Inverse Kinematics (part 1)

CSE169: Computer Animation

Instructor: Steve Rotenberg
UCSD, Winter 2020

Welman, 1993

- "Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulation", Chris Welman, 1993
- Masters thesis on IK algorithms
- Examines Jacobian methods and Cyclic Coordinate Descent (CCD)
- Please read sections 1-4 (about 40 pages)

Forward Kinematics

- The local and world matrix construction within the skeleton is an implementation of *forward kinematics*
- Forward kinematics refers to the process of computing world space geometric descriptions (matrices…) based on joint DOF values (usually rotation angles and/or translations)

Kinematic Chains

- For today, we will limit our study to linear kinematic chains, rather than the more general hierarchies (i.e., stick with individual arms & legs rather than an entire body with multiple branching chains)

End Effector

- The joint at the root of the chain is sometimes called the *base*
- The joint (bone) at the leaf end of the chain is called the *end effector*
- Sometimes, we will refer to the end effector as being a bone with position and orientation, while other times, we might just consider a point on the tip of the bone and only think about it’s position

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 Kinematics

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

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

Inverse Kinematics

- 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
- In other words, it is the inverse of the forward kinematics problem

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

Inverse Kinematics Issues

- IK is challenging because while f() may be relatively easy to evaluate, $f^{-1}()$ usually isn't
- For one thing, there may be several possible solutions for **Φ**, or there may be no solutions
- Even if there is a solution, it may require complex and expensive computations to find it
- As a result, there are many different approaches to solving IK problems

Analytical vs. Numerical Solutions

- One major way to classify IK solutions is into analytical and numerical methods
- Analytical methods attempt to mathematically solve an exact solution by directly inverting the forward kinematics equations. This is only possible on relatively simple chains.
- Numerical methods use approximation and iteration to converge on a solution. They tend to be more expensive, but far more general purpose.
- Today, we will examine a numerical IK technique based on Jacobian matrices

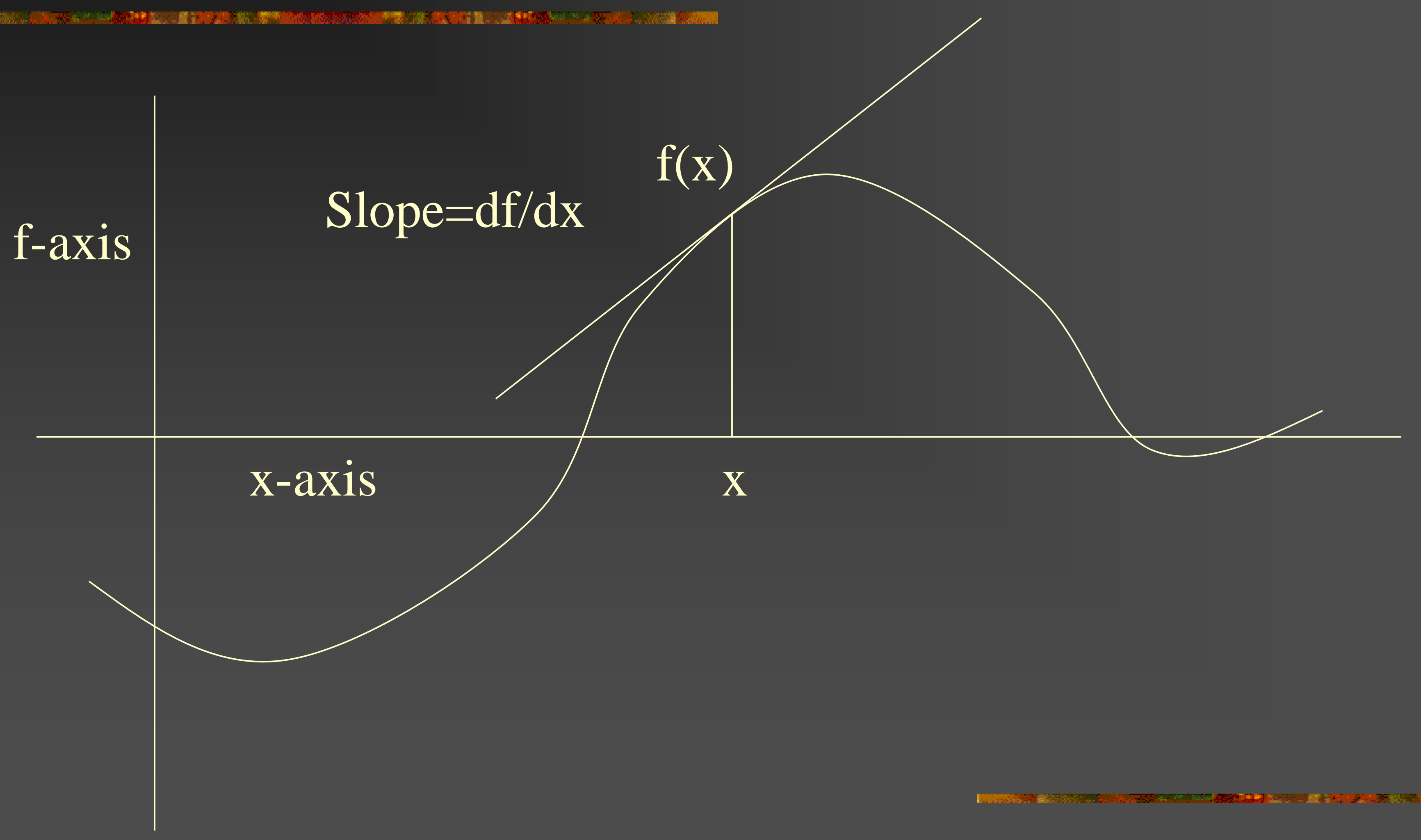
Calculus Review

Derivative of a Scalar Function

- If we have a scalar function f of a single variable x, we can write it as f(x)
- The derivative of the function with respect to x is df/dx
- The derivative is defined as:

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Derivative of a Scalar Function



