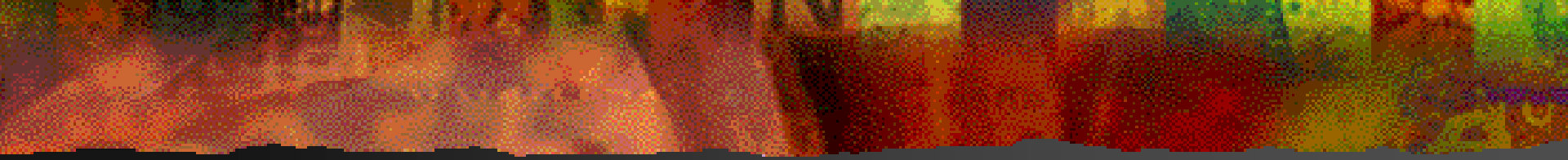


# Inverse Kinematics (part 2)

CSE169: Computer Animation

Instructor: Steve Rotenberg

UCSD, Winter 2020

# Forward Kinematics

- We will use the vector:

$$\mathbf{\Phi} = \begin{bmatrix} \phi_1 & \phi_2 & \dots & \phi_M \end{bmatrix}$$

  to represent the array of M joint DOF values
- We will also use the vector:

$$\mathbf{e} = \begin{bmatrix} e_1 & e_2 & \dots & e_N \end{bmatrix}$$

  to represent an array of N DOFs that describe the *end effector* in world space. For example, if our end effector is a full joint with orientation, **e** would contain 6 DOFs: 3 translations and 3 rotations. If we were only concerned with the end effector position, **e** would just contain the 3 translations.

# Forward & Inverse Kinematics

- The forward kinematic function computes the world space end effector DOFs from the joint DOFs:

$$\mathbf{e} = f\left(\mathbf{\Phi}\right)$$

- The goal of inverse kinematics is to compute the vector of joint DOFs that will cause the end effector to reach some desired goal state

$$\mathbf{\Phi} = f^{-1}\left(\mathbf{e}\right)$$

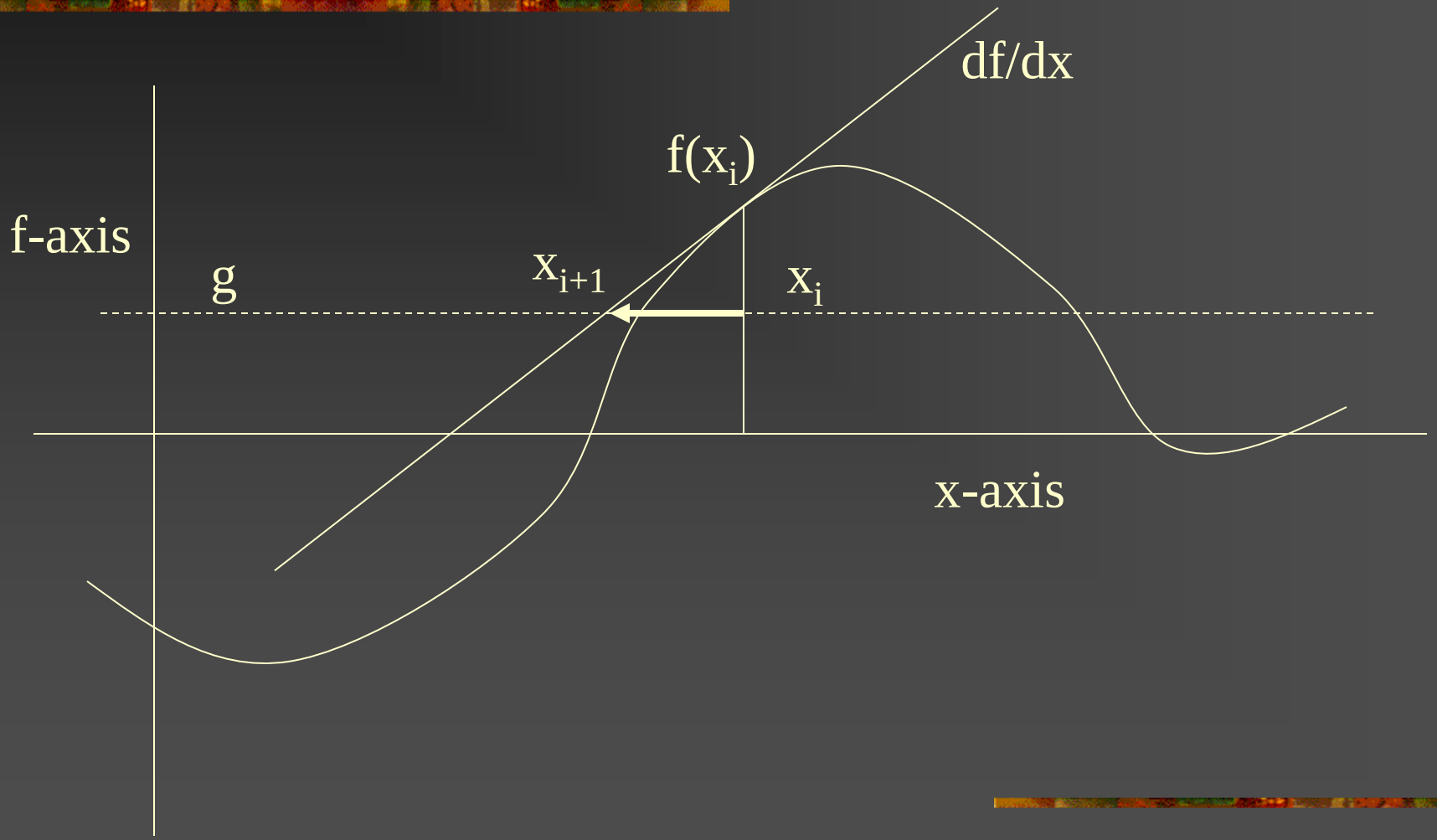
# Gradient Descent

- We want to find the value of x that causes f(x) to equal some goal value g
- We will start at some value $x_0$ and keep taking small steps:

  $$x_{i+1} = x_i + \Delta x$$

  until we find a value $x_N$ that satisfies $f(x_N)=g$
- For each step, we try to choose a value of $\Delta x$ that will bring us closer to our goal
- We can use the derivative to approximate the function nearby, and use this information to move ‘downhill’ towards the goal

# Gradient Descent for f(x)=g



# Gradient Descent Algorithm

$x_0 = \text{initial starting value}$

$f_0 = f(x_0)$ // evaluate $f$ at $x_0$

while $(f_i \neq g)$ {

$s_i = \frac{df}{dx}(x_i)$ // compute slope

$x_{i+1} = x_i + \beta(g - f_i)\frac{1}{s_i}$ // take step along $\Delta x$

$f_{i+1} = f(x_{i+1})$ // evaluate $f$ at new $x_{i+1}$

}

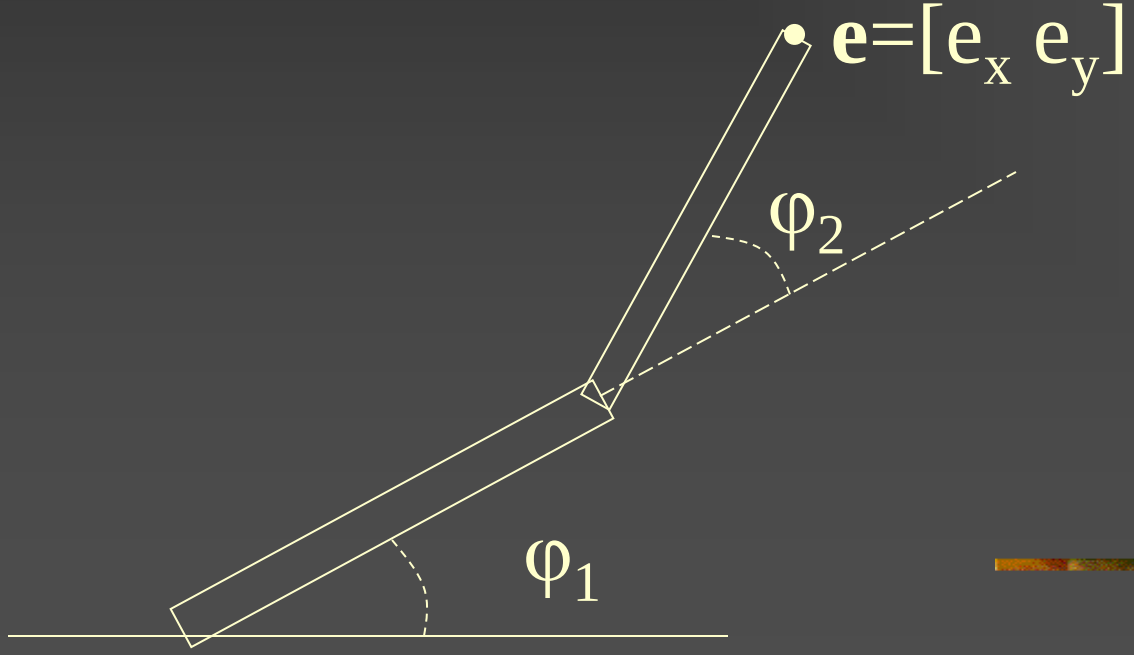
# Jacobian Inverse Kinematics

# Jacobians

$$J(\mathbf{f},\mathbf{x}) = \frac{d\mathbf{f}}{d\mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_M}{\partial x_1} & \cdots & \cdots & \frac{\partial f_M}{\partial x_N} \end{bmatrix}$$

