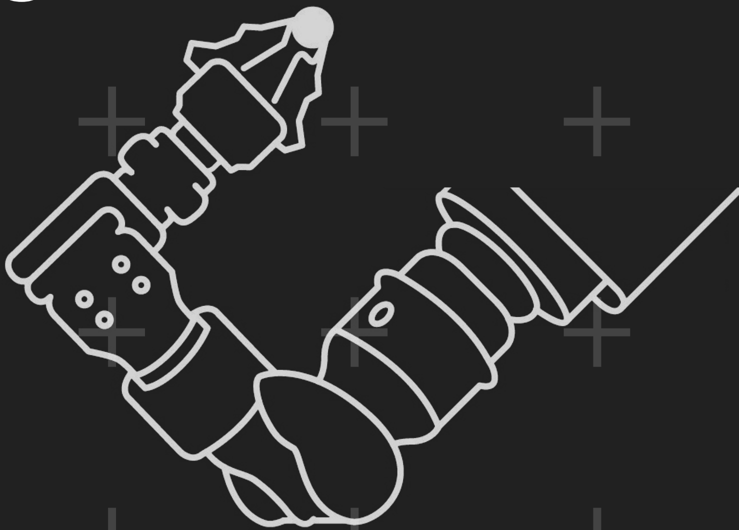
Inverse Kinematics

Numerical, **Sampling-based**, ***Analytical***, and ***Geometric*** Solvers



March 3, 2024
Ctrl^H Hackerspace

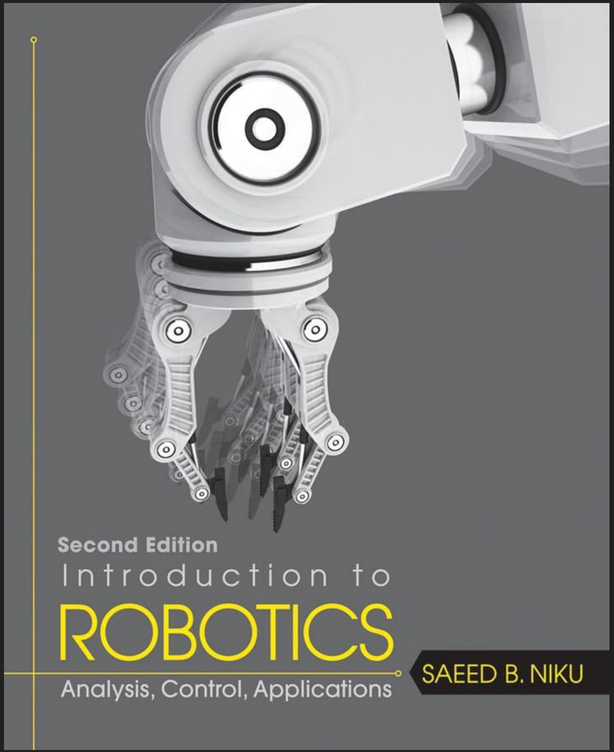
What is Inverse Kinematics (IK)?

- Inverse Kinematics, or IK, is a way to calculate robot arm joint variables given the desired end-effector position and/or orientation (pose).
- Some types of IK solutions are based on Forward Kinematics, or calculating end-effector pose given joint variables.
- A software plugin that can compute joint variables given the robot description and the desired goal pose is called an IK solver.

Why this presentation?

- Open-source **sampling-based** solver (**KDL**) can only find solutions for **6-DOF or greater** robots.
- Open-source **analytical** solver (**IKFast**) is outdated and has poor support for **mimic** and **fixed** joints.
- No **tutorials** for inverse kinematics with **less than 6-DOF** (only research papers and course materials, some of which are included in */resources* folder).
- Hobbyists working with **3-DOF** to **5-DOF** robots have no simple tools like setup wizards and no accessible instructional materials.

If you're looking for more background...



- Covers IK and Control Systems
- High-school math level for IK
- Differential equations and calculus for Control Systems
- Available at Ctrl^H library
- Available on Amazon ($10 used)

What we'll cover

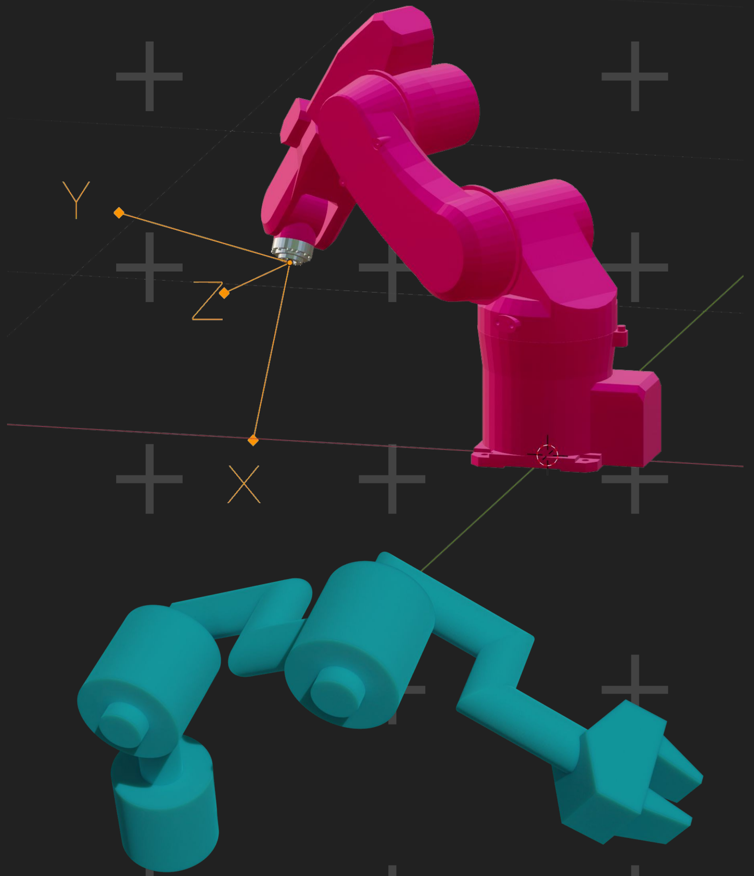
- Robot description and joints
- Forward Kinematics solutions
  - Denavit-Hartenberg (DH) parameters
- Numerical IK solutions
  - Gradient Descent
  - Newton-Raphson Iterator
- Sampling-based IK solutions
- Analytical IK solutions
  - IKFast
  - Solving a system of nonlinear equations
- Geometric IK solutions