Derivative of $f(x)=x^2$

For example : $f(x) = x^2$

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{(x+\Delta x)^2 - (x)^2}{\Delta x} \qquad = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

$$= \lim_{\Delta x \to 0} \frac{x^2 + 2x\Delta x + \Delta x^2 - x^2}{\Delta x}$$

$$= \lim_{\Delta x \to 0} \frac{2x\Delta x + \Delta x^2}{\Delta x}$$

$$= \lim_{\Delta x \to 0} (2x + \Delta x) = \boxed{2x}$$

Exact vs. Approximate

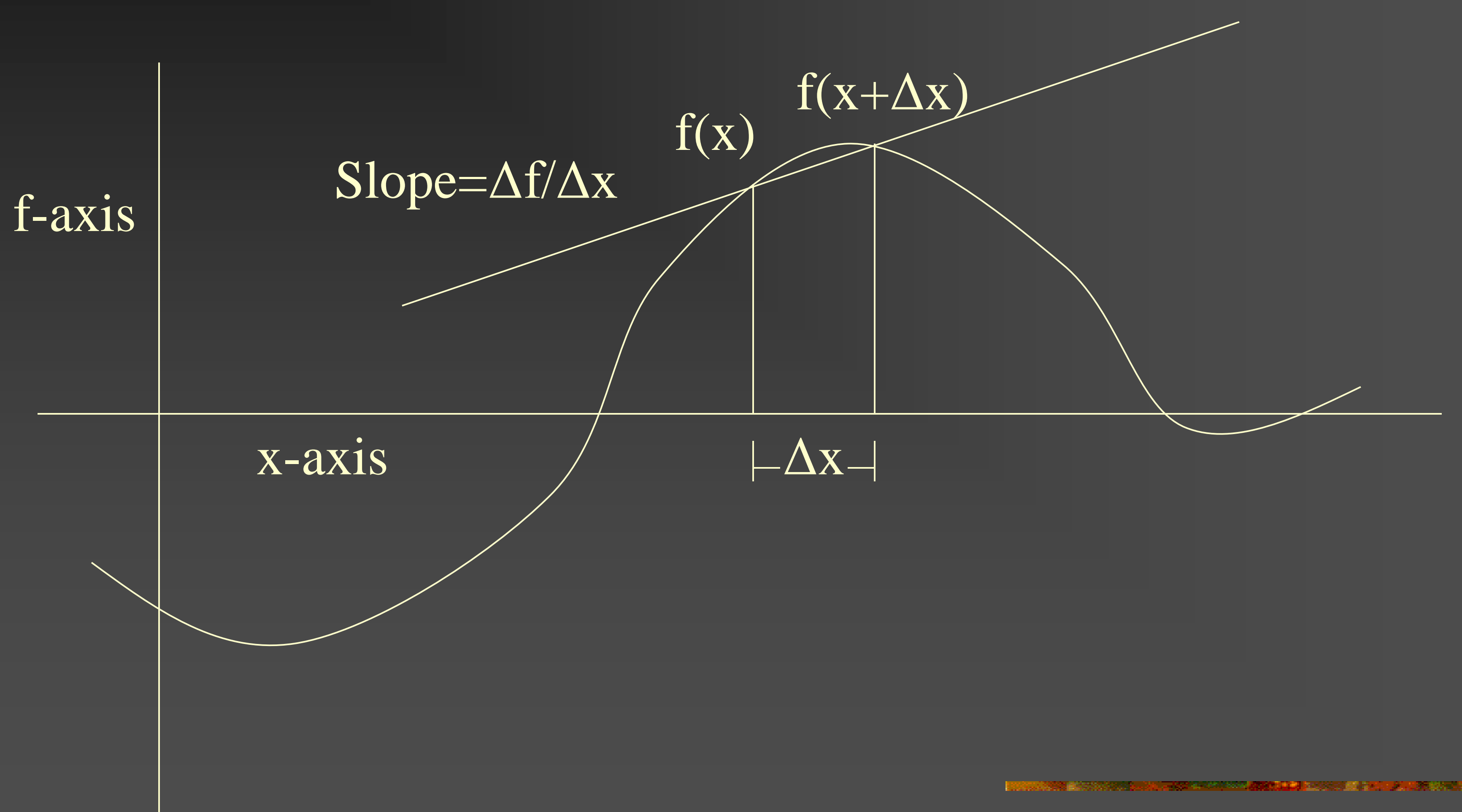
- Many algorithms require the computation of derivatives
- Sometimes, we can compute analytical derivatives. For example:

$$f(x) = x^2 \qquad \frac{df}{dx} = 2x$$

- Other times, we have a function that's too complex, and we can't compute an exact derivative
- As long as we can evaluate the function, we can always approximate a derivative

$$\frac{df}{dx} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \quad \text{for small } \Delta x$$

Approximate Derivative



Nearby Function Values

- If we know the value of a function and its derivative at some x, we can estimate what the value of the function is at other points near x

$$\frac{\Delta f}{\Delta x} \approx \frac{df}{dx}$$

$$\Delta f \approx \Delta x \frac{df}{dx}$$

$$f\left(x + \Delta x\right) \approx f\left(x\right) + \Delta x \frac{df}{dx}$$

Finding Solutions to f(x)=0

- There are many mathematical and computational approaches to finding values of x for which f(x)=0
- One such way is the *gradient descent* method
- If we can evaluate f(x) and df/dx for any value of x, we can always follow the gradient (slope) in the direction towards 0