# Jacobians

- Let's say we have a simple 2D robot arm with two 1-DOF rotational joints:



# Jacobians

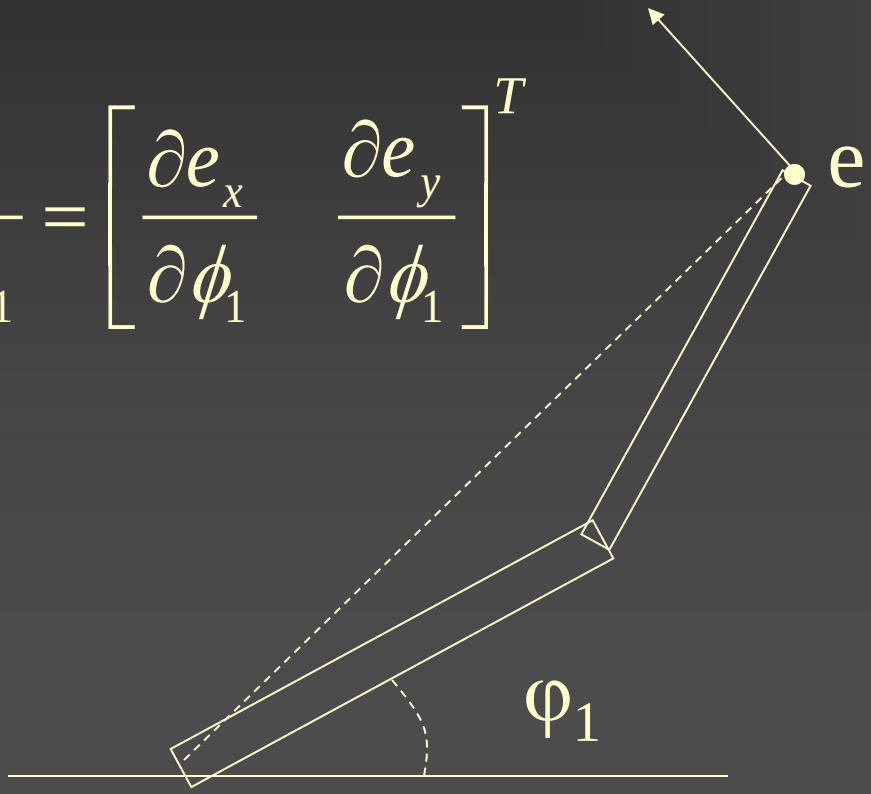
- The Jacobian matrix J(**e**,**Φ**) shows how each component of **e** varies with respect to each joint angle

$$J(\mathbf{e}, \mathbf{\Phi}) = \begin{bmatrix} \frac{\partial e_x}{\partial \phi_1} & \frac{\partial e_x}{\partial \phi_2} \\ \frac{\partial e_y}{\partial \phi_1} & \frac{\partial e_y}{\partial \phi_2} \end{bmatrix}$$

# Jacobians

- Consider what would happen if we increased $\varphi_1$ by a small amount. What would happen to **e** ?

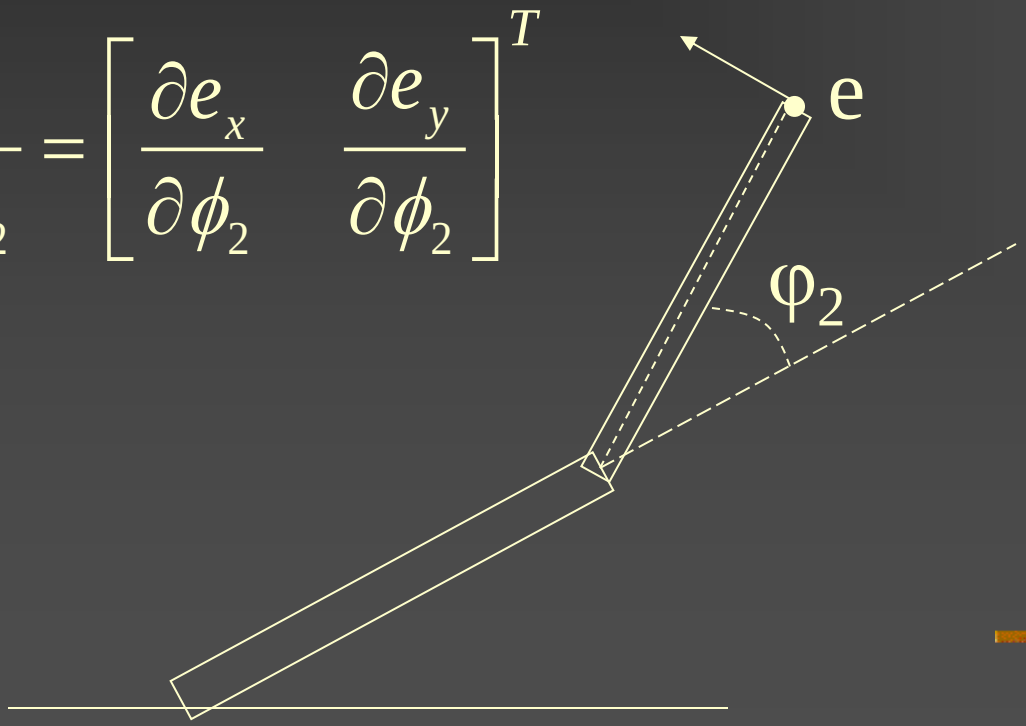
$$\frac{\partial \mathbf{e}}{\partial \phi_1} = \left[ \frac{\partial e_x}{\partial \phi_1} \quad \frac{\partial e_y}{\partial \phi_1} \right]^T$$



# Jacobians

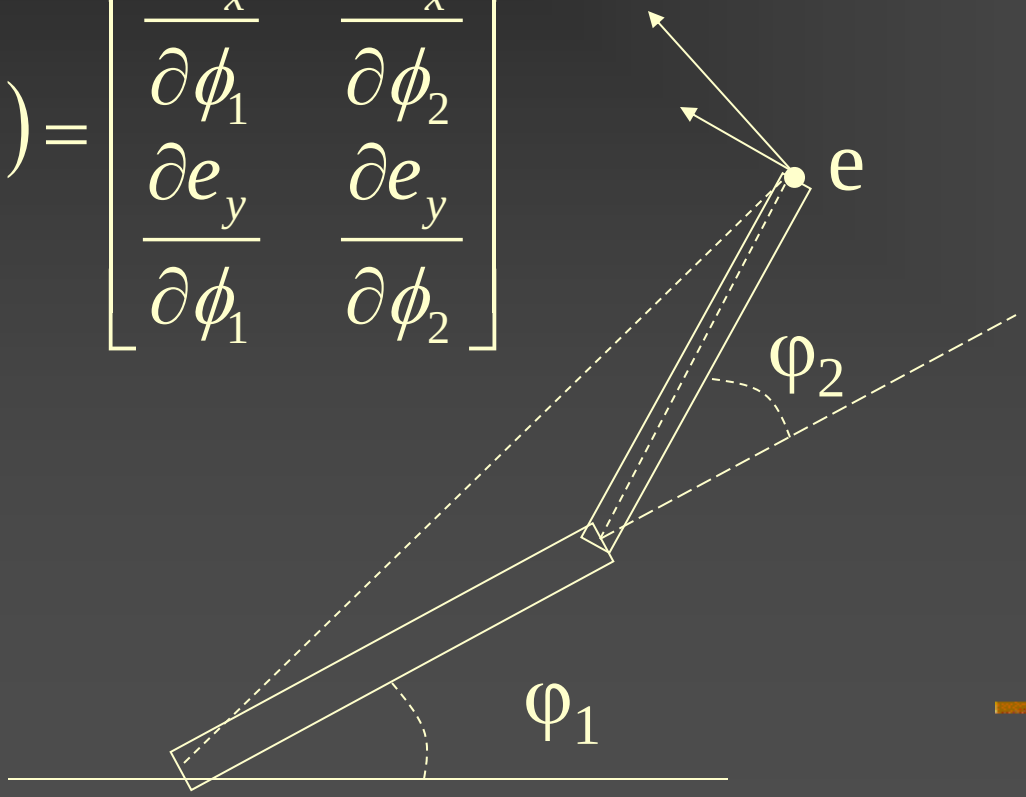
- What if we increased $\varphi_2$ by a small amount?