What you’ll need

- Blender for visualizing concepts
- Visual Studio Code IDE for debugging C++
- C++ compiler
- Eigen3 library
- Matlab or Octave for solving nonlinear equations

Robot Description

- Both IK and FK require joint frames to work with
- Robot Description defines links and joints in a tree structure
- Joints define limits, offset, and movement axis
- Links define visual and simulation properties



Representing Joints

- Each joint is represented as a 4x4 (homogenous) matrix that encodes 3D *offset* and *orientation*.
  - Revolute joints: axis and angle of the joint
  - Prismatic joints: sliding offset of the joint
- This matrix will have one or more joint variables.
- All other terms are constants.
- Multiplying all joint matrices with their variables filled in will give you the end-effector pose.

Joint Matrix in Octave

To build a **joint matrix** in GNU **Octave**:

```
pkg load matgeom

Joint = ...

  createTranslation3d(0, 0, 0) * ...      % constant offset from previous

  createRotationOz(jointVariable) * ...   % variable rotation on Z axis

  createRotationOx(pi / 2);               % constant rotation on X axis
```

Joint Matrix in MatLab

To build a **joint matrix** in MathWorks **Matlab**:

```
Joint = ...
  makehgtform('translate', [0, 0, 0]) * ...   % constant offset from previous
  makehgtform('zrotate', jointVariable) * ... % variable rotation on Z axis
  makehgtform('xrotate', pi / 2)              % constant rotation on X axis
```

Later we'll define custom functions that can build matrices with symbols (*makehgtform* does not take symbols).

Denavit-Hartenberg Parameters

- Denavit-Hartenberg (DH) parameters are a convention for building a joint transformation matrix:
  - Joint offset d: translation on Z axis
  - Joint angle θ: rotation on Z axis
  - Link length a: translation on X axis
  - Joint twist α: rotation on X axis
- Developed to save computing power, but still a good convention and continues being taught.
- Revolute joints always rotate on Z axis.
- Prismatic joints always move on X axis.

Denavit-Hartenberg Matrix

Denavit-Hartenberg matrix can be built as follows:

```
[
cos(θ), -sin(θ) * cos(α), sin(θ) * sin(α), a * cos(θ)
sin(θ), cos(θ) * cos(α), -cos(θ) * sin(α), a * sin(θ)
0,      sin(α),           cos(α),          d
0,      0,                0,               1
]
```

Types of IK Solutions

- **Numerical** solvers measure **changes** to end effector **pose** resulting from **changes** in **joint variables**, until they converge on a solution within tolerance.
- **Sampling-based** solvers build a **graph** of random possible joint **states** and look for **connections** that lead to end effector reaching **goal pose**.
- **Analytical** solvers equate the **product of all joint matrices** with the end effector **pose**, and solve the resulting **system of nonlinear equations**.
- **Geometric** solvers use **trigonometry**.

Numerical IK Solvers

- Iteratively **sample** poses calculated from combinations of **joint variables** by using **Forward Kinematics.**
- Each joint variable is **increased** or **decreased** based on whether a **change** in this variable resulted in FK pose getting **closer** to or **further** away from the goal.
- The search **stops** when the pose is within **tolerance** of the goal or upon reaching a **timeout.**
- Can be done with **Gradient Descent** and **Newton-Raphson Iterator** among others.

Numerical IK - Gradient Descent

- The **amount** by which to **increase** or **decrease** each joint variable is calculated using a **gradient equation**:

$$(joints_{j\ n} - joints_{j\ n-1}) \ / \ (fk(joints_{j\ n}) - fk(joints_{j\ n-1}))$$

- j - joint index (0 to number of joints)
- n - current iteration
- joints - array of joint variables, one for each joint
- fk(joints) - forward kinematics pose from joints

Numerical IK - Newton-Raphson Iterator

- The amount by which to **increase** or **decrease** each joint variable is computed by using the **Jacobian matrix (J) inverse/transpose**, which describes **how much** the end effector moves and rotates on **each axis** when **each joint variable** changes.
- For each joint and end effector axis...

