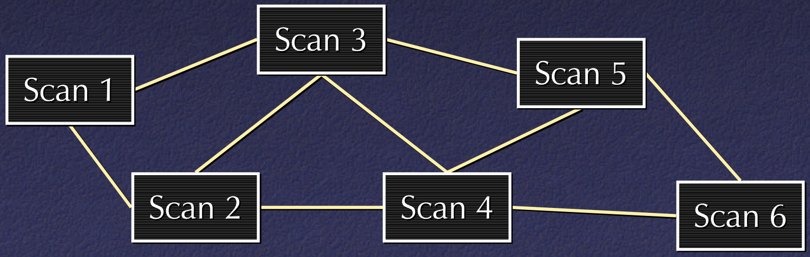
- While not converged:
  - For each scan:
    - For each point:
      - For every other scan
        - Find closest point
    - Minimize error w.r.t. transform of this scan
- Faster than previous method (matrices are 6×6)
  [Bergevin 96, Benjemaa 97]

A slightly less brute-force idea is to, essentially, repeatedly align each scan to the union of all other scans. Given enough passes over the whole collection of scans, this will eventually converge.

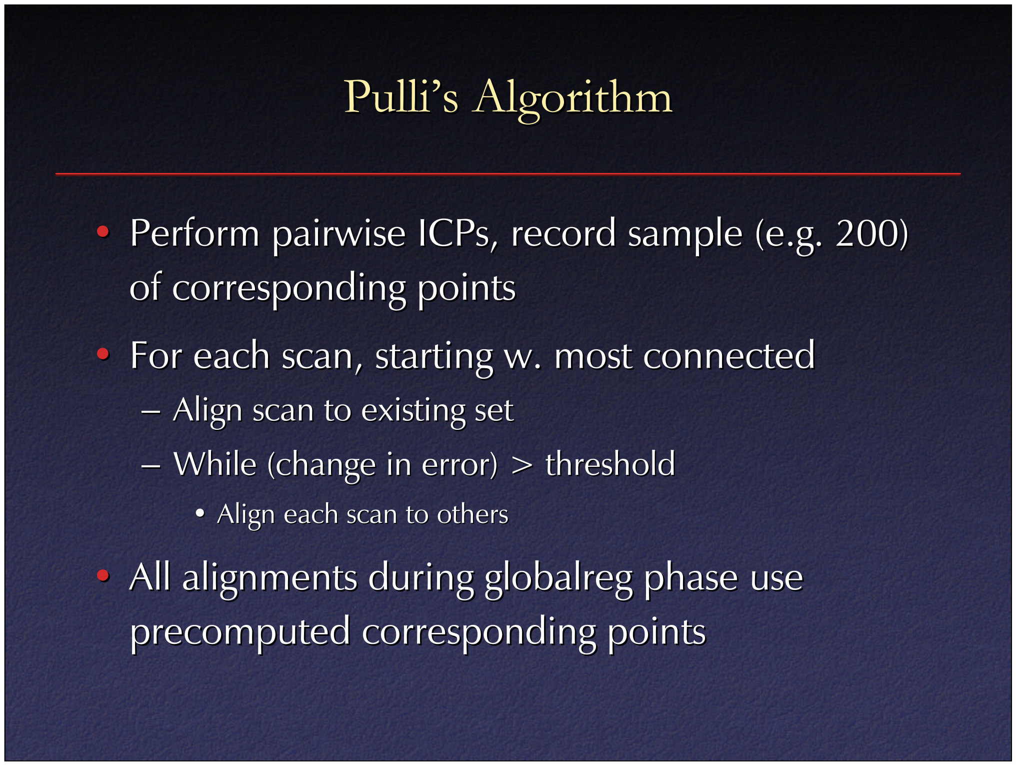


Graph Methods

- Many globalreg algorithms create a graph of pairwise alignments between scans



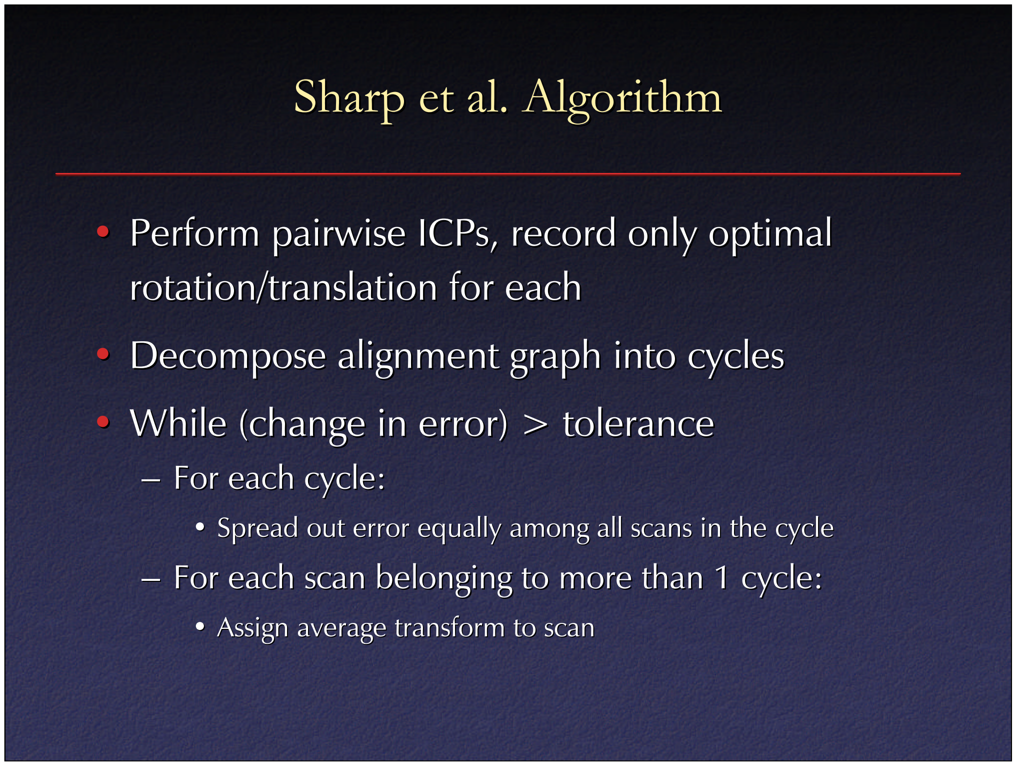
A simple way of speeding up the previous idea is to precompute pairwise alignments between all overlapping scans. The globalreg step then just has to solve for a set of transformations that are as