$$\frac{\partial \mathbf{e}}{\partial \phi_2} = \left[ \frac{\partial e_x}{\partial \phi_2} \quad \frac{\partial e_y}{\partial \phi_2} \right]^T$$



# Jacobian for a 2D Robot Arm

$$J(\mathbf{e}, \mathbf{\Phi}) = \begin{bmatrix} \frac{\partial e_x}{\partial \phi_1} & \frac{\partial e_x}{\partial \phi_2} \\ \frac{\partial e_y}{\partial \phi_1} & \frac{\partial e_y}{\partial \phi_2} \end{bmatrix}$$



# Incremental Change in Pose

- Lets say we have a vector $\Delta\mathbf{\Phi}$ that represents a small change in joint DOF values
- We can approximate what the resulting change in **e** would be:

$$\Delta\mathbf{e} \approx \frac{d\mathbf{e}}{d\mathbf{\Phi}} \cdot \Delta\mathbf{\Phi} = J\left(\mathbf{e}, \mathbf{\Phi}\right) \cdot \Delta\mathbf{\Phi} = \mathbf{J} \cdot \Delta\mathbf{\Phi}$$

# Incremental Change in Effector

- What if we wanted to move the end effector by a small amount $\Delta\mathbf{e}$. What small change $\Delta\mathbf{\Phi}$ will achieve this?

$$\Delta\mathbf{e} \approx \mathbf{J} \cdot \Delta\mathbf{\Phi}$$

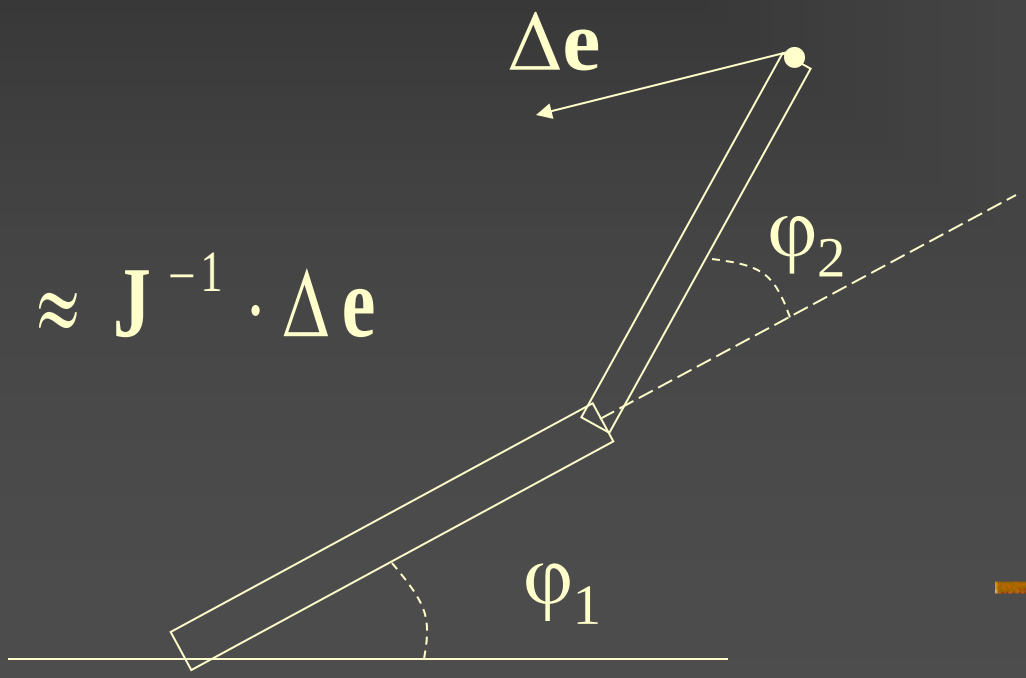
*so* :

$$\Delta\mathbf{\Phi} \approx \mathbf{J}^{-1} \cdot \Delta\mathbf{e}$$

# Incremental Change in **e**

- Given some desired incremental change in end effector configuration $\Delta\mathbf{e}$, we can compute an appropriate incremental change in joint DOFs $\Delta\boldsymbol{\Phi}$

$$\Delta\boldsymbol{\Phi} \approx \mathbf{J}^{-1} \cdot \Delta\mathbf{e}$$



# Incremental Changes

- Remember that forward kinematics is a nonlinear function (as it involves sin's and cos's of the input variables)
- This implies that we can only use the Jacobian as an approximation that is valid near the current configuration
- Therefore, we must repeat the process of computing a Jacobian and then taking a small step towards the goal until we get to where we want to be

# Choosing $\Delta\mathbf{e}$

- We want to choose a value for $\Delta\mathbf{e}$ that will move $\mathbf{e}$ closer to $\mathbf{g}$. A reasonable place to start is with

$$\Delta\mathbf{e} = \mathbf{g} - \mathbf{e}$$

- We would hope then, that the corresponding value of $\Delta\mathbf{\Phi}$ would bring the end effector exactly to the goal
- Unfortunately, the nonlinearity prevents this from happening, but it should get us closer
- Also, for safety, we will take smaller steps:

$$\Delta\mathbf{e} = \beta(\mathbf{g} - \mathbf{e})$$

where $0 < \beta \leq 1$

# Basic Jacobian IK Technique

while (**e** is too far from **g**) {

Compute J(**e**,**Φ**) for the current pose **Φ**

Compute $J^{-1}$ // invert the Jacobian matrix

$\Delta\mathbf{e} = \beta(\mathbf{g} - \mathbf{e})$ // pick approximate step to take

$\Delta\mathbf{\Phi} = J^{-1} \cdot \Delta\mathbf{e}$ // compute change in joint DOFs

$\mathbf{\Phi} = \mathbf{\Phi} + \Delta\mathbf{\Phi}$ // apply change to DOFs

Compute new **e** vector // apply forward
// kinematics to see
// where we ended up

}

# Inverting the Jacobian Matrix

# Inverting the Jacobian

- If the Jacobian is square (number of joint DOFs equals the number of DOFs in the end effector), then we *might* be able to invert the matrix
- Most likely, it won't be square, and even if it is, it's definitely possible that it will be singular and non-invertable
- Even if it is invertable, as the pose vector changes, the properties of the matrix will change and may become singular or near-singular in certain configurations
- The bottom line is that just relying on inverting the matrix is not going to work

# Underconstrained Systems

- If the system has more degrees of freedom in the joints than in the end effector, then it is likely that there will be a continuum of *redundant* solutions (i.e., an infinite number of solutions)
- In this situation, it is said to be underconstrained or redundant
- These should still be solvable, and might not even be too hard to find a solution, but it may be tricky to find a ‘best’ solution

# Overconstrained Systems

- If there are more degrees of freedom in the end effector than in the joints, then the system is said to be overconstrained, and it is likely that there will not be any possible solution
- In these situations, we might still want to get as close as possible
- However, in practice, overconstrained systems are not as common, as they are not a very useful way to build an animal or robot (they might still show up in some special cases though)

# Well-Constrained Systems

- If the number of DOFs in the end effector equals the number of DOFs in the joints, the system could be well constrained and invertable
- In practice, this will require the joints to be arranged in a way so their axes are not redundant
- This property may vary as the pose changes, and even well-constrained systems may have trouble