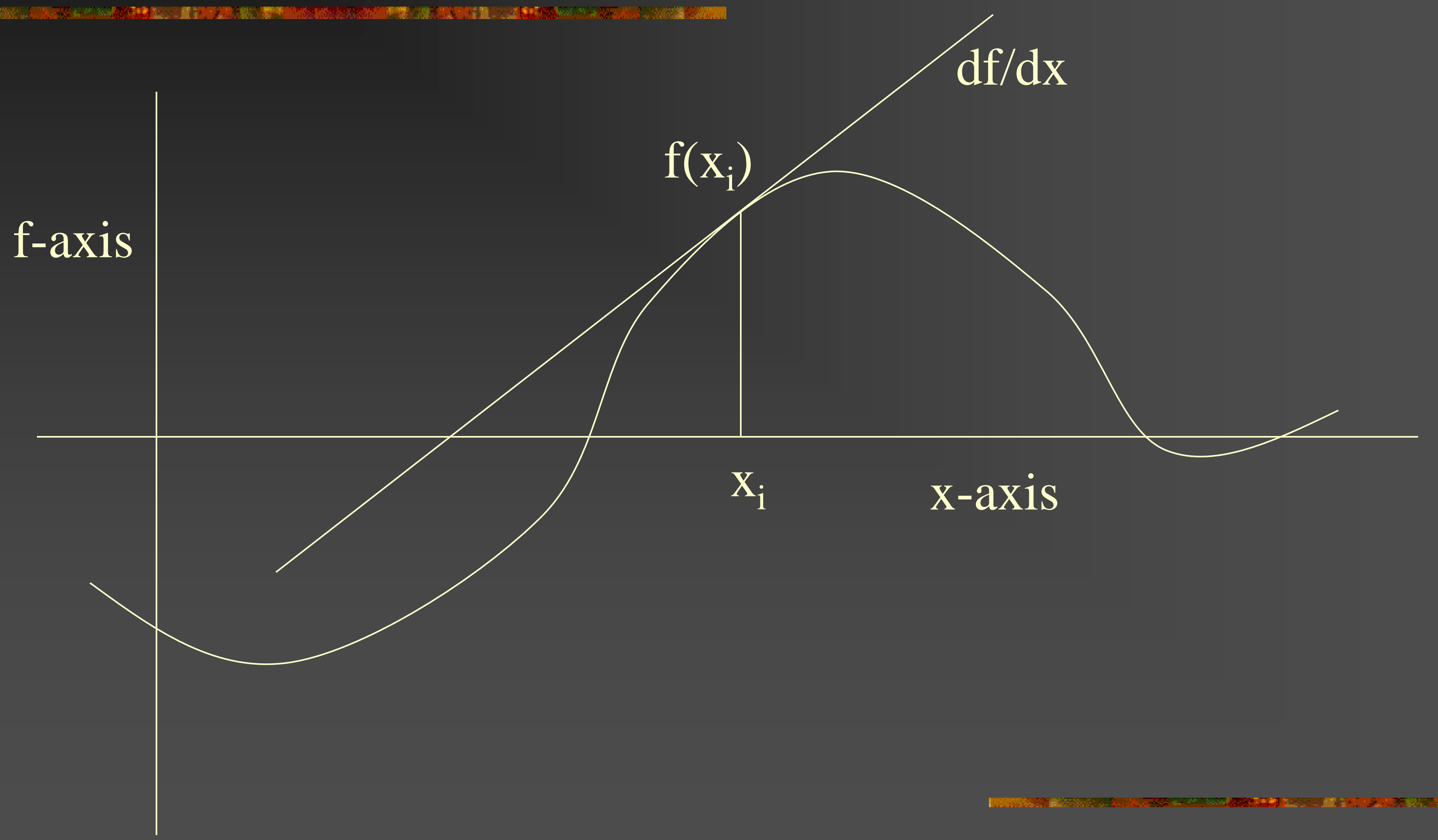
Gradient Descent

- We want to find the value of x that causes f(x) to equal 0
- 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 $f(x_N)=0$
- For each step, we try to choose a value of Δx that will bring us closer to our goal
- We can use the derivative as an approximation to the slope of the function and use this information to move ‘downhill’ towards zero

Gradient Descent



Minimization

- If $f(x_i)$ is not 0, the value of $f(x_i)$ can be thought of as an error. The goal of gradient descent is to minimize this error, and so we can refer to it as a *minimization* algorithm
- Each step Δx we take results in the function changing its value. We will call this change Δf.
- Ideally, we could have $\Delta f = -f(x_i)$. In other words, we want to take a step Δx that causes Δf to cancel out the error
- More realistically, we will just hope that each step will bring us closer, and we can eventually stop when we get 'close enough'
- This iterative process involving approximations is consistent with many *numerical* algorithms

Choosing Δx Step

- If we have a function that varies heavily, we will be safest taking small steps
- If we have a relatively smooth function, we could try stepping directly to where the linear approximation passes through 0

Choosing Δx Step

- If we want to choose Δx to bring us to the value where the slope passes through 0, we can use:

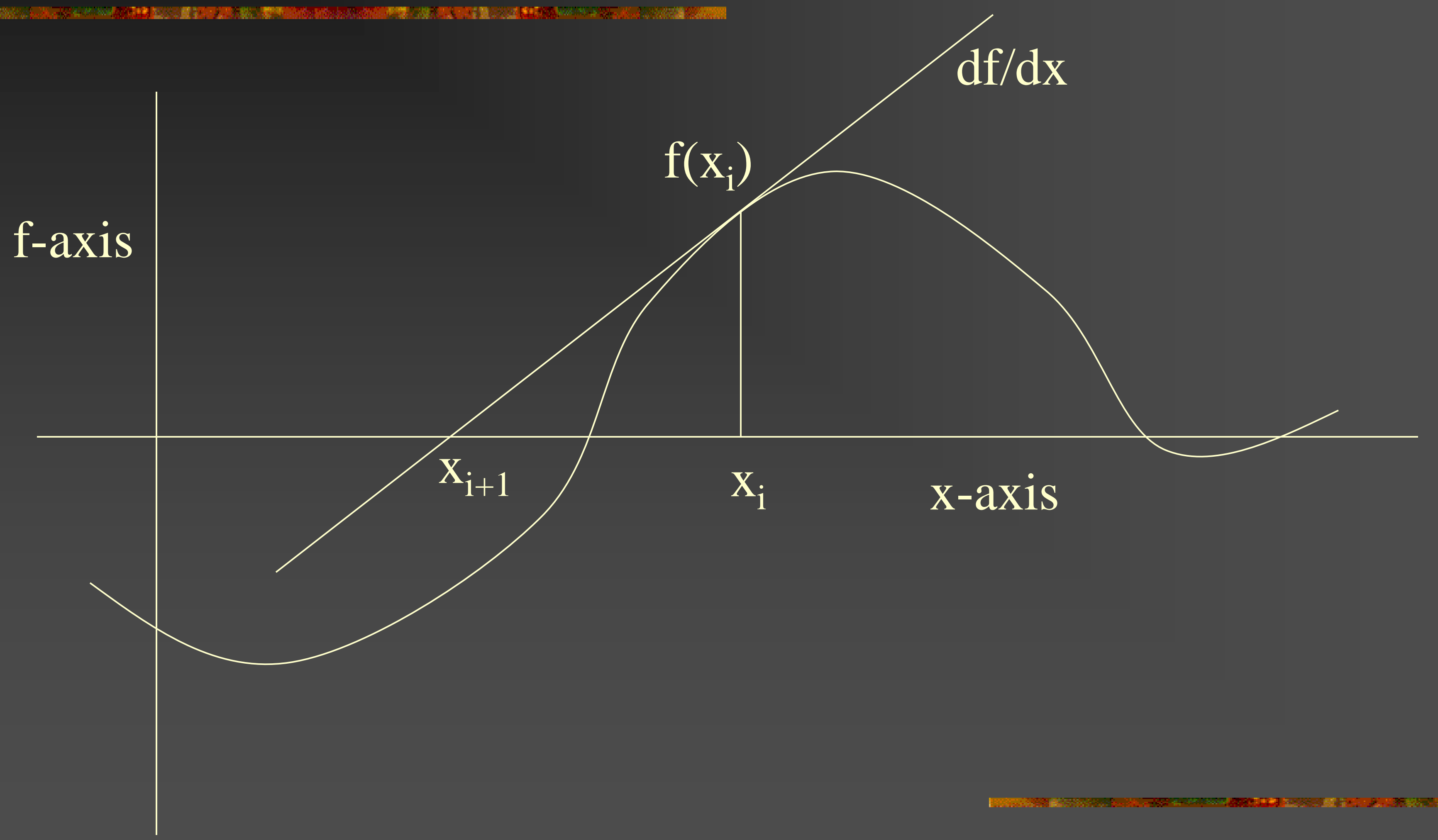
$$\frac{\Delta f}{\Delta x} \approx \frac{df}{dx}$$