consistent as possible with all the pairwise ICPs. The nice thing about this is that the globalreg step does not need all the scans themselves in memory, just some data computed during the pairwise registrations. This makes it much more practical for models with lots of scans.



Pulli's Algorithm

- Perform pairwise ICPs, record sample (e.g. 200) of corresponding points
- For each scan, starting w. most connected
  - Align scan to existing set
  - While (change in error) > threshold
    - Align each scan to others
- All alignments during globalreg phase use precomputed corresponding points

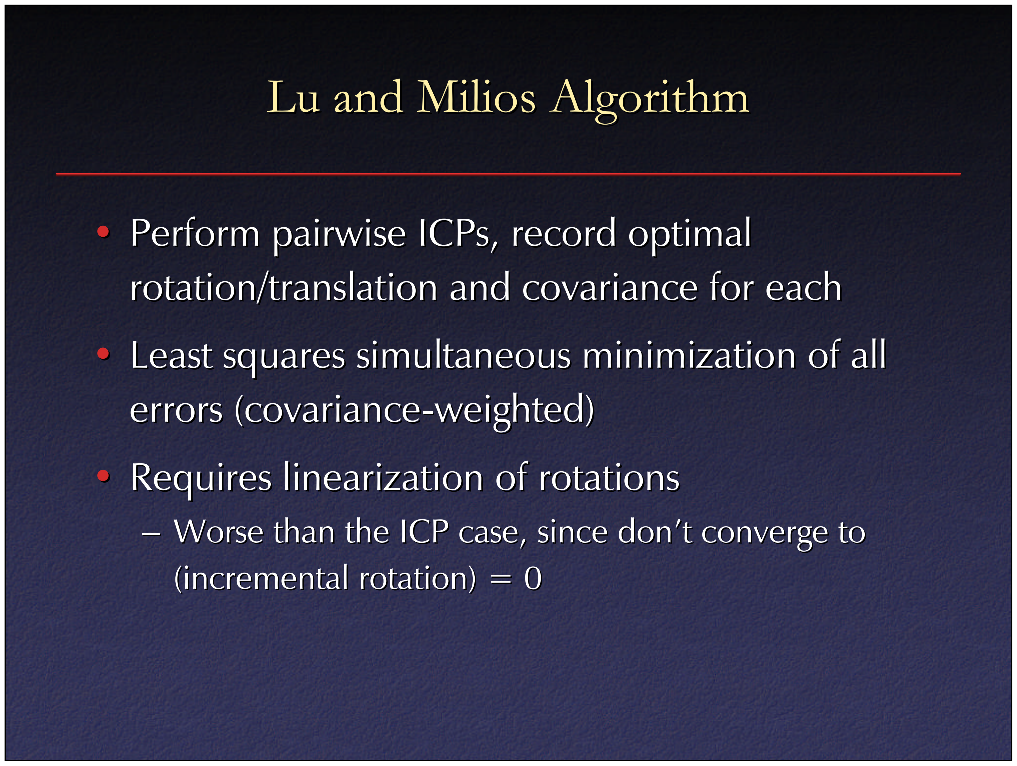
Pulli's algorithm is a combination of the ideas on the two previous slides: the result of the pairwise ICPs is a small set of point correspondences between the scans, which are the only thing that is used during the global registration.