# Pseudo-Inverse

- If we have a non-square matrix arising from an overconstrained or underconstrained system, we can try using the *pseudoinverse*:

$$\mathbf{J}^* = (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T$$

- This is a method for finding a matrix that effectively inverts a non-square matrix
- Some properties of the pseudoinverse:

$J^*J = I$

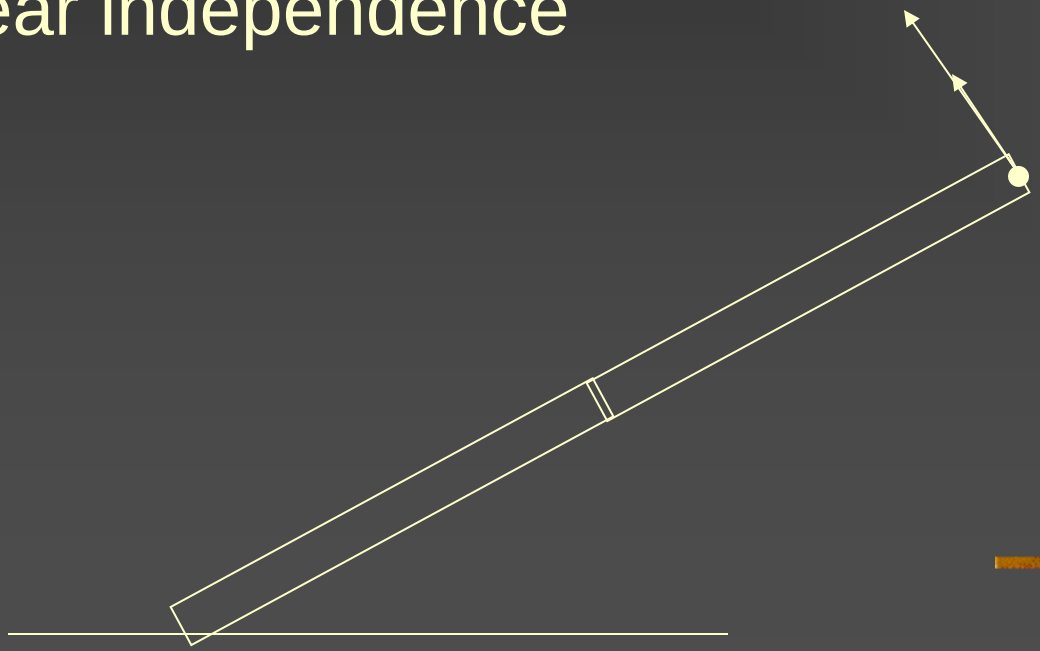
$JJ^* = I$

$(J^*)^* = J$

and for square matrices, $J^* = J^{-1}$

# Degenerate Cases

- Occasionally, we will get into a configuration that suffers from degeneracy
- If the derivative vectors line up, they lose their linear independence



# Single Value Decomposition

- The SVD is an algorithm that decomposes a matrix into a form whose properties can be analyzed easily
- It allows us to identify when the matrix is singular, near singular, or well formed
- It also tells us about what regions of the multidimensional space are not adequately covered in the singular or near singular configurations
- The bottom line is that it is a more sophisticated, but expensive technique that can be useful both for analyzing the matrix and inverting it

# Jacobian Transpose

- Another technique is to simply take the transpose of the Jacobian matrix!
- Surprisingly, this technique actually works pretty well
- It is *much* faster than computing the inverse or pseudo-inverse
- Also, it has the effect of localizing the computations. To compute $\Delta\varphi_i$ for joint i, we compute the column in the Jacobian matrix $\mathbf{J}_i$ as before, and then just use:

$$\Delta\varphi_i = \mathbf{J}_i^T \cdot \Delta e$$

# Jacobian Transpose

- With the Jacobian transpose (JT) method, we can just loop through each DOF and compute the change to that DOF directly
- With the inverse (JI) or pseudo-inverse (JP) methods, we must first loop through the DOFs, compute and store the Jacobian, invert (or pseudo-invert) it, then compute the change in DOFs, and then apply the change
- The JT method is far friendlier on memory access & caching, as well as computations
- However, if one prefers quality over performance, the JP method might be better…

# Iterating to the Solution

# Iteration

- Whether we use the JI, JP, or JT method, we must address the issue of iteration towards the solution
- We should consider how to choose an appropriate step size β and how to decide when the iteration should stop

# When to Stop

- There are three main stopping conditions we should account for
  - Finding a successful solution (or close enough)
  - Getting stuck in a condition where we can't improve (local minimum)
  - Taking too long (for interactive systems)
- All three of these are fairly easy to identify by monitoring the progress of **Φ**
- These rules are just coded into the while() statement for the controlling loop

# Finding a Successful Solution

- We really just want to get close enough within some tolerance
- If we're not in a big hurry, we can just iterate until we get within some floating point error range
- Alternately, we could choose to stop when we get within some tolerance measurable in pixels
- For example, we could position an end effector to 0.1 pixel accuracy
- This gives us a scheme that should look good and automatically adapt to spend more time when we are looking at the end effector up close (level-of-detail)

# Local Minima

- If we get stuck in a local minimum, we have several options
  - Don’t worry about it and just accept it as the best we can do
  - Switch to a different algorithm (CCD…)
  - Randomize the pose vector slightly (or a lot) and try again
  - Send an error to whatever is controlling the end effector and tell it to try something else
- Basically, there are few options that are truly appealing, as they are likely to cause either an error in the solution or a possible discontinuity in the motion

# Taking Too Long

- In a time critical situation, we might just limit the iteration to a maximum number of steps
- Alternately, we could use internal timers to limit it to an actual time in seconds

# Iteration Step Size

- We want to limit the step size we take by scaling it by some value $\beta$ where $0 < \beta \leq 1$
- As $\beta$ is just a scalar, we can actually choose it after computing $\Delta\mathbf{\Phi}$
- A reasonable approach is to limit the stepping of the joint rotations so that we never rotate more than some threshold value (maybe 5 degrees) in a single iteration
- To do this, we compute $\Delta\mathbf{\Phi}$ without $\beta$, then check to see if any values of $\Delta\mathbf{\Phi}$ exceed our threshold

$$\beta = \text{threshold} / \max(\text{threshold}, \max(\text{abs}(\Delta\varphi_i)))$$

# Iteration Step Size

$\Delta \mathbf{e} = \mathbf{g} - \mathbf{e}$

$\Delta \mathbf{\Phi} = J^{-1} \cdot \Delta \mathbf{e}$

$\beta = \text{threshold} / \max(\text{threshold}, \max(\text{abs}(\Delta \varphi_i)))$

$\mathbf{\Phi} = \mathbf{\Phi} + \beta(\Delta \mathbf{\Phi})$

- Note that this works for any IK method that interatively generates $\Delta \mathbf{\Phi}$ (JI, JT, JP, etc.)

# Other IK Issues

# Joint Limits

- A simple and reasonably effective way to handle joint limits is to simply clamp the pose vector as a final step in each iteration
- One can’t compute a proper derivative at the limits, as the function is effectively discontinuous at the boundary
- The derivative going towards the limit will be 0, but coming away from the limit will be non-zero. This leads to an inequality condition, which can’t be handled in a continuous manner
- We could just choose whether to set the derivative to 0 or non-zero based on a reasonable guess as to which way the joint would go. This is easy in the JT method, but can potentially cause trouble in JI or JP

# Higher Order Approximation

- The first derivative gives us a linear approximation to the function
- We can also take higher order derivatives and construct higher order approximations to the function
- This is analogous to approximating a function with a Taylor series

# Repeatability

- If a given goal vector **g** always generates the same pose vector **Φ**, then the system is said to be repeatable
- This is not likely to be the case for redundant systems unless we specifically try to enforce it
- If we always compute the new pose by starting from the last pose, the system will probably not be repeatable
- If, however, we always reset it to a ‘comfortable’ default pose, then the solution should be repeatable
- One potential problem with this approach however is that it may introduce sharp discontinuities in the solution

# Multiple End Effectors

- Remember, that the Jacobian matrix relates each DOF in the skeleton to each scalar value in the **e** vector
- The components of the matrix are based on quantities that are all expressed in world space, and the matrix itself does not contain any actual information about the connectivity of the skeleton
- Therefore, we extend the IK approach to handle tree structures and multiple end effectors without much difficulty
- We simply add more DOFs to the end effector vector to represent the other quantities that we want to constrain
- However, the issue of scaling the derivatives becomes more important as more joints are considered

