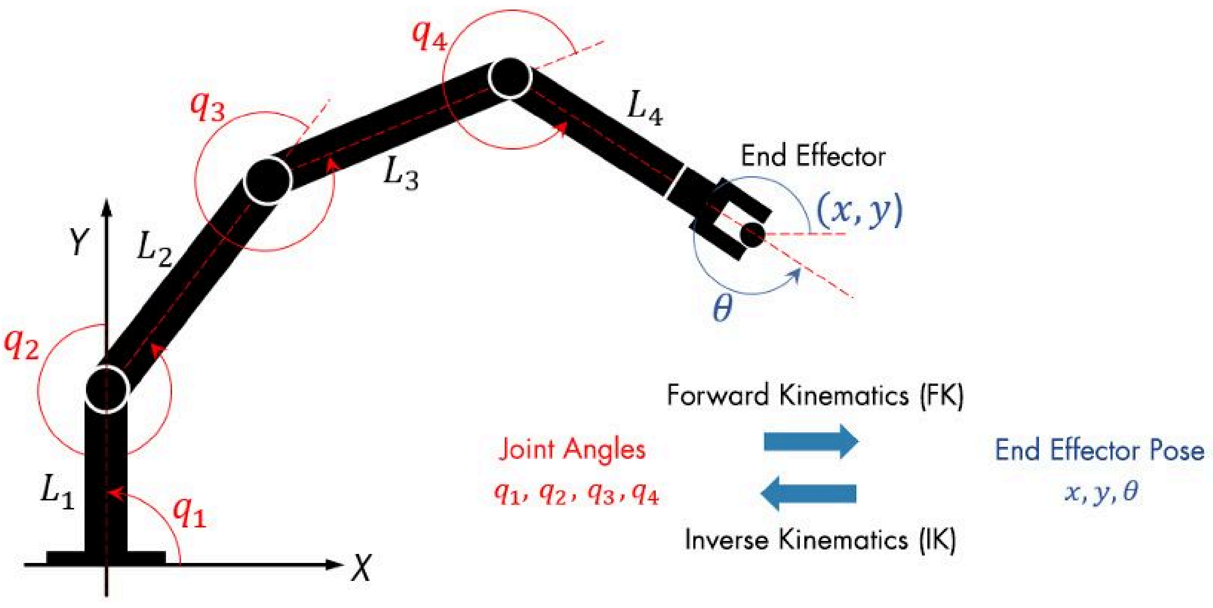
Period 1
Ian Shi
Group Name: abcdefghijklmnopqrstuvwxyz
Project Title: Inverse Kinematics and PID control

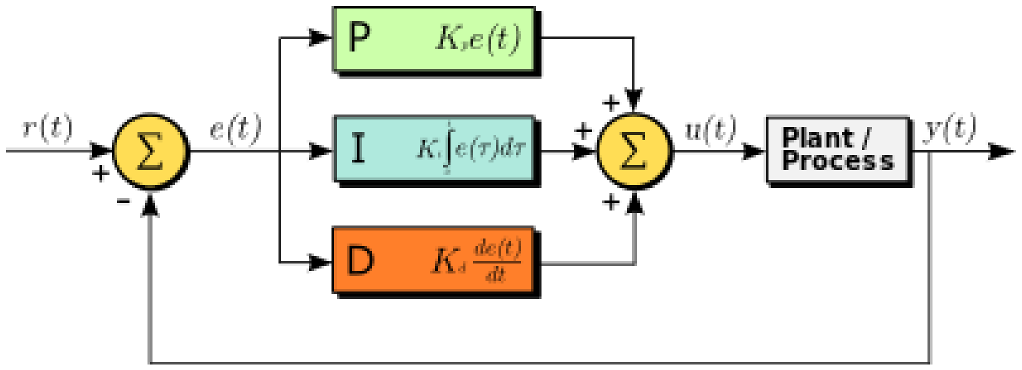
MEETING 2

Inverse kinematics is a process in robotics and computer graphics used to determine the joint parameters that provide a desired position for the end effector (the part of the robot or character that interacts with the environment, such as a hand or tool). It involves calculating the necessary angles or positions of the joints in a multi-jointed arm or limb so that the end effector reaches a specified target position and/or orientation.