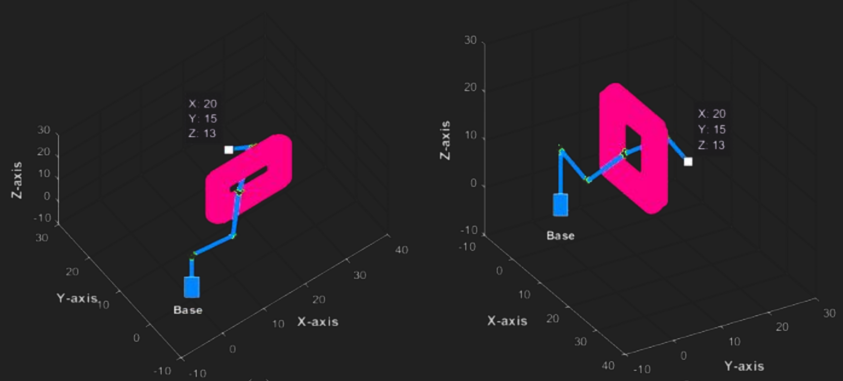
```
J[axis, joint] = (fk(joints… + Δ) - fk(joints…)) / Δ
error = goal - fk(joints)
joints += (Jtrans * error) * damping
```

Numerical IK - Newton-Raphson Iterator (Notes)

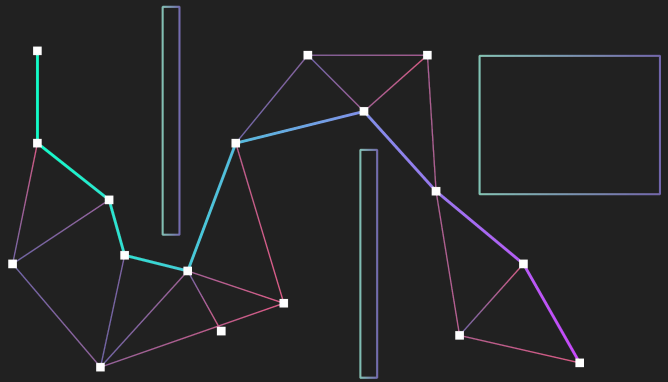
- Forward kinematics function needs to be able to calculate position of **any link** in the chain.
- Since Jacobian matrix encodes end effector **movements** in response to a **change** in any of the joint variables, **inverting** this matrix will give you the **change** in joint variables resulting from end effector **movement**.
- This algorithm changes joint variables by **Δ**, determines the resulting **error**, and adjusts the **direction of change** for each joint (for next time) based on error **increasing** or **decreasing**.

Sampling-based IK Solvers

- Create a **graph** of randomly sampled **states** (each state with different **joint variables**).
- Look for a **path** through this graph that avoids **collisions** and reaches the desired **goal pose**.



(a) (b)



Analytical IK

- Analytical solvers equate the product of all joint matrices with the end-effector pose (itself a matrix).
- This matrix equation then breaks down into a system of nonlinear equations (one for each matrix cell).

```
A1 * A2 * A3 == EE

LHS          == EE

// Left-Hand Side    // End Effector
```

```
{ LHS[1, 1] == EE[1, 1]
  LHS[1, 2] == EE[1, 2]
  LHS[1, 3] == EE[1, 3]
  ...
```

Analytical IK - Basic

- Ease into analytical IK by **auto-generating** a solution!
- **IKFast** is a **python script** included in OpenRAVE robotics toolbox.
- It uses **python equation solving libraries** like *sympy* and *lapack* to solve an equation that will return end effector pose given joint variables.
- The **solution** is then converted to **C++** and wrapped into a ROS interface so that it can be used as a **plugin**.

Analytical IK - Intermediate

- Get more comfortable by solving with pose from FK:
  - Setup the end-effector matrix w/constants
  - Setup the joint matrices w/variables and constants
  - Solve for joint variables:

```
solve(A1 * A2 * A3 == EE, [a1, a2, a3])

solve(A1 * A2 * A3 * A4 == EE, [a1, a2, a3, a4])

...
```

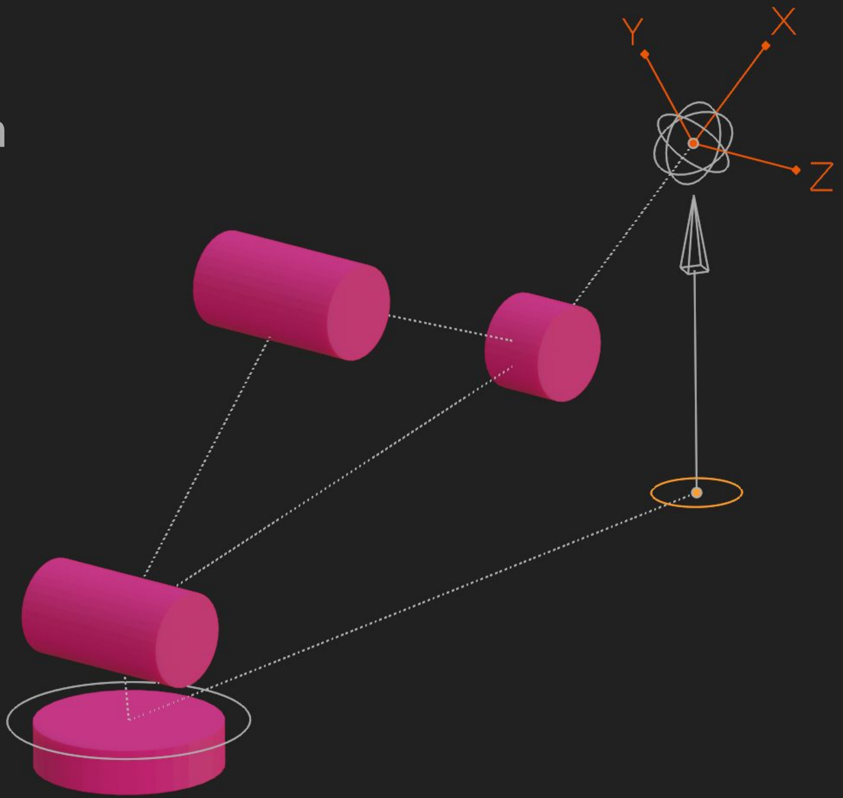
- The basic equation solver cannot handle more complex equations, less precision, and will not generate C++.