# Multiple Chains

- Another approach to handling tree structures and multiple end effectors is to simply treat it as several individual chains
- This works for characters often, as we can animate the body with a forward kinematic approach, and then animate each limb with IK by positioning the hand/foot as the end effector goal
- This can be faster and simpler, and actually offer a nicer way to control the character

# Geometric Constraints

- One can also add more abstract geometric constraints to the system
  - Constrain distances, angles within the skeleton
  - Prevent bones from intersecting each other or the environment
  - Apply different weights to the constraints to signify their importance
  - Have additional controls that try to maximize the ‘comfort’ of a solution
  - Etc.
- Welman talks about this in section 5

# Other IK Techniques

- Cyclic Coordinate Descent
  - This technique is more of a trigonometric approach and is more heuristic. It does, however, tend to converge in fewer iterations than the Jacobian methods, even though each iteration is a bit more expensive. Welman talks about this method in section 4.2
- Analytical Methods
  - For simple chains, one can directly invert the forward kinematic equations to obtain an exact solution. This method can be very fast, very predictable, and precisely controllable. With some finesse, one can even formulate good analytical solvers for more complex chains with multiple DOFs and redundancy
- Other Numerical Methods
  - There are lots of other general purpose numerical methods for solving problems that can be cast into **f**(**x**)=**g** format

# Jacobian Method as a Black Box

- The Jacobian methods were not invented for solving IK. They are a far more general purpose technique for solving systems of non-linear equations
- The Jacobian solver itself is a black box that is designed to solve systems that can be expressed as $\mathbf{f}(\mathbf{x})=\mathbf{g}$ ( $\mathbf{e}(\Phi)=\mathbf{g}$ )
- All we need is a method of evaluating **f** and **J** for a given value of **x** to plug it into the solver
- If we design it this way, we could conceivably swap in different numerical solvers (JI, JP, JT, damped least-squares, conjugate gradient…)

# Computing the Jacobian

# Computing the Jacobian Matrix

- We can take a geometric approach to computing the Jacobian matrix
- Rather than look at it in 2D, let's just go straight to 3D
- Let's say we are just concerned with the end effector position for now. Therefore, **e** is just a 3D vector representing the end effector position in world space. This also implies that the Jacobian will be an 3xN matrix where N is the number of DOFs
- For each joint DOF, we analyze how **e** would change if the DOF changed

# 1-DOF Rotational Joints

- We will first consider DOFs that represents a rotation around a single axis (1-DOF hinge joint)
- We want to know how the world space position **e** will change if we rotate around the axis. Therefore, we will need to find the axis and the pivot point in world space
- Let's say $\varphi_i$ represents a rotational DOF of a joint. We also have the offset $\mathbf{r}_i$ of that joint relative to it's parent and we have the rotation axis $\mathbf{a}_i$ relative to the parent as well
- We can find the world space offset and axis by transforming them by their parent joint's world matrix

# 1-DOF Rotational Joints

- To find the pivot point and axis in world space:

$$\mathbf{a}'_i = \mathbf{W}_{i-parent} \cdot \mathbf{a}_i$$

$$\mathbf{r}'_i = \mathbf{W}_{i-parent} \cdot \mathbf{r}_i$$

- Remember these transform as homogeneous vectors. **r** transforms as a position $[r_x\ r_y\ r_z\ 1]$ and **a** transforms as a direction $[a_x\ a_y\ a_z\ 0]$

# Rotational DOFs

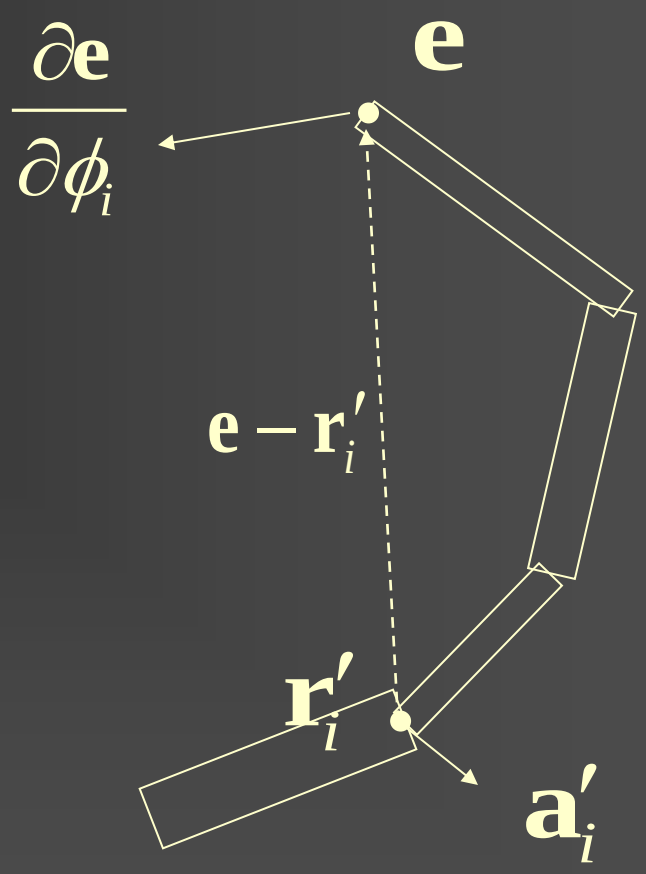
- Now that we have the axis and pivot point of the joint in world space, we can use them to find how **e** would change if we rotated around that axis

$$\frac{\partial \mathbf{e}}{\partial \phi_i} = \mathbf{a}'_i \times \left( \mathbf{e} - \mathbf{r}'_i \right)$$

- This gives us a column in the Jacobian matrix

# Rotational DOFs

$$\frac{\partial \mathbf{e}}{\partial \phi_i} = \mathbf{a}'_i \times \left(\mathbf{e} - \mathbf{r}'_i\right)$$




**a**'$_i$: unit length rotation axis in world space

**r**'$_i$: position of joint pivot in world space

**e**: end effector position in world space

# 3-DOF Rotational Joints

- For a 2-DOF or 3-DOF joint, we just have 2 or 3 columns in the Jacobian matrix, one for each separate rotation
- Remember that we want to do all of the computations in world space- however it is actually a little tricky to get the world space rotation axes
- Consider how we would find the world space x-axis of a 3-DOF ball joint
- Not only do we need to consider the parent's world matrix, but we need to include the rotation around the next two axes (y and z-axis) as well
- This is because those following rotations will rotate the first axis itself

# 3-DOF Rotational Joints

- For example, assuming we have a 3-DOF ball joint that rotates in XYZ order:

$$x-dof: \quad \mathbf{a}'_i = \mathbf{W}_{parent} \cdot \mathbf{R}_z(\theta_z) \cdot \mathbf{R}_y(\theta_y) \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^T$$

$$y-dof: \quad \mathbf{a}'_i = \mathbf{W}_{parent} \cdot \mathbf{R}_z(\theta_z) \cdot \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}^T$$

$$z-dof: \quad \mathbf{a}'_i = \mathbf{W}_{parent} \cdot \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}^T$$

- Where $R_y(\theta_y)$ and $R_z(\theta_z)$ are y and z rotation matrices

# 3-DOF Rotational Joints

- Remember that a 3-DOF XYZ ball joint's local matrix will look something like this:

$$\mathbf{L}(\theta_x, \theta_y, \theta_z) = \mathbf{T}(\mathbf{r}) \cdot \mathbf{R}_z(\theta_z) \cdot \mathbf{R}_y(\theta_y) \cdot \mathbf{R}_x(\theta_x)$$

- Where $R_x(\theta_x)$, $R_y(\theta_y)$, and $R_z(\theta_z)$ are x, y, and z rotation matrices, and $T(\mathbf{r})$ is a translation by the (constant) joint offset
- So it's world matrix looks like this:

$$\mathbf{W} = \mathbf{W}_{parent} \cdot \mathbf{T}(\mathbf{r}) \cdot \mathbf{R}_z(\theta_z) \cdot \mathbf{R}_y(\theta_y) \cdot \mathbf{R}_x(\theta_x)$$