$$\Delta f \approx \Delta x \frac{df}{dx}$$

$$-f(x_i) \approx \Delta x \frac{df}{dx}$$

$$\Delta x = -f(x_i)\left(\frac{df}{dx}\right)^{-1}$$

Gradient Descent

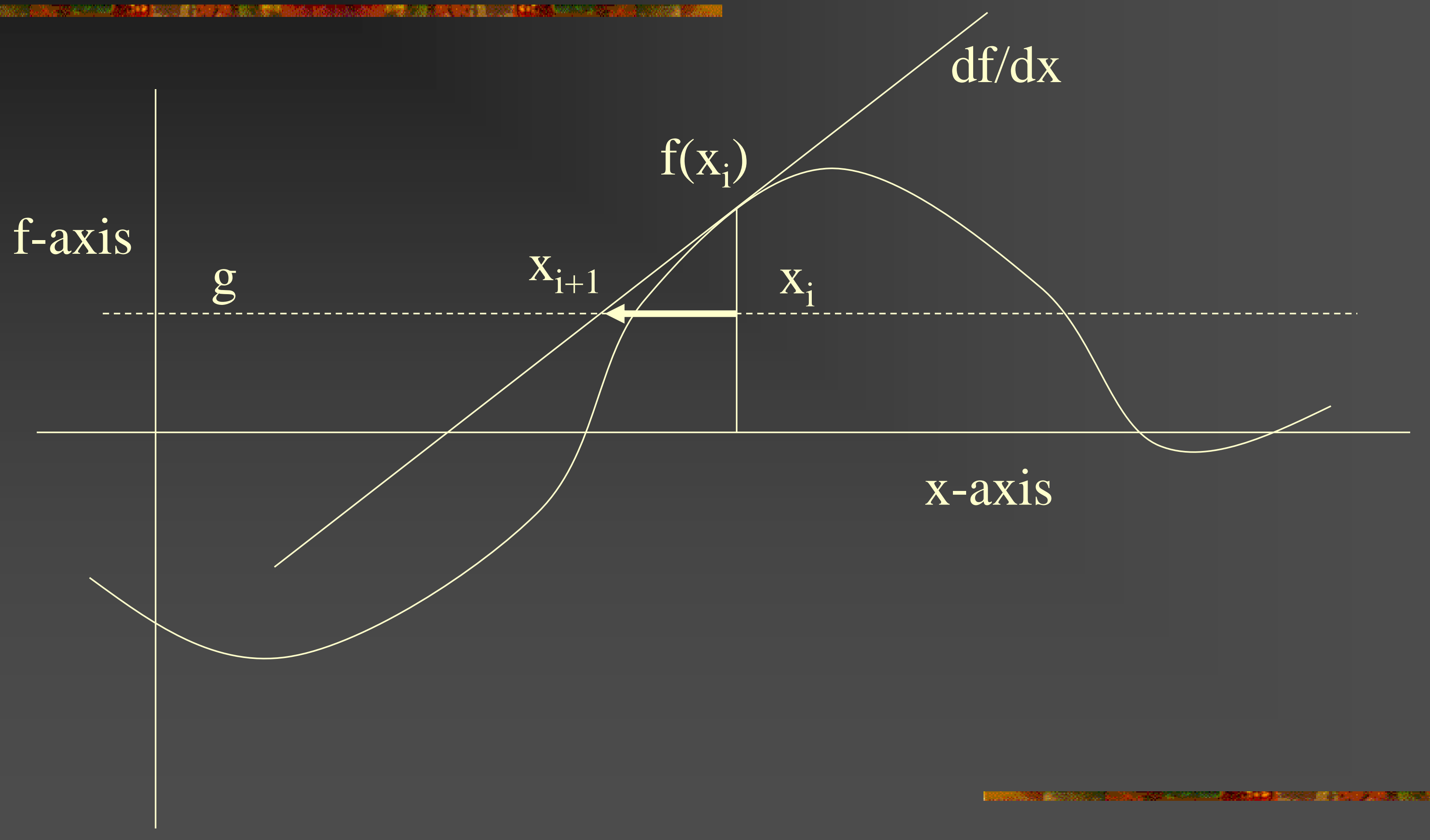


Solving f(x)=g

- If we don't want to find where a function equals some value 'g' other than zero, we can simply think of it as minimizing f(x)-g and just step towards g:

$$\Delta x = \left(g - f\left(x_i\right)\right)\left(\frac{df}{dx}\right)^{-1}$$

Gradient Descent for f(x)=g



Taking Safer Steps

- Sometimes, we are dealing with non-smooth functions with varying derivatives
- Therefore, our simple linear approximation is not very reliable for large values of Δx
- There are many approaches to choosing a more appropriate (smaller) step size
- One simple modification is to add a parameter β to scale our step (0≤ β ≤1)

$$\Delta x = \beta\left(g - f\left(x_i\right)\right)\left(\frac{df}{dx}\right)^{-1}$$

Inverse of the Derivative

- By the way, for *scalar* derivatives:

$$\left(\frac{df}{dx}\right)^{-1} = \frac{1}{\left(\frac{df}{dx}\right)} = \frac{dx}{df}$$