Analytical IK - Advanced

- Setup end-effector and joint matrices as before.
- Setup IK equation with joint matrices equal to end-effector pose as before.
- De-couple unknowns: multiply by inverse of joint frame(s) to decouple respective joint variable(s).
- Setup a system of 12 equations (last row is constant).
- Solve for any exposed variables and substitute.
- Lacking exposed variables, solve by combining pairs of equations and squaring both sides (*linearization*).

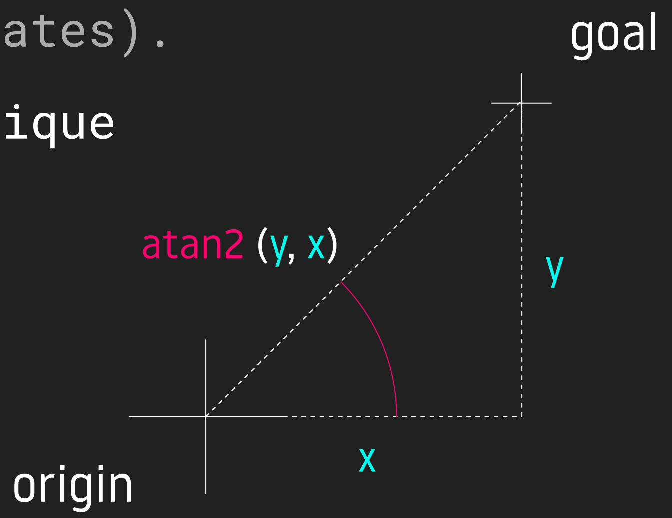
Geometric IK

- Solve for each variable on one plane at a time
- You'll be using a lot of inverse trig functions
- Trig identities (SOH-CAH-TOA)
- The Law of Cosines



Geometric IK - Arctangent

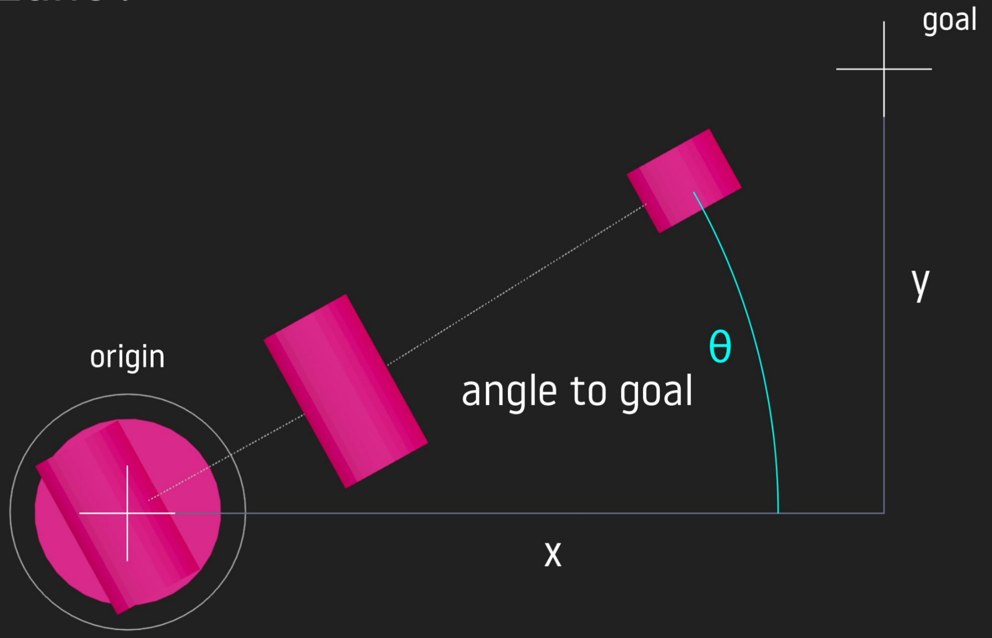
- Given cartesian coordinates on a plane, solve for the **angle** with ***arctangent***.
- 1-parameter arctangent is **ambiguous** (same angle for different coordinates).
- 2-parameter arctangent returns **unique angle** for different coordinates.