# 3-DOF Rotational Joints

- Once we have each axis in world space, each one will get a column in the Jacobian matrix
- At this point, it is essentially handled as three 1-DOF joints, so we can use the same formula for computing the derivative as we did earlier:

$$\frac{\partial \mathbf{e}}{\partial \phi_i} = \mathbf{a}'_i \times \left( \mathbf{e} - \mathbf{r}'_i \right)$$

- We repeat this for each of the three axes

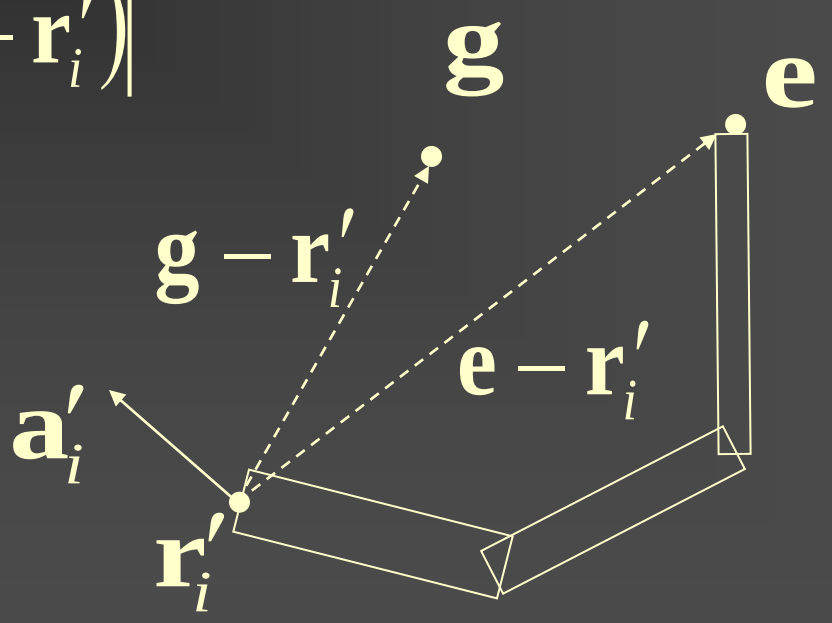
# Quaternion Joints

- What about a quaternion joint? How do we incorporate them into our IK formulation?
- We will assume that a quaternion joint is capable of rotating around any axis
- However, since we are trying to find a way to move **e** towards **g**, we should pick the best possible axis for achieving this

$$\mathbf{a}'_i = \frac{\left(\mathbf{e} - \mathbf{r}'_i\right) \times \left(\mathbf{g} - \mathbf{r}'_i\right)}{\left|\left(\mathbf{e} - \mathbf{r}'_i\right) \times \left(\mathbf{g} - \mathbf{r}'_i\right)\right|}$$

# Quaternion Joints

$$\mathbf{a}'_i = \frac{(\mathbf{e} - \mathbf{r}'_i) \times (\mathbf{g} - \mathbf{r}'_i)}{\left|(\mathbf{e} - \mathbf{r}'_i) \times (\mathbf{g} - \mathbf{r}'_i)\right|}$$



# Quaternion Joints

- We compute $\mathbf{a}_i$' directly in world space, so we don't need to transform it
- Now that we have $\mathbf{a}_i$', we can just compute the derivative the same way we would do with any other rotational axis

$$\frac{\partial \mathbf{e}}{\partial \phi_i} = \mathbf{a}'_i \times \left(\mathbf{e} - \mathbf{r}'_i\right)$$

- We must remember what axis we use, so that later, when we've computed $\Delta\varphi_i$, we know how to update the quaternion

# Translational DOFs

- For translational DOFs, we start in the same way, namely by finding the translation axis in world space
- If we had a prismatic joint (1-DOF translation) that could translate along an arbitrary axis $\mathbf{a}_i$ defined in the parent's space, we can use:

$$\mathbf{a'}_i = \mathbf{W}_{i-parent} \cdot \mathbf{a}_i$$

# Translational DOFs

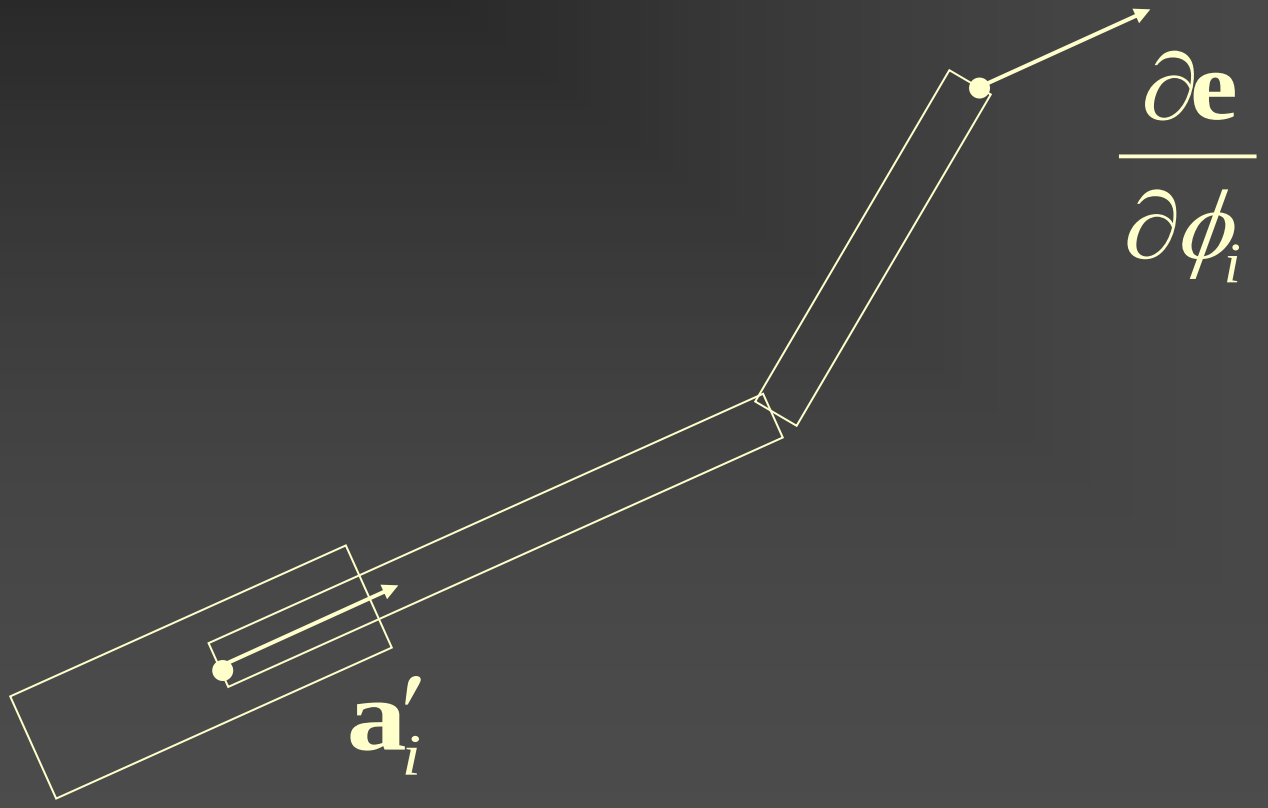
- For a more general 3-DOF translational joint that just translates along the local x, y, and z-axes, we don't need to do the same thing that we did for rotation
- The reason is that for translations, a change in one axis doesn't affect the other axes at all, so we can just use the same formula and plug in the x, y, and z axes [1 0 0 0], [0 1 0 0], [0 0 1 0] to get the 3 world space axes
- Note: this will just return the **a**, **b**, and **c** axes of the parent's world space matrix, and so we don't actually have to compute them!