Gradient Descent Algorithm

$x_0 = \text{initial starting value}$

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

while $(f_n \neq g)$ {

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

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

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

}

Stopping the Descent

- At some point, we need to stop iterating
- Ideally, we would stop when we get to our goal
- Realistically, we will stop when we get to within some acceptable tolerance
- However, occasionally, we may get ‘stuck’ in a situation where we can’t make any small step that takes us closer to our goal
- We will discuss some more about this later

Derivative of a Vector Function

- If we have a vector function **r** which represents a particle's position as a function of time t:

$$\mathbf{r} = \begin{bmatrix} r_x & r_y & r_z \end{bmatrix}$$

$$\frac{d\mathbf{r}}{dt} = \begin{bmatrix} \frac{dr_x}{dt} & \frac{dr_y}{dt} & \frac{dr_z}{dt} \end{bmatrix}$$

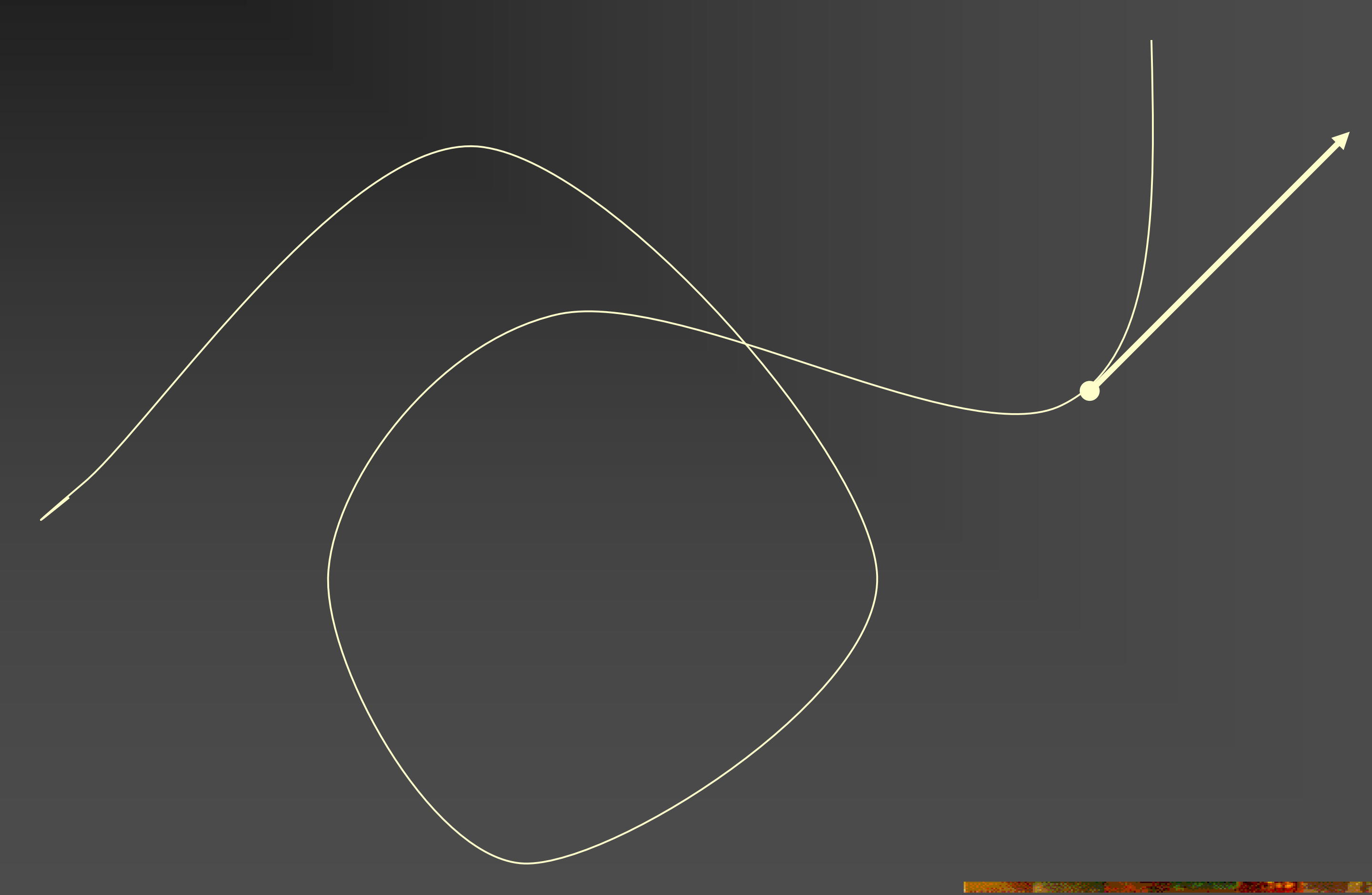
Derivative of a Vector Function

- By definition, the derivative of position is called velocity, and the derivative of velocity is acceleration

$$\mathbf{v} = \frac{d\mathbf{r}}{dt}$$

$$\mathbf{a} = \frac{d\mathbf{v}}{dt} = \frac{d^2\mathbf{r}}{dt^2}$$

Derivative of a Vector Function



Vector Derivatives

- We’ve seen how to take a derivative of a scalar vs. a scalar, and a vector vs. a scalar
- What about the derivative of a scalar vs. a vector, or a vector vs. a vector?

Vector Derivatives

- Derivatives of scalars with respect to vectors show up often in field equations, used in fun subjects like fluid dynamics, solid mechanics, and other physically based animation techniques. If we are lucky, we'll have time to look at these later in the quarter
- Today, however, we will be looking at derivatives of vector quantities with respect to other vector quantities

Jacobians

- A Jacobian is a vector derivative with respect to another vector
- If we have a vector valued function of a vector of variables $\mathbf{f}(\mathbf{x})$, the Jacobian is a matrix of partial derivatives- one partial derivative for each combination of components of the vectors
- The Jacobian matrix contains all of the information necessary to relate a change in any component of $\mathbf{x}$ to a change in any component of $\mathbf{f}$
- The Jacobian is usually written as $J(\mathbf{f},\mathbf{x})$, but you can really just think of it as $d\mathbf{f}/d\mathbf{x}$