Geometric IK - Arctangent

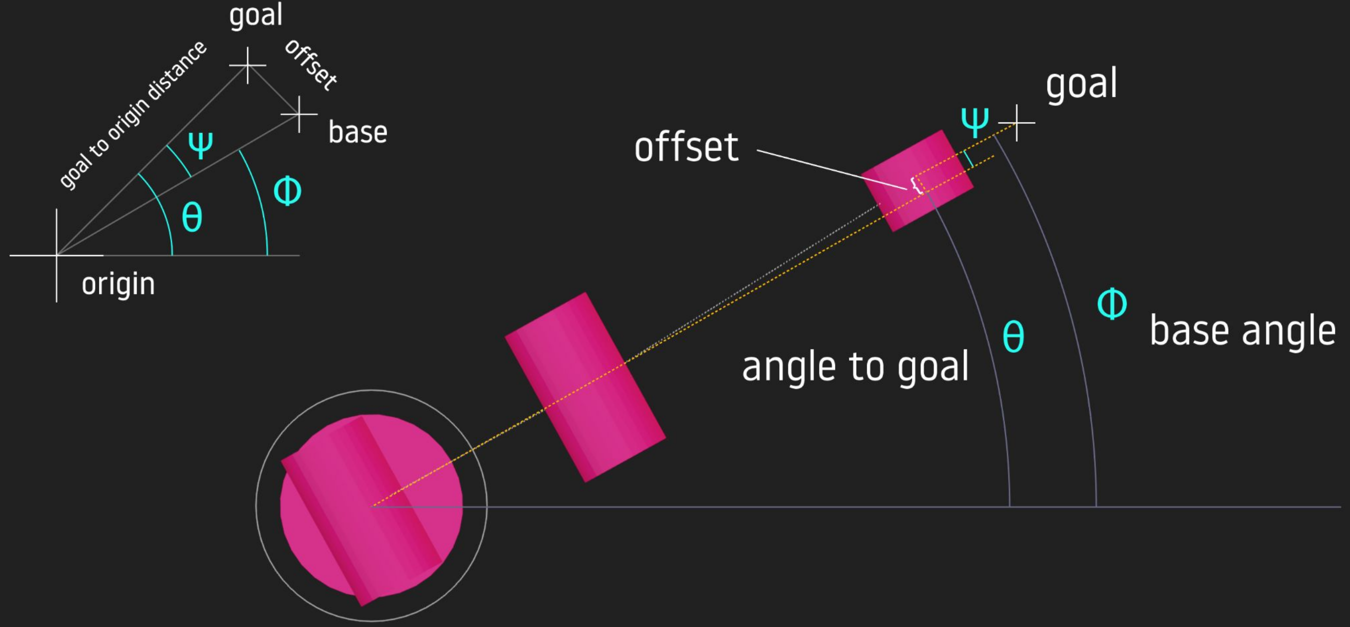
- Solving for **base** angle of a 3-DOF arm using **arctangent** on XY plane:

```
θ = atan2(y, x)
```



Geometric IK - Arcsine

- Solving for **base** angle w/**offset** using **arcsine:**

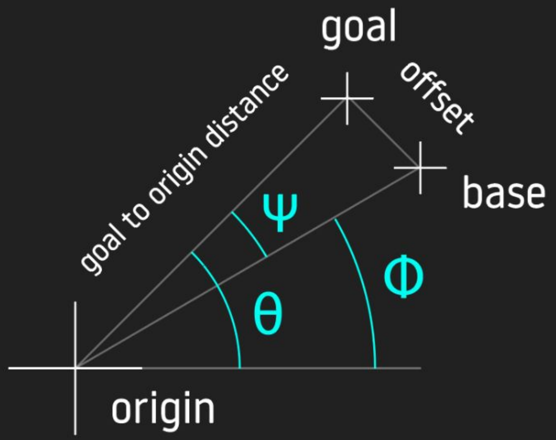


Geometric IK - Arcsine

- Solving for **base** angle w/**offset** using **arcsine**:

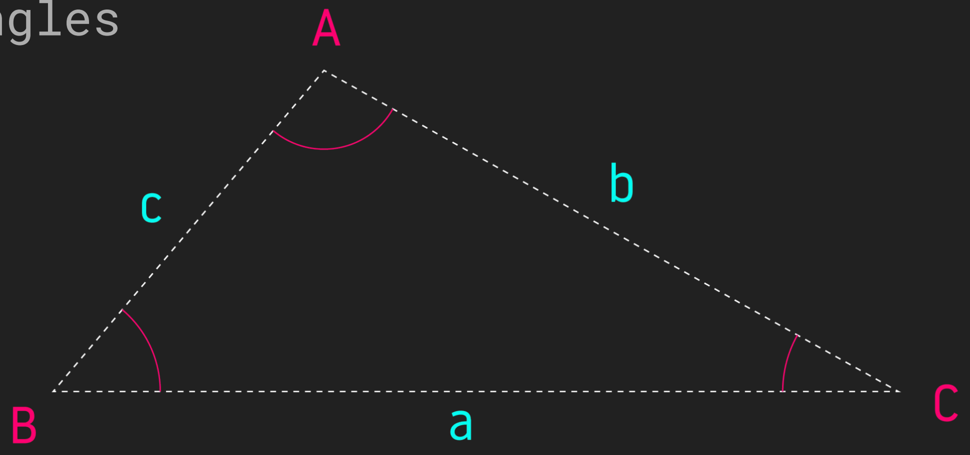
```
θ = atan2(goal.y, goal.x)
distanceToGoalXY = sqrt(goal.x² + goal.y²)
sin(Ψ) = offset / distanceToGoalXY
Ψ = asin(offset / distanceToGoalXY)
Φ = θ - Ψ
```