# Translational DOFs

- As with rotation, each translational DOF is still treated separately and gets its own column in the Jacobian matrix
- A change in the DOF value results in a simple translation along the world space axis, making the computation trivial:

$$\frac{\partial \mathbf{e}}{\partial \phi_i} = \mathbf{a}'_i$$

# Translational DOFs



# Building the Jacobian

- To build the entire Jacobian matrix, we just loop through each DOF and compute a corresponding column in the matrix
- If we wanted, we could use more elaborate joint types (scaling, translation along a path, shearing…) and still compute an appropriate derivative
- If absolutely necessary, we could always resort to computing a numerical approximation to the derivative

# Units & Scaling

- What about units?
- Rotational DOFs use radians and translational DOFs use meters (or some other measure of distance)
- How can we combine their derivatives into the same matrix?
- Well, it’s really a bit of a hack, but we just combine them anyway
- If desired, we can scale any column to adjust how much the IK will favor using that DOF

# Units & Scaling

- For example, we could scale all rotations by some constant that causes the IK to behave how we would like
- Also, we could use this as an additional way to get control over the behavior of the IK
- We can store an additional parameter for each DOF that defines how ‘stiff’ it should behave
- If we scale the derivative larger (but preserve direction), the solution will compensate with a smaller value for $\Delta\varphi i$, therefore making it act stiff
- There are several proposed methods for automatically setting the stiffness to a reasonable default value. They generally work based on some function of the length of the actual bone. The Welman paper talks about this.

# End Effector Orientation

# End Effector Orientation

- We've examined how to form the columns of a Jacobian matrix for a position end effector with 3 DOFs
- How do we incorporate orientation of the end effector?
- We will add more DOFs to the end effector vector **e**
- Which method should we use to represent the orientation? (Euler angles? Quaternions?…)
- Actually, a popular method is to use the 3 DOF scaled axis representation!

# Scaled Rotation Axis

- We learned that any orientation can be represented as a single rotation around some axis
- Therefore, we can store an orientation as an 3D vector
  - The direction of the vector is the rotation axis
  - The length of the vector is the angle to rotate in radians
- This method has some properties that work well with the Jacobian approach
  - Continuous and consistent
  - No redundancy or extra constraints
  - It’s also a nice method to store incremental changes in rotation

# 6-DOF End Effector

- If we are concerned about both the position and orientation of the end effector, then our **e** vector should contain 6 numbers
- But remember, we don't actually need the **e** vector, we really just need the Δ**e** vector
- To generate Δ**e**, we compare the current end effector position/orientation (matrix **E**) to the goal position/orientation (matrix **G**)
- The first 3 components of Δ**e** represent the desired change in position: β(**G.d - E.d**)
- The next 3 represent a desired change in orientation, which we will express as a scaled axis vector

# Desired Change in Orientation

- We want to choose a rotation axis that rotates **E** in to **G**
- We can compute this using some quaternions:

  $\mathbf{M}=\mathbf{E}^{-1}\cdot\mathbf{G}$

  **q**.FromMatrix(**M**);
- This gives us a quaternion that represents a rotation from **E** to **G**
- To extract out the rotation axis and angle, we just remember that:

$$\mathbf{q} = \left[ \cos\frac{\theta}{2} \quad a_x \sin\frac{\theta}{2} \quad a_y \sin\frac{\theta}{2} \quad a_z \sin\frac{\theta}{2} \right]$$

- We can then scale the final axis by β

# End Effector

- So we now can define our goal with a matrix and come up with some desired change in end effector values that will bring us closer to that goal:

$$\Delta \mathbf{e} = \begin{bmatrix} \Delta t_x & \Delta t_y & \Delta t_z & \Delta \theta_x & \Delta \theta_y & \Delta \theta_z \end{bmatrix}^T$$

- We must now compute a Nx6 Jacobian matrix, where each column represents how a particular DOF will affect both the position and orientation of the end effector

# Rotational DOFs

- We need to compute additional derivatives that show how the end effector orientation changes with respect to an incremental change in each DOF
- We will use the scaled axis to represent the incremental change
- For a rotational DOF, we first find the rotation axis in world space (as we did earlier)
- Then- we're done! That axis already represents the incremental rotation caused by that DOF
- By default, the length of the axis should be 1, indicating that a change of 1 in the DOF value results in a rotation of 1 radian around the axis. We can scale this by a stiffness value if desired

# Rotational DOFs

- The column in the Nx6 Jacobian matrix corresponding to a rotational DOF is:

$$\mathbf{J}_i = \frac{\partial \mathbf{e}}{\partial \phi_i} = \begin{bmatrix} \left[\mathbf{a}'_i \times \left(\mathbf{e}_{pos} - \mathbf{r}'_i\right)\right]^T \\ \left[\mathbf{a}'_i\right]^T \end{bmatrix}$$

- **a**’ is the rotation axis in world space
- **r**’ is the pivot point in world space
- $\mathbf{e}_{pos}$ is the position of the end effector in world space

# Translational DOFs

- Translational DOFs don't affect the end effector orientation, so their contribution to the derivative of orientation will be [0 0 0]

$$\mathbf{J}_i = \frac{\partial \mathbf{e}}{\partial \phi_i} = \begin{bmatrix} [\mathbf{a}'_i]^T \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

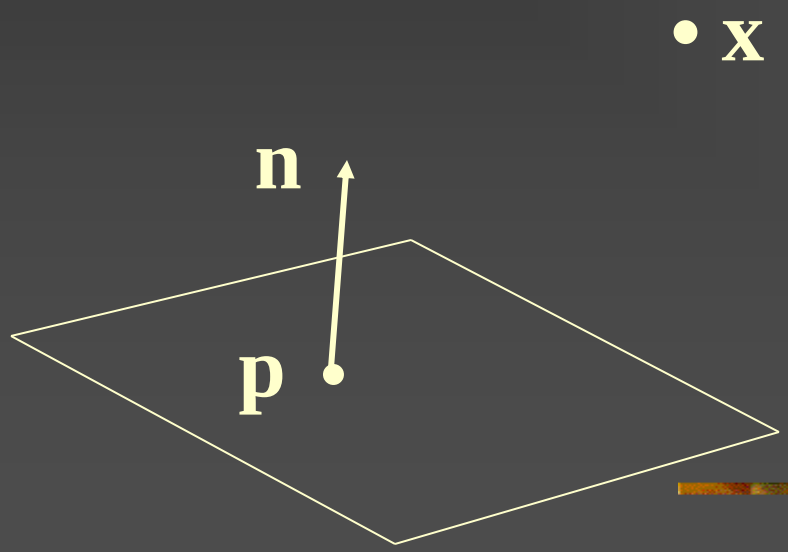
# Midterm Samples

# Midterm

- The midterm is on Thursday, Feb 6
- It has 10 questions, and is worth 10% of your grade
- It covers everything up to and including today's lecture
- No books, notes, cell phones, etc.

# Question 1: Distance to Plane

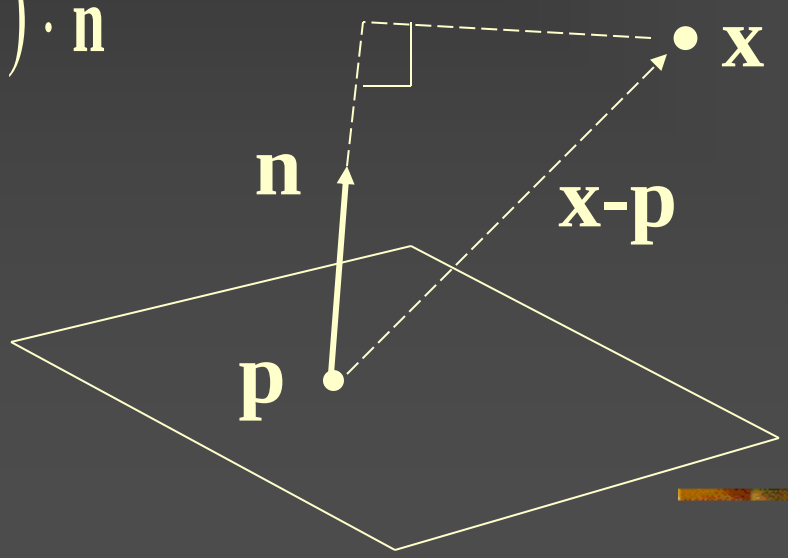
- A plane is described by a point **p** on the plane and a unit normal **n**. Find the distance from point **x** to the plane



