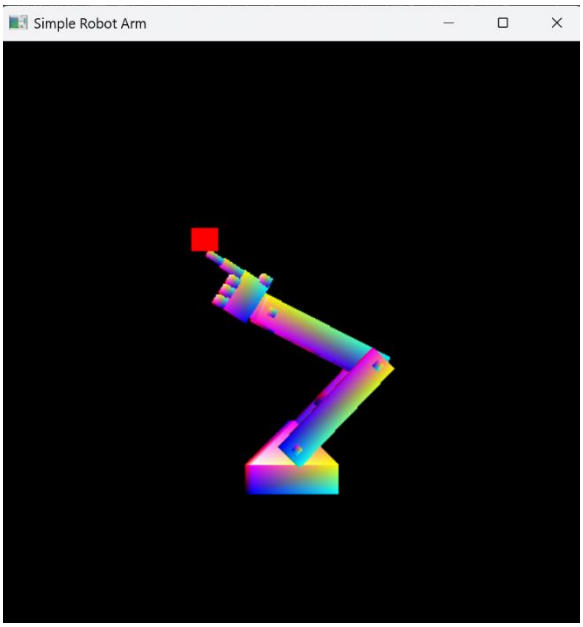
Homework: A Simple Robot Arm에 관한 보고

21011794 황재동(최종수정 2025.11.08)

개요:

우리는 Forward Kinematics(FK)을 중점적으로 배웠고 동시에 Inverse Kinematics(IK)와의 차이도 배웠다. FK는 쉽게 말해 관절의 각도를 통해 끝 점을 계산할 때 사용할 수 있고 반면 IK는 정해진 위치에 끝점을 두기 위해 관절 각도를 계산하는 용도로 사용할 수 있다. 우리 과제에서는 Robot Arm을 주어진 각도로 특정 점으로 이동시키는 것이 아닌 특정 점에 Robot Arm을 이동시키기 위한 각도를 역으로 구해야 하므로 IK를 중점적으로 사용할 것이다. Gradient Descent(경사 하강법) 알고리즘을 사용하여 Hand 끝 부분과 목표 지점(빨간색 점) 사이의 오차를 줄여가면서 각도를 업데이트하고 이로 구해진 각도를 Robot Arm에 적용한다. Gradient Descent에 대한 자세한 설명은 아래서 언급한다.



구현 방법 구상

우리가 궁극적으로 수행해야 하는 것은 빨간색 점과 Hand의 특정 지점(빨간 점을 따라갔으면 하는 Hand 부위의 좌표) 사이의 거리 줄이기다. 빨간색 점의 위치를 반환하는 것은 내부 구현이 되어 있어 따로 구현은 불필요하고 Hand의 좌표는 직접 구해야 한다. Default Robot Arm을 만들기 위해 CTM을 이용했는데 이 CTM을 이용하면 Hand의 좌표를 쉽게 구할 수 있을 것이다. 만약 Hand의 관절이 아닌 특정 지점의 좌표가 필요하다면 CTM에서 조금만 Translate해주면 쉽게 구할 수 있을 것이다.

Hand의 좌표를 구했다면 가장 쉽게 ang1, ang2, ang3의 적정 값을 구하는 방법은 당연히 3중 for문을 돌려 최적의 값을 구하는 것이다. 가장 내부에 있는 for 문에서 Hand와 빨간색 점 사이의 오차를 계산하고 오차가 가장 적은 각도 값을 ang1, ang2, ang3에 넣으면 된다. 이 방법을 응용하여 허용 오차보다 현재 오차가 같거나 작을 때 바로 반복문을 탈출하게 할 수도 있을 것 같다. 이러면 반복문을 전부 돌지 않아 계산 횟수가 줄어든다.

다음은 앞선 방법 보다는 복잡할 수는 있으나 시간 복잡도를 매우 극한으로 낮출 수 있는 Gradient Descent 알고리즘을 사용하는 것이다. 이는 한국말로 경사 하강법으로 말 그대로 U자 협곡에서 공을 굴리면 협곡에 가장 낮은 지점에 도달(수렴)하는 것과 유사하다. 단 몇번의 계산만으로 Robot Arm의 Hand가 우리가 원하는 협곡의 가장 낮은 지점(빨간색 점)에 도달하게 할 수 있을 것이다.