The goal of this project is to create a simulation of a two or more jointed arm in processing and implement the necessary calculations to achieve the desired end effector position. The lengths of the arm segments will be configurable, and the end effector will try its best to track the position of the user's mouse on the screen.

The second goal of this project is to use a one-jointed arm to simulate a PID controller. PID control stands for Proportional-Integral-Derivative control, which is a widely used feedback control system. It combines three distinct control actions to maintain a desired output level or setpoint despite external disturbances or changes in system dynamics.



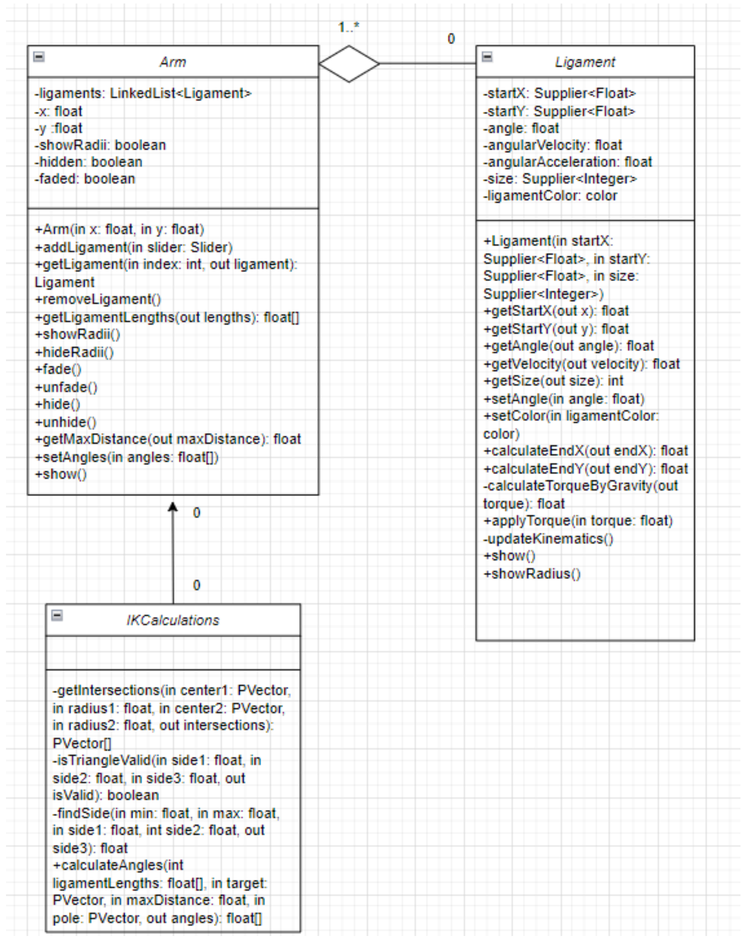
The first component is P (Proportional Control). The proportional term produces an output value that is proportional to the current error value.