# Question 1: Distance to Plane

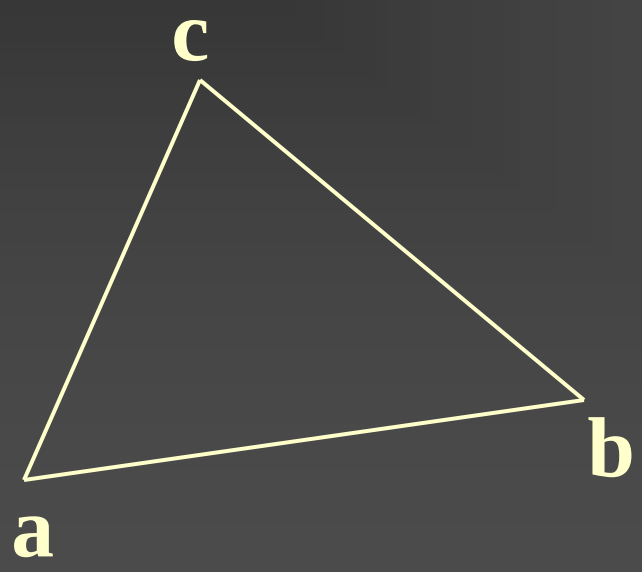
- The distance is the length of the projection of **x-p** onto **n**:

$$dist = (\mathbf{x} - \mathbf{p}) \cdot \mathbf{n}$$



# Question 2: Normal of a Triangle

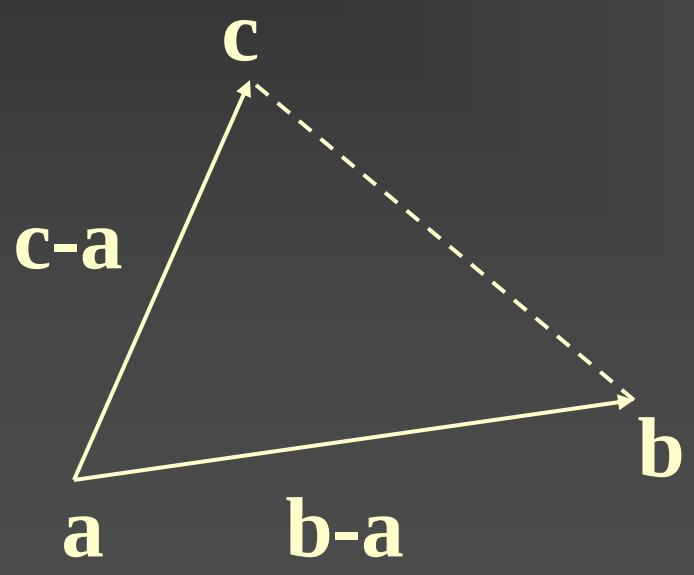
- Find the unit length normal of the triangle defined by 3D points **a**, **b**, and **c**



# Question 2: Normal of a Triangle

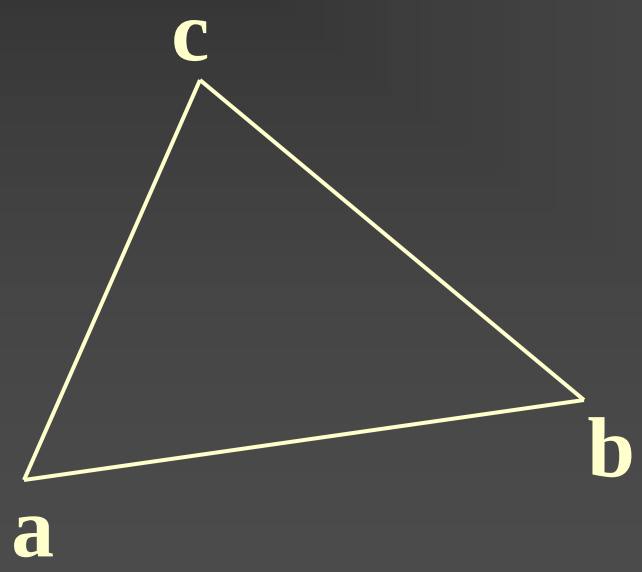
$$\mathbf{n}^* = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$$

$$\mathbf{n} = \frac{\mathbf{n}^*}{|\mathbf{n}^*|}$$



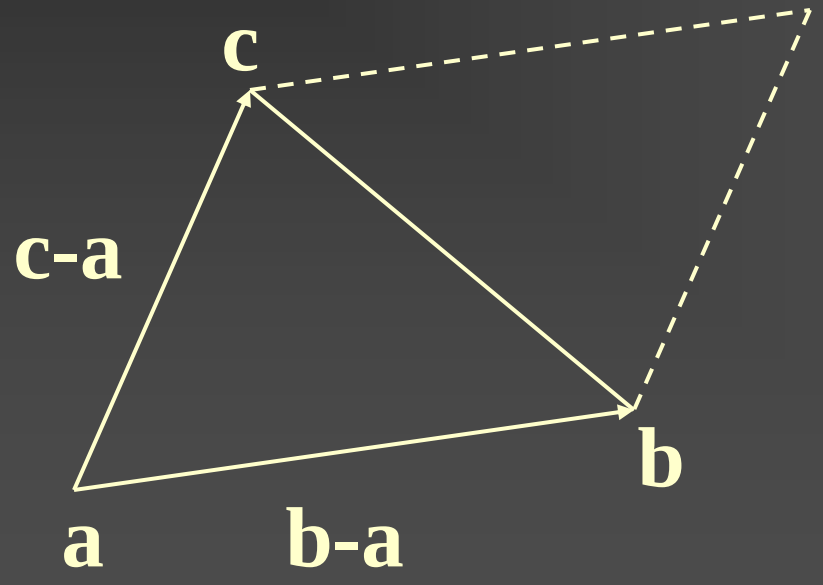
# Question 3: Area of a Triangle

- Find the area of the triangle defined by 3D points **a**, **b**, and **c**



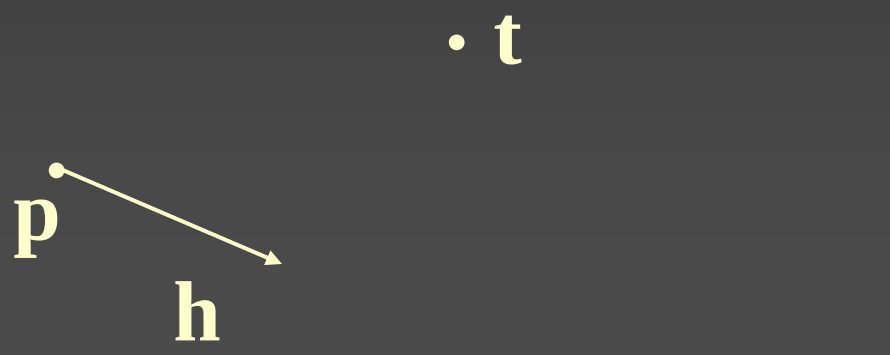
# Question 3: Area of a Triangle

$$area = \frac{1}{2}\left|\left(\mathbf{b}-\mathbf{a}\right)\times\left(\mathbf{c}-\mathbf{a}\right)\right|$$

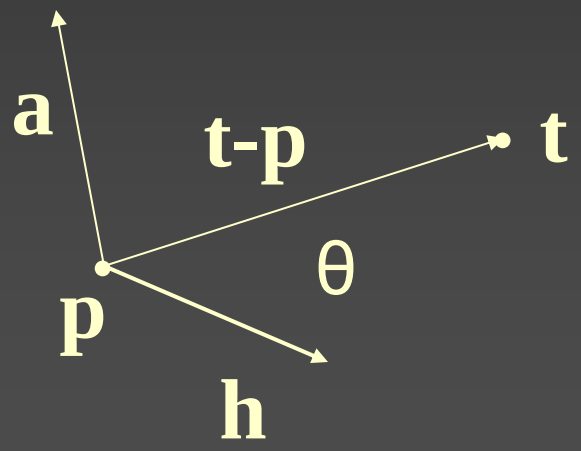


# Question 4: Alignment to Target

- An object is at position **p** with a unit length heading of **h**. We want to rotate it so that the heading is facing some target **t**. Find a unit axis **a** and an angle θ to rotate around.



# Question 4: Alignment to Target




$$\mathbf{a} = \frac{\mathbf{h} \times (\mathbf{t} - \mathbf{p})}{\left|\mathbf{h} \times (\mathbf{t} - \mathbf{p})\right|}$$

$$\theta = \cos^{-1}\left(\frac{\mathbf{h} \cdot (\mathbf{t} - \mathbf{p})}{\left|(\mathbf{t} - \mathbf{p})\right|}\right)$$

# Question 5: IK Iteration

- List three reasons to stop iterating an IK solution

# Question 5: IK Iteration

- List three reasons to stop iterating an IK solution

1. Reached goal (within tolerance)
2. Stuck in local minimum
3. Taking too long