Jacobians

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

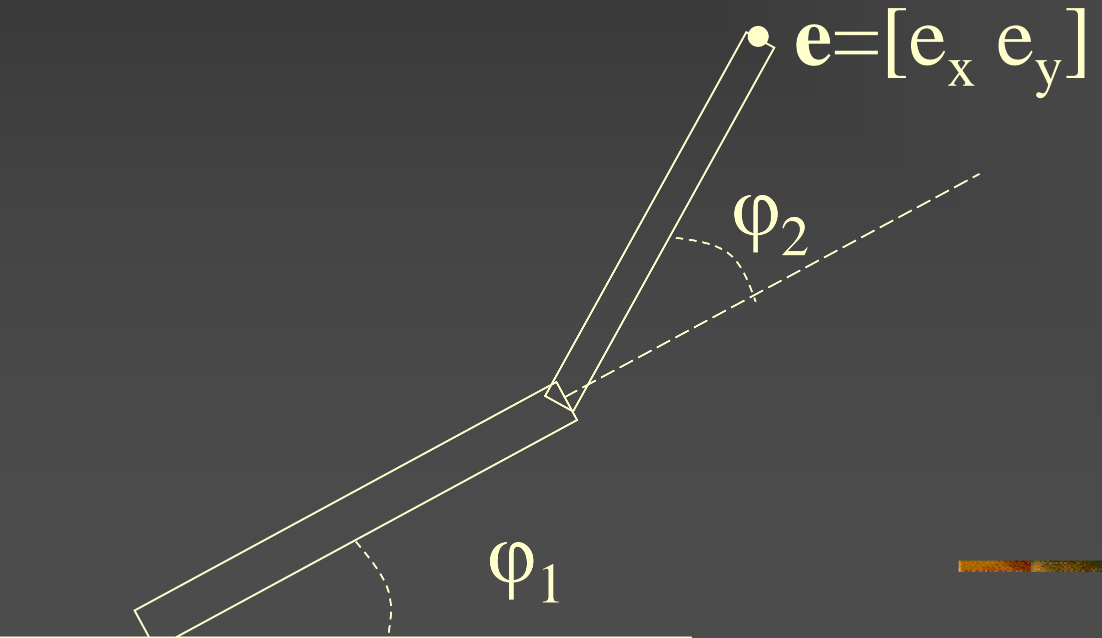
Partial Derivatives

- The use of the ∂ symbol instead of d for partial derivatives really just implies that it is a single component in a vector derivative
- For *many* practical purposes, an individual partial derivative behaves like the derivative of a scalar with respect to another scalar

Jacobian Inverse Kinematics

Jacobians

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



Jacobians

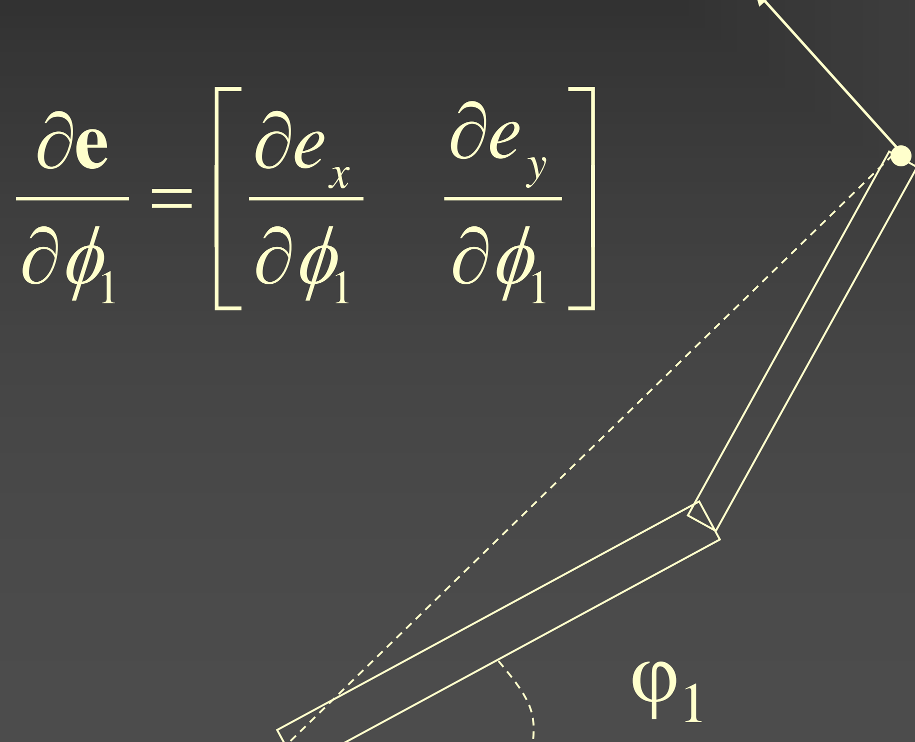
- The Jacobian matrix J(**e**,**Φ**) shows how each component of **e** varies with respect to each joint angle

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

Jacobians

- Consider what would happen if we increased φ_1 by a small amount. What would happen to **e** ?

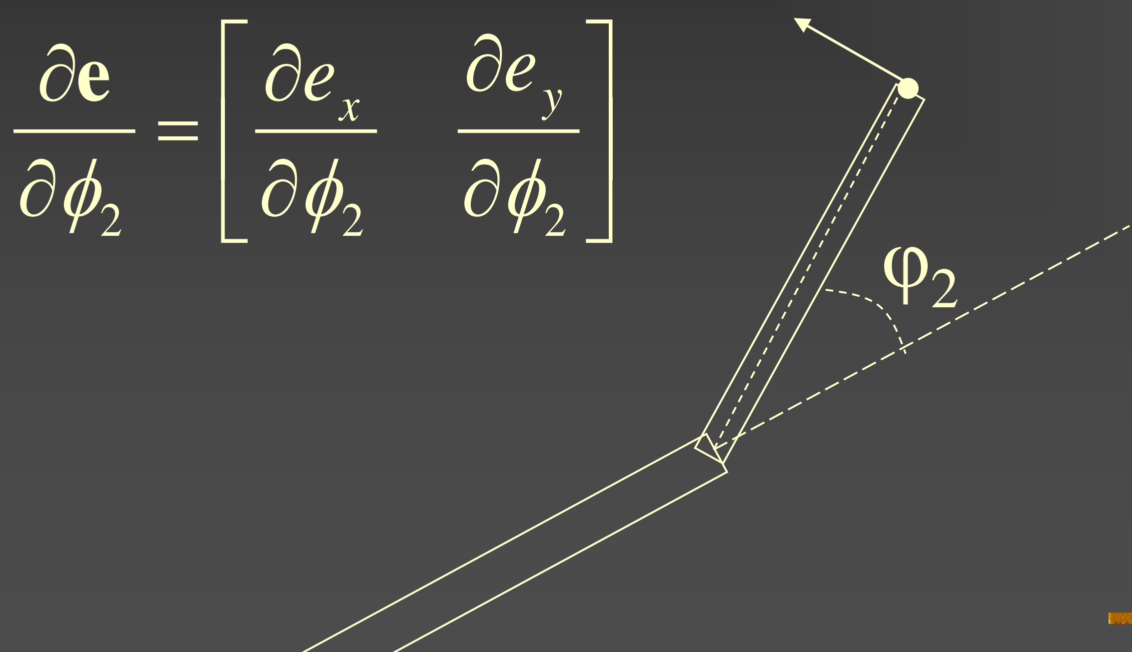
$$\frac{\partial \mathbf{e}}{\partial \phi_1} = \begin{bmatrix} \frac{\partial e_x}{\partial \phi_1} & \frac{\partial e_y}{\partial \phi_1} \end{bmatrix}$$