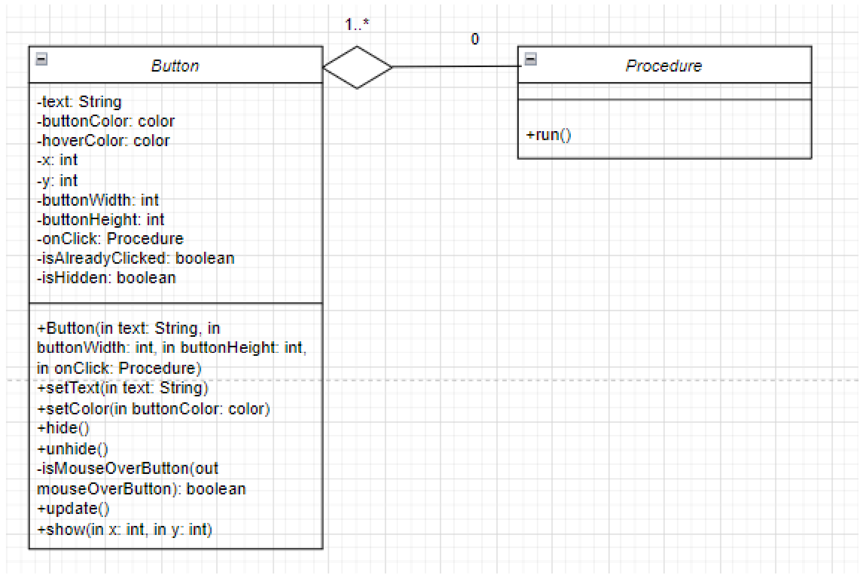
The second component is I (Integral Control). The integral term produces an output value that is proportional to the accumulation of past errors. It sums up the error over time to eliminate any residual steady-state error that the proportional term alone cannot correct.

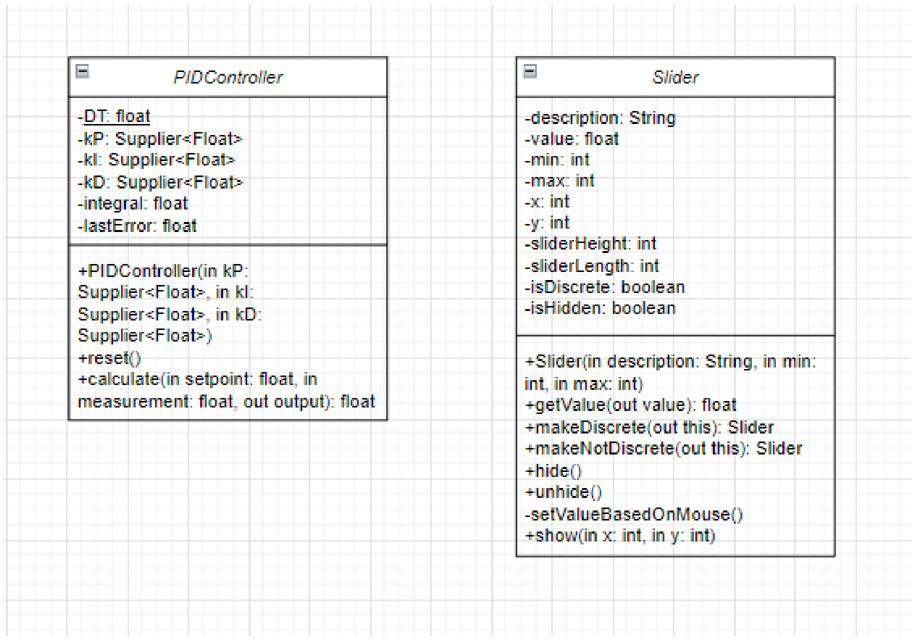
The last component is D (Derivative Control). The derivative term produces an output value that is proportional to the rate of change of the error.

The simulation will allow the user to configure the P, I, and D constants to allow the user to tune the PID controller.

Below is the UML Diagram for this project:


1..*
0
Arm
-ligaments: LinkedList<Ligament>
-x: float
-y :float
-showRadii: boolean
-hidden: boolean
-faded: boolean
+Arm(in x: float, in y: float)
+addLigament(in slider: Slider)
+getLigament(in index: int, out ligament): Ligament
+removeLigament()
+getLigamentLengths(out lengths): float[]
+showRadii()
+hideRadii()
+fade()
+unfade()
+hide()
+unhide()
+getMaxDistance(out maxDistance): float
+setAngles(in angles: float[])
+show()
Ligament
-startX: Supplier<Float>
-startY: Supplier<Float>
-angle: float
-angularVelocity: float
-angularAcceleration: float
-size: Supplier<Integer>
-ligamentColor: color
+Ligament(in startX: Supplier<Float>, in startY: Supplier<Float>, in size: Supplier<Integer>)
+getStartX(out x): float
+getStartY(out y): float
+getAngle(out angle): float
+getVelocity(out velocity): float
+getSize(out size): int
+setAngle(in angle: float)
+setColor(in ligamentColor: color)
+calculateEndX(out endX): float
+calculateEndY(out endY): float
-calculateTorqueByGravity(out torque): float
+applyTorque(in torque: float)
-updateKinematics()
+show()
+showRadius()
0
0
IKCalculations
-getIntersections(in center1: PVector, in radius1: float, in center2: PVector, in radius2: float, out intersections): PVector[]
-isTriangleValid(in side1: float, in side2: float, in side3: float, out isValid): boolean
-findSide(in min: float, in max: float, in side1: float, int side2: float, out side3): float
+calculateAngles(int ligamentLengths: float[], in target: PVector, in maxDistance: float, in pole: PVector, out angles): float[]


1..*
0
Button
-text: String
-buttonColor: color
-hoverColor: color
-x: int
-y: int
-buttonWidth: int
-buttonHeight: int
-onClick: Procedure
-isAlreadyClicked: boolean
-isHidden: boolean
+Button(in text: String, in buttonWidth: int, in buttonHeight: int, in onClick: Procedure)
+setText(in text: String)
+setColor(in buttonColor: color)
+hide()
+unhide()
-isMouseOverButton(out mouseOverButton): boolean
+update()
+show(in x: int, in y: int)
Procedure
+run()


PIDController
-DT: float
-kP: Supplier<Float>
-kI: Supplier<Float>
-kD: Supplier<Float>
-integral: float
-lastError: float
+PIDController(in kP: Supplier<Float>, in kI: Supplier<Float>, in kD: Supplier<Float>)
+reset()
+calculate(in setpoint: float, in measurement: float, out output): float
Slider
-description: String
-value: float
-min: int
-max: int
-x: int
-y: int
-sliderHeight: int
-sliderLength: int
-isDiscrete: boolean
-isHidden: boolean
+Slider(in description: String, in min: int, in max: int)
+getValue(out value): float
+makeDiscrete(out this): Slider
+makeNotDiscrete(out this): Slider
+hide()
+unhide()
-setValueBasedOnMouse()
+show(in x: int, in y: int)

How does it work?

Inverse Kinematics:

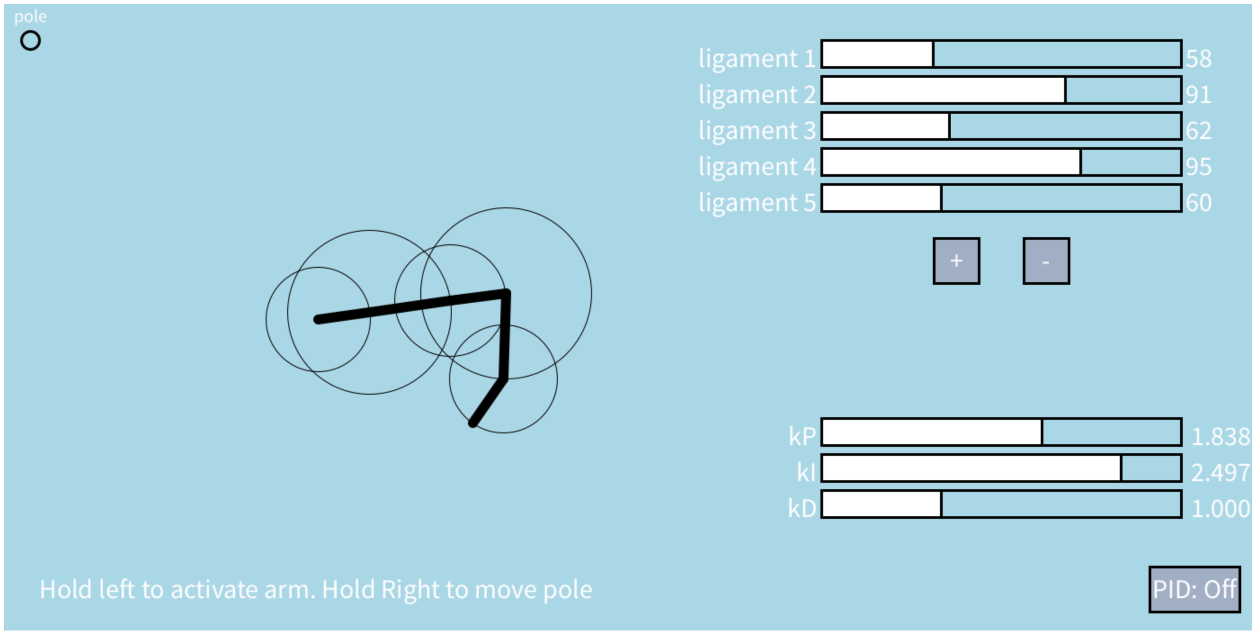
As the user moves their mouse across the screen while holding left click, the end effector of the arm will try its best to get to where their mouse is. The user can change the length of each of the ligaments by sliding the sliders on the right side of the screen, and increase or decrease the amount of ligaments the arm has.