- Result: turn to Φ to reach θ



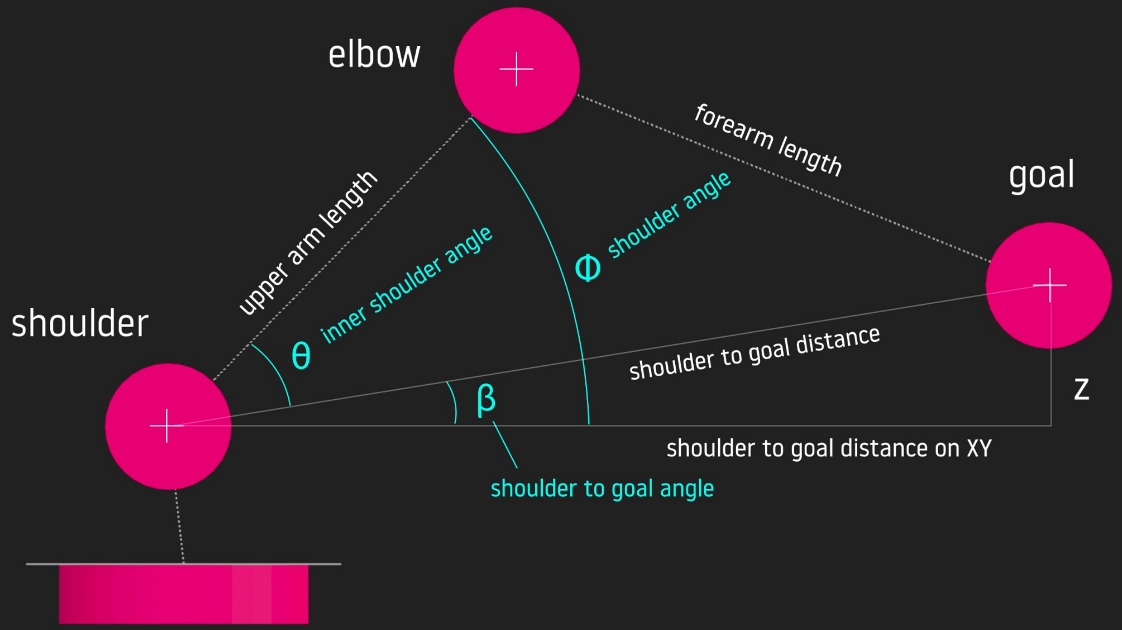
Geometric IK - Law of Cosines

- cos(A) == (b² + c² - a²) / 2bc
- cos(B) == (a² + c² - b²) / 2ac
- cos(C) == (a² + b² - c²) / 2ab
- Sides are **opposite** angles with the same name



Geometric IK - Law of Cosines

- Solving for **shoulder** angle using **law of cosines:**

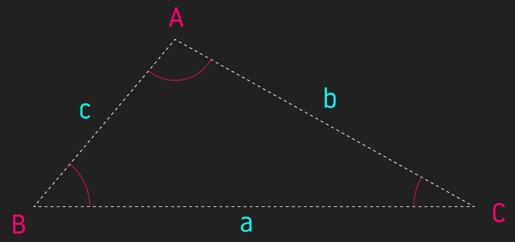


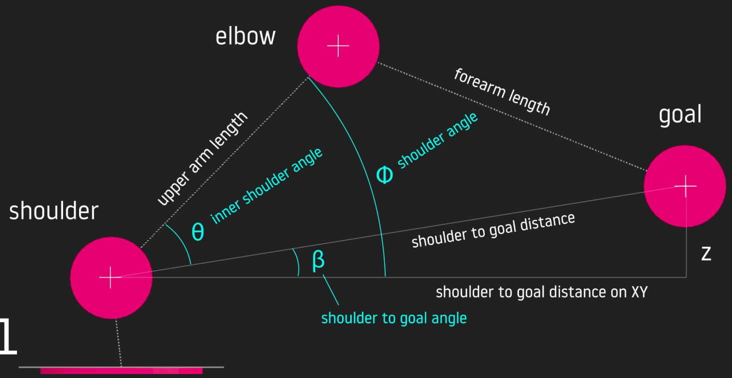
Geometric IK - Law of Cosines

- Solving for *inner* shoulder angle with law of cosines:

  $\theta == \cos(B) == (a^2 + c^2 - b^2) / 2ac$

  - B: shoulder joint angle
  - A: elbow joint angle
  - C: wrist joint angle
  - *c*: upper arm link length
  - *b*: forearm link length
  - *a*: distance shoulder to goal





Geometric IK - Law of Cosines

- Solving for shoulder to goal angle:

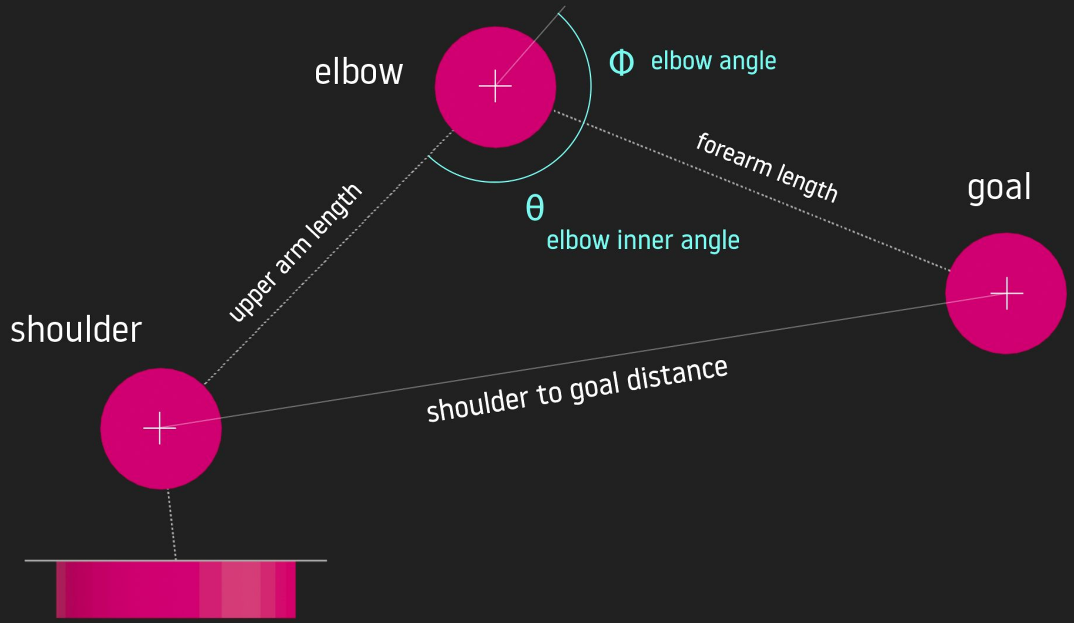
```
shoulderToGoalDistanceXY = sqrt(...)
sin(β) == goal.z / shoulderToGoalDistanceXY
β == asin(goal.z / shoulderToGoalDistanceXY)
```

- Solving for *outer* shoulder angle

```
ϕ = θ + β
```

Geometric IK - Law of Cosines

- Solving for *inner* elbow angle with law of cosines:

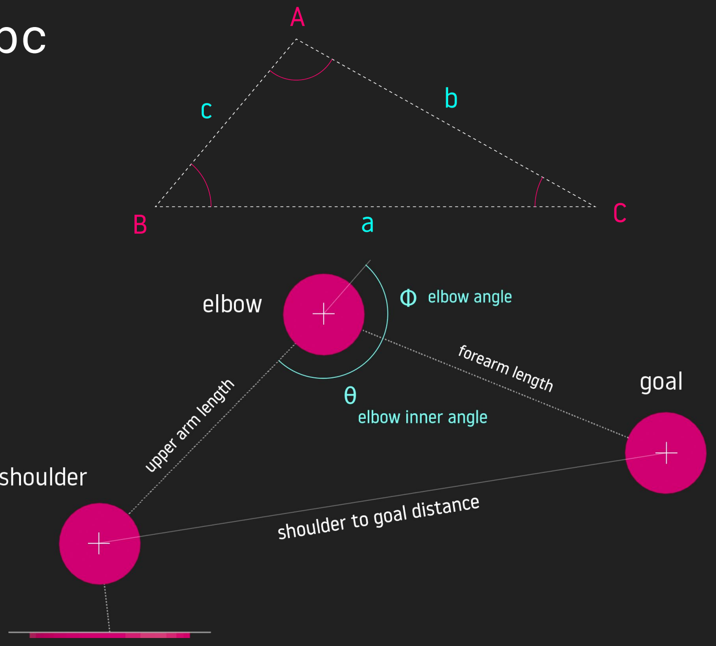


Geometric IK - Law of Cosines

- Solving for *inner* elbow angle with law of cosines:

  θ == cos(A) == (b² + c² - a²) / 2bc

  - B: shoulder joint angle
  - A: elbow joint angle
  - C: wrist joint angle
  - *c*: upper arm link length
  - *b*: forearm link length
  - *a*: distance shoulder to goal

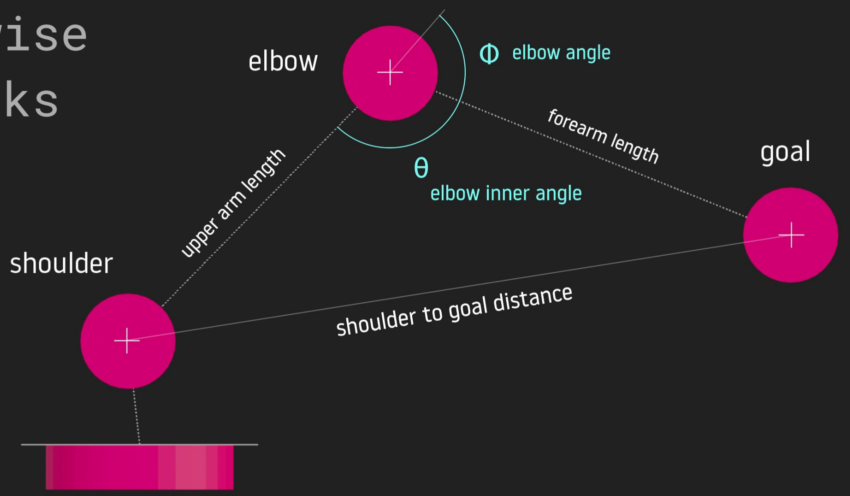


Geometric IK - Law of Cosines

- Solving for *outer* elbow angle:

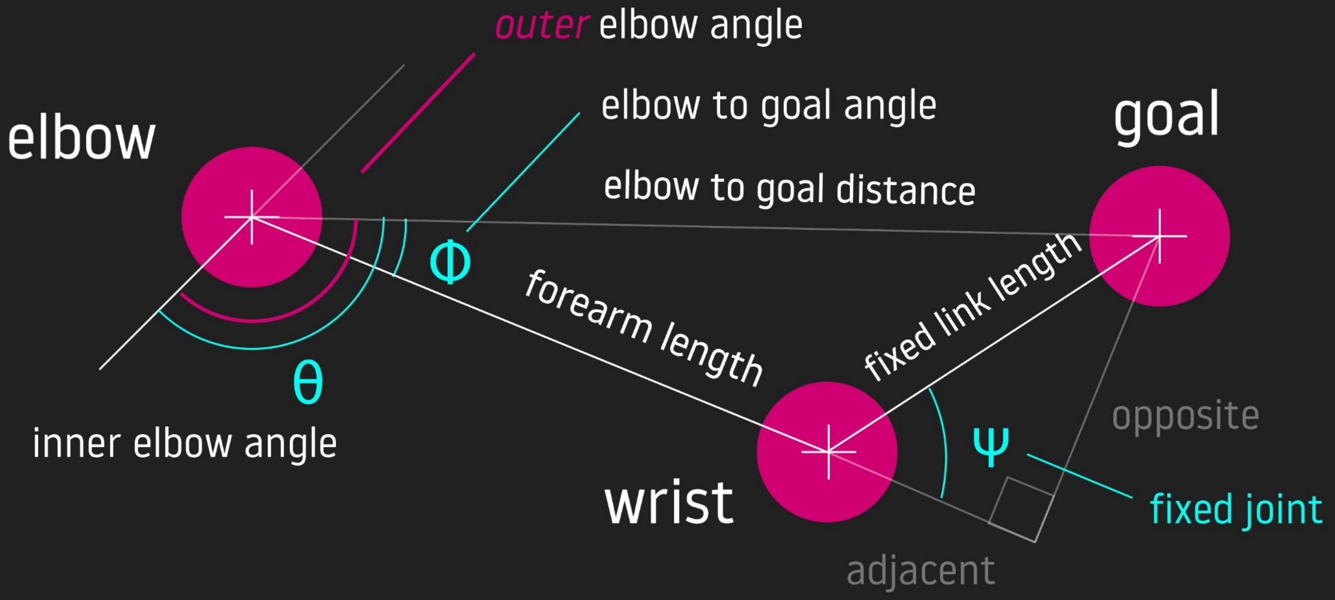
  `-(PI - θ)`

  - Angles increase counter-clockwise
  - Joint is defined clockwise
  - Both extremes align links



Geometric IK - Additional Fixed Joint

- Solving for **elbow** angle with a **fixed joint:**



Geometric IK - Additional Fixed Joint

- Solving for **elbow** angle with a **fixed joint:**
  - Solve unknown side(s) of **wrist-elbow-goal** triangle
  - Solve for **elbow to goal** angle with **arcsine** or **arctangent** once **opposite/adjacent** sides are known
  - Solve for **combined** elbow angle with **law of cosines**
  - Solve for **inner** elbow angle by subtracting **elbow to goal** angle from **combined** elbow angle
  - Solve for *outer* **elbow** angle by subtracting from PI and negating as before

Geometric IK - Additional Fixed Joint

- Solve for unknown sides of wrist-elbow-goal triangle:

```
opposite = sin(Ψ) * fixedLinkLength
adjacent = cos(Ψ) * fixedLinkLength + forearmLength
```

- Solve for elbow to goal angle:

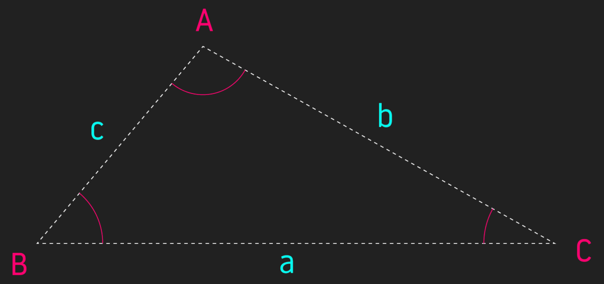
```
Φ = asin(opposite, fixedLinkLength)
or
Φ = atan2(opposite, adjacent)
```

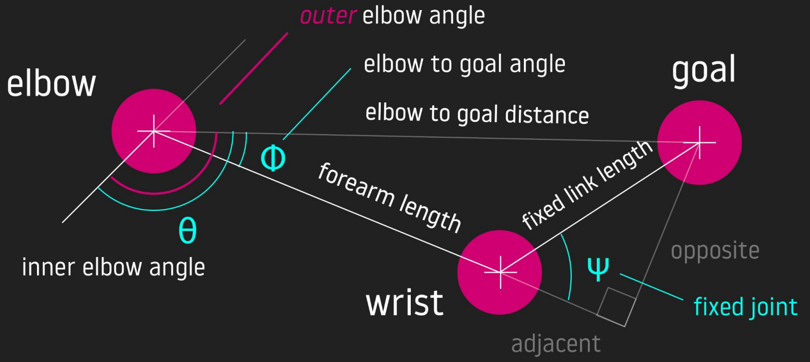
Geometric IK - Additional Fixed Joint

- Solve for **inner elbow** angle:

  $\theta == \cos(A) == (b^2 + c^2 - a^2) / 2bc$

- B: **shoulder** joint angle
- A: **elbow** joint angle
- C: **wrist** joint angle
- *c*: **upper arm** link length
- *b*: **forearm** link length
- *a*: **distance** shoulder **to goal**





Geometric IK - Additional Fixed Joint

- Solving for *outer* **elbow** angle:

```
-(PI - (Θ - Φ))
```