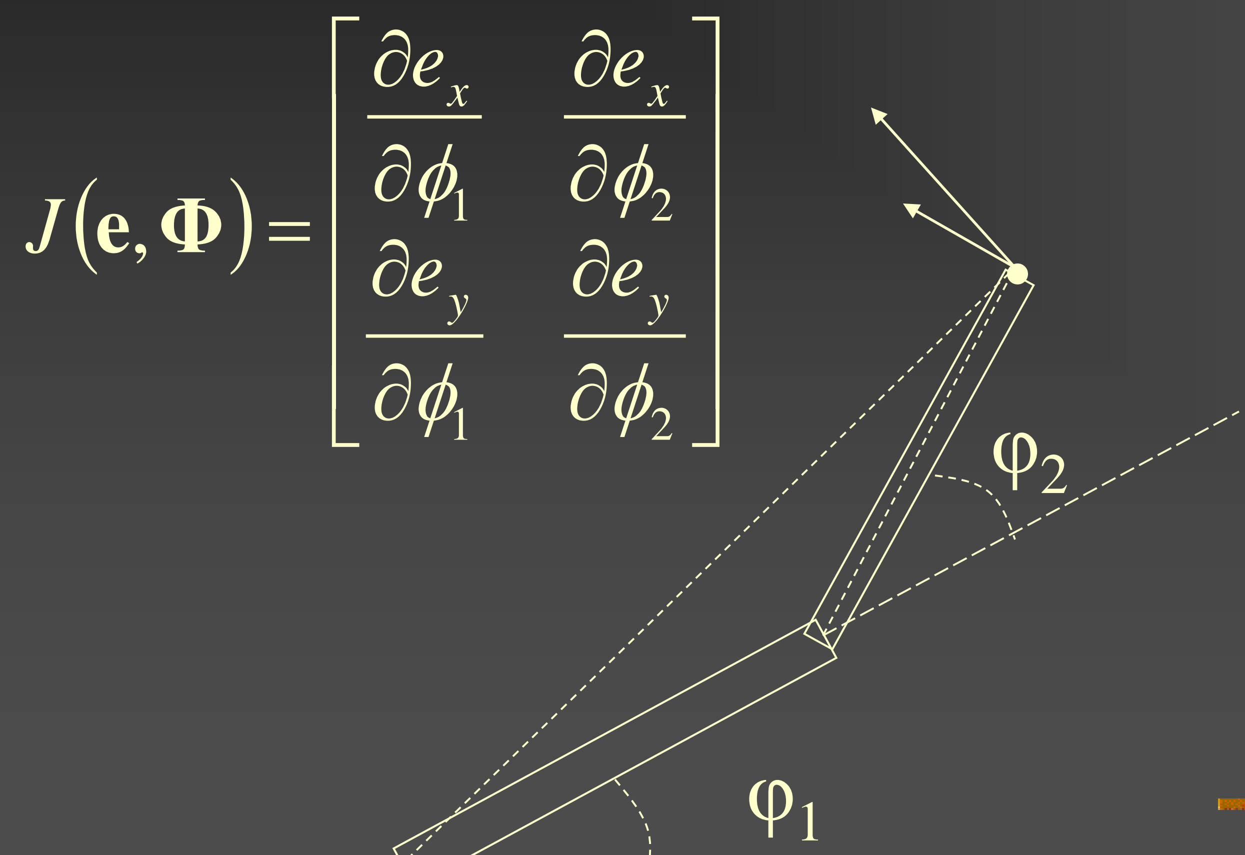
Jacobians

- What if we increased φ_2 by a small amount?

$$\frac{\partial \mathbf{e}}{\partial \phi_2} = \begin{bmatrix} \frac{\partial e_x}{\partial \phi_2} & \frac{\partial e_y}{\partial \phi_2} \end{bmatrix}$$



Jacobian for a 2D Robot Arm



Jacobian Matrices

- Just as a scalar derivative df/dx of a function f(x) can vary over the domain of possible values for x, the Jacobian matrix J(**e**,**Φ**) varies over the domain of all possible poses for **Φ**
- For any given joint pose vector **Φ**, we can explicitly compute the individual components of the Jacobian matrix

Jacobian as a Vector Derivative

- Once again, sometimes it helps to think of:

$$J(\mathbf{e}, \mathbf{\Phi}) = \frac{d\mathbf{e}}{d\mathbf{\Phi}}$$

because J(**e**,**Φ**) contains all the information we need to know about how to relate changes in any component of **Φ** to changes in any component of **e**

Incremental Change in Pose

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

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

Incremental Change in Effector

- What if we wanted to move the end effector by a small amount Δ**e**. What small change Δ**Φ** will achieve this?