If the user is holding right click, they can change the location of the pole, which will cause the arm to behave differently at certain points.

PID:

Similar to Inverse Kinematics, the arm will try its best to reach a certain setpoint. However, there will be two arms. One arm represents where the arm wants to go. This arm will behave the same as the arm from the IK simulation. However, the second arm represents PID control trying its best to get to the location of the first arm. This arm will be controlled with a PID controller with tuneable constants. The user can tune PID constants with sliders to attempt to give the arm the best characteristics to reach its setpoint as it fights gravity. The user is also able to configure the length of the arm to see how the length affects the required PID constants.

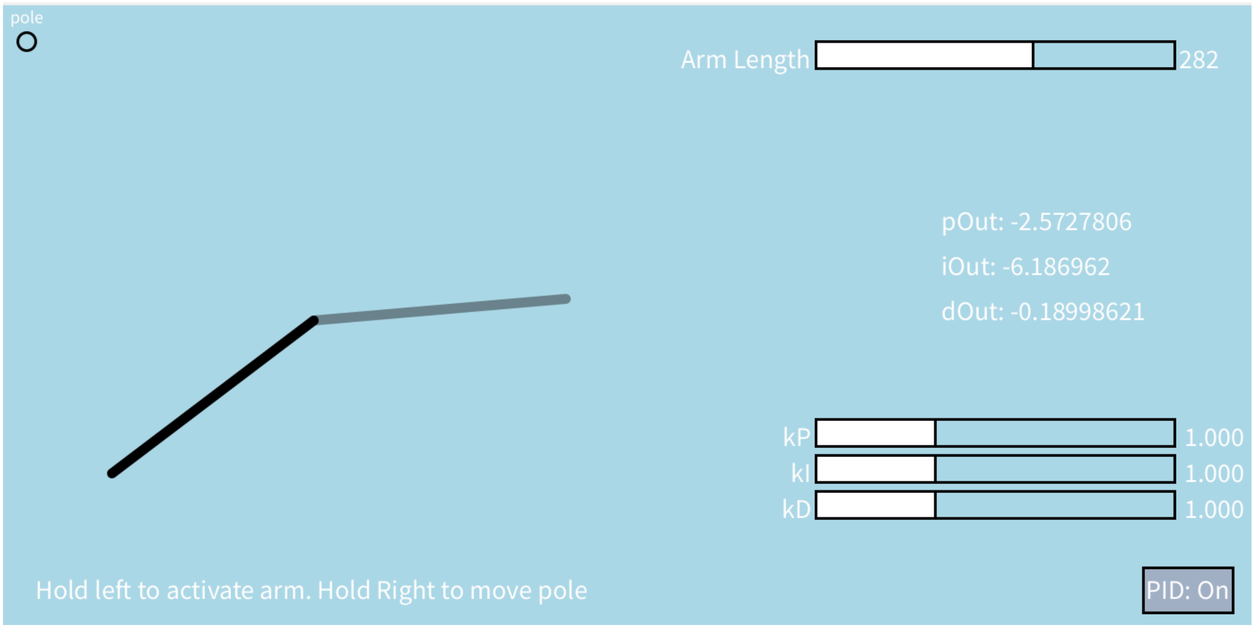
Unlike the arm from the IK simulation, however, this arm only has one ligament.