Sharp et al. Algorithm

- Perform pairwise ICPs, record only optimal rotation/translation for each
- Decompose alignment graph into cycles
- While (change in error) > tolerance
  - For each cycle:
    - Spread out error equally among all scans in the cycle
  - For each scan belonging to more than 1 cycle:
    - Assign average transform to scan

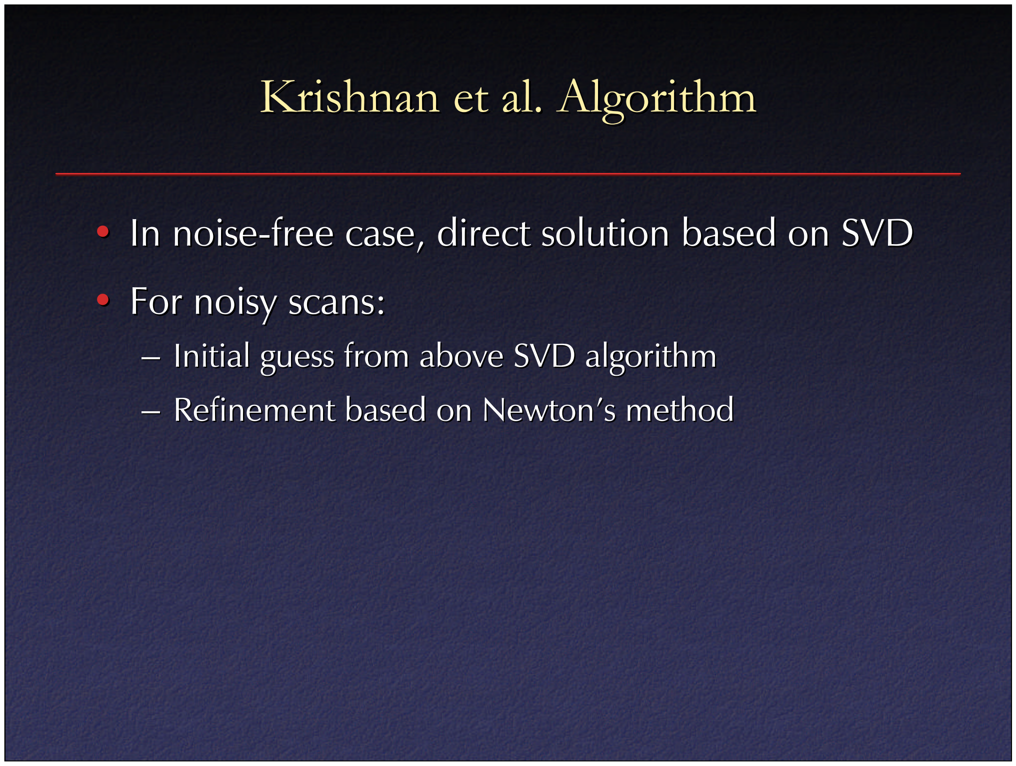
Sharp had a different variant on the same theme, which focused specifically on “spreading out” the mutual inconsistency between pairwise alignments. One potential weakness of this is that it doesn’t take into account the fact that different pairs of scans may have their alignments determined with different degrees of confidence.



Lu and Milios Algorithm

- Perform pairwise ICPs, record optimal rotation/translation and covariance for each
- Least squares simultaneous minimization of all errors (covariance-weighted)
- Requires linearization of rotations
  – Worse than the ICP case, since don't converge to (incremental rotation) = 0

Lu and Milios proposed extending the “linearization” approach used in point-to-plane alignment to solve for all the transformations.



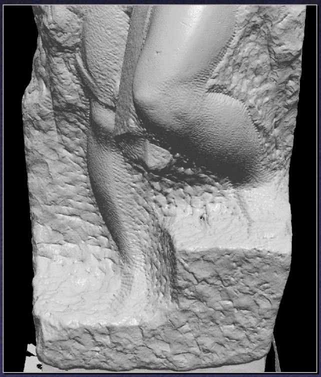
Krishnan et al. Algorithm

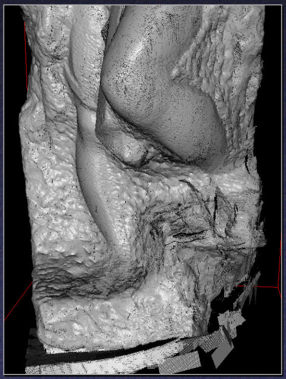
- In noise-free case, direct solution based on SVD
- For noisy scans:
  - Initial guess from above SVD algorithm
  - Refinement based on Newton's method

Finally, Krishnan et al. proposed an SVD-based method that can solve for the transformations directly in the absence of noise. Otherwise, the SVD provides an initial estimate, and the global registration proceeds via a Newton iteration.

Bad ICP in Globalreg

One bad ICP can throw off the entire model!


Correct Globalreg


Globalreg Including Bad ICP

One difficulty with all the global registration algorithms that precompute pairwise alignments is that they can be sensitive to outliers. If any of the ICPs converge to the wrong alignment, they can pull large chunks of the model along for the ride. Believe it or not, this is not a contrived example: this actually happened when we were trying to put together the scans of St. Matthew…