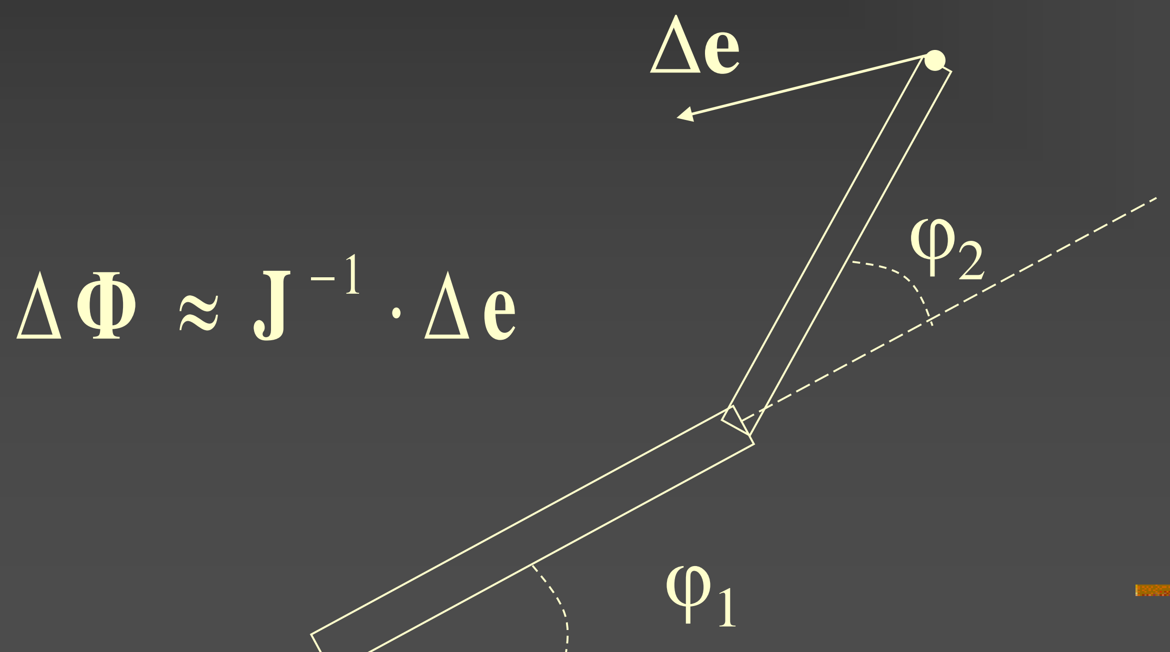
$$\Delta \mathbf{e} \approx \mathbf{J} \cdot \Delta \mathbf{\Phi}$$

$$so:$$

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

Incremental Change in **e**

- Given some desired incremental change in end effector configuration Δ**e**, we can compute an appropriate incremental change in joint DOFs Δ**Φ**



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

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

End Effector Goals

- If **Φ** represents the current set of joint DOFs and **e** represents the current end effector DOFs, we will use **g** to represent the goal DOFs that we want the end effector to reach

Choosing $\Delta\mathbf{e}$

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

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

- We would hope then, that the corresponding value of $\Delta\boldsymbol{\Phi}$ 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 $0 \leq \beta \leq 1$

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
        Δe = β(g - e)         // pick approximate step to take
        ΔΦ = J^-1 · Δe        // compute change in joint DOFs
        Φ = Φ + ΔΦ            // apply change to DOFs
        Compute new e vector  // apply forward
                              // kinematics to see
                              // where we ended up
}
```

A Few Questions

- How do we compute **J** ?
- How do we invert **J** to compute $\mathbf{J}^{-1}$?
- How do we choose β (step size)
- How do we determine when to stop the iteration?

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 φ_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

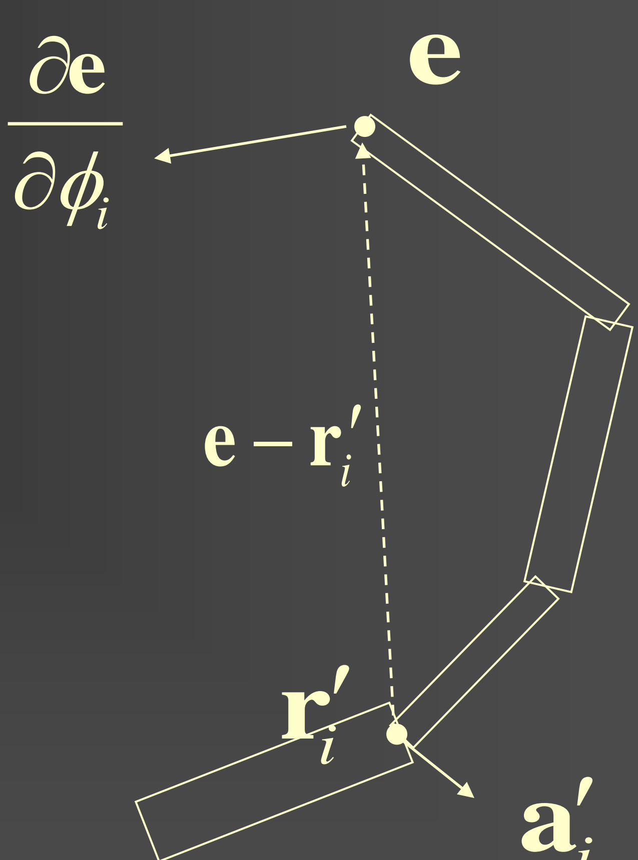
- Now that we have the axis and pivot point of the joint in world space, we can use them to find how **e** would change if we rotated around that axis

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

- 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

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

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…

To Be Continued…

Project 3

- Load an .anim file and play back a keyframed animation on a skinned character
- Due Thursday, February 13, 4:50 pm
- Extra Credit:
  - Display the channel curve
  - Simple channel editor

Anim File

```
animation {
    range  [time_start]  [time_end]
    numchannels [num]
    channel {
        extrapolate [extrap_in] [extrap_out]
        keys [numkeys] {
            [time] [value] [tangent_in] [tangent_out]
            …
        }
    }
    channel …
}
```

Project 3

- The first 3 channels will be the root translation (x,y,z)
- After that, there will be 3 rotational channels for every joint (x,y,z) in the same order that the joints are listed in the .skel file

Project 3

- Suggested classes:
  - Keyframe: stores time, value, tangents, cubics…
  - Channel: stores an array (or list) of Keyframes
  - Animation: stores an array of Channels
  - Player: stores pointer to an animation & pointer to skeleton. Keeps track of time, accesses animation data & poses the skeleton.
- Optional:
  - Rig: simple container for a skeleton, skin, and morphs
  - Pose: array of floats (or just use stl vector)
  - ChannelEditor: it's always nice to separate editor classes from the data that they edit