ligament 1 58
ligament 2 91
ligament 3 62
ligament 4 95
ligament 5 60

kP 1.838
kI 2.497
kD 1.000

Hold left to activate arm. Hold Right to move pole



Arm Length 282

pOut: -2.5727806
iOut: -6.186962
dOut: -0.18998621

kP 1.000
kI 1.000
kD 1.000

Hold left to activate arm. Hold Right to move pole

Functionalities/Issues

IK Simulation
I’ve successfully implemented the calculations required to reach the desired angles to allow the end effector of the arm to achieve the desired position. I’ve also created sliders that make the length of the ligaments configurable, and buttons that allow the user to add or remove the total ligaments in the arm and switch between the PID simulation and the IK simulation.

When the user holds left click, the arm tries to track the user’s mouse. When the user holds right click, the user can move the pole around.

The pole is an important part of the calculations required to reach the desired joint configurations. Without getting too deep into the math, the position of the pole alters the behavior of the arm at certain points.

(the following issues are the same as the last meeting. I still havent figured out why these issues occur)

One issue is that the arm is jittery, I’m not sure if this is how it's supposed to behave (maybe the desired joint configurations really do change that fast…) Unfortunately, this could be a side effect of the simpler method that I used to calculate desired angles. Instead of using linear algebra, I used only vector math and circles.

Another issue is this logic right here:

```
// not sure why this works lol
private static boolean isTriangleValid(float side1, float side2, float side3) {
  return (side1 + side2) >= side3 || (side1 + side3) >= side2 || (side2 + side3) >= side1;
}
```

For a triangle to be valid, all three of the conditions must be satisfied. For some reason, the simulation works when any of the three conditions are met. I plan to examine why this is.

PID Simulation
I’ve created a PID controller that takes in a measurement and a setpoint and returns a desired output. Using the output of the PID controller, I apply a “torque” to the ligament that causes it to gain an angular acceleration in a certain direction.

Speaking of which, I’ve allowed the ligaments to be controlled using torque analysis and rotational kinematics. At any given moment in time, two forces create a torque on the ligament: the force of gravity and the force created by the PID controller (imagine the output is powering a motor at the joint).

The user can control the setpoint (where the arm wants to go) by dragging their mouse around and holding left click (similar to the IK simulation). The “actual” arm will try to follow the setpoint by using the PID constants that the user provides. These constants can be changed by sliding a slider at the right of the screen.

Unfortunately, as of right now, my integral output keeps blowing up out of proportion, even when my kI constant is really small. Because of this, the arm keeps spinning super fast and never actually gets to the setpoint, unfortunately. I believe the issue is the angle math that I use to calculate the error, but it might also be because of how I calculated and apply gravity. I’ll have to look into why.

Other
I’ve created buttons that allow the user to add or remove the total ligaments in the arm for the IK simulation, and switch between the PID simulation and the IK simulation.

Plans for the next meeting:

- Look into why the arm for PID is swinging wildly
    - Is it how I calculated gravity?
    - Is it how I calculate the angles and the error?
    - Is it how I calculate iOut?
- Make it so that the target doesnt change when I’m clicking on the sliders and buttons