구현 방법 1(3중 for문을 통한 최적 각도 알아내기)

3중 for문을 사용해서 각도를 구하는 것은 IK보다는 FK에 가까운 느낌이다. 왜냐하면 각도를 정말 역으로 구하는 것이 아닌 각도를 0도부터 360도까지 하나하나 다 돌려가면서 적정 값을 찾기 때문이다. 우선 가장 적었던 오차를 저장할 minDistance 변수와 그때의 각도를 임시 저장할 minAng1, minAng2, minAng3를 사용했다. for문은 처음에는 1도씩 증가하는 방식을 사용했고 가장 내부 for문에서는 현재의 각도 값으로 얻어지는 Hand의 좌표와 빨간색 점 사이의 오차를 계산했다. 그후 오차가 가장 적었던 각도를 원래의 각도에 저장했다.

Hand와 빨간색 점 사이의 각도를 구하는 것은 자주 쓰일 것 같아 함수로 따로 구현했다. 이 방식을 사용하니 복잡도가 높아서 그런지 프로그램이 계속 반복문을 돌다가 터졌다. 그래서 반복문에서 각도를 늘리는 양을 조금씩 늘려봤는데 5~10도씩 늘렸을 때 그제서야 터지지 않았지만 매우 느리게 작동했다. 당연히 큰 각도씩 늘렸기 때문에 그 사이의 값은 확인을 못해 그렇게 정교하지 못했다. 또한 허용오차 범위를 따로 정해서 그 이하로 떨어지는 즉시 반복문을 탈출하게 했다. 여전히 만족스러운 결과를 얻지는 못했다.

구현 방법1의 장점 및 단점

해당 구현 방식은 사실 반복문만 돌면 언젠가는 가장 근접한 각도를 내놓기 때문에 매우 간단하게 구현할 수 있다는 장점이 존재하나 그것이 정말 최적의 해가 아니라는 단점이 동시에 존재한다. 시간 복잡도 또한 매우 높다 1도씩 3개의 관절만 탐색해도 4600만회가 넘어가고 더 정밀한 탐색을 원한다면 더욱 복잡해질 것이다. 우리는 초당 60프레임을 소화하는 것이 목표인데 사실상 불가능한 방법이라고 생각한다.

구현 방법 2(Gradient Descent 사용하여 최적 각도로 수렴하기)

앞에서 겪었던 매우 높았던 시간 복잡도를 극적으로 해결할 수 있는 방식이다. 앞서 언급했듯이 U자형 협곡에서 공을 굴려 가장 낮은 지점(과제 기준 Hand와 빨간색 점 사이 오차가 0에 매우 근접)에 수렴하는 방식과 같은 알고리즘이다. 실상은 함수에서 현재 값에 대한 기울기를 구하고 현재 값에서 얻어진 기울기를 빼는 방식이다. 이러면 협곡의 최하단 지점에 갈수록 기울기가 점점 0에 수렴할 것이고 더 이상 값에서 기울기를 빼도 의미가 없어진다.

이런 의미 없는 계산을 방지하기 위해 값에서 기울기를 몇번 뺄것인지 정해줘야 한다(반복문의 반복 횟수). 필자는 우선 for문의 첫 줄에 현재 각도를 기준으로 한 Hand의 위치를 구하고 그 지점에서 바라본 빨간색 점의 좌표와의 거리(오차)를 구했다. 이후 각각의 각도(ang1, ang2, ang3)가 변화했을 때 순간 변화율 즉, 기울기를 구하기 위해 매우 작은 크기의 각도를 더해서 더하기 전의 손 위치와 더한 후의 위치의 차를 구해서 더한 각도(라디안)만큼 나눠줬다. 이는 각을 더 했을 때 손의 위치 변화량을 알려준다.

사실 직전에 구한 위치 변화량은 그것만으로는 도움이 되지 않는다 왜냐하면 특정 각도만큼 증가했을 때 손위치의 변화만 보여주기 때문이다. 우리는 이 움직임이 오차를 줄이는지를 구해야 한다. 따라서 전에 구한 위치 변화량과 오차를 내적해야 한다. 그럼 최적의 방향(부호)과 크기(스칼라)를 알 수 있다. 이제 내적으로 구해진 라디안을 각도로 변환하여 현재 각도 에다가 빼고 반복문 탈출하고 구해진 최종 각도를 ang1, ang2, ang3에 반영한다. 후에 계속 디버깅 해보면서 반복 횟수와 Learning Rate의 값을 바꿔가면서 최적이라고 생각한 값으로 고정했다. Learning Rate에 대한 설명은 다음 Gradient Descent의 장단점에서 설명한다.

구현 방법2의 장점 및 단점

Gradient Descent의 장점은 당연 낮아진 시간 복잡도와 매우 높아지는 정교함이다. 이렇기 때문에 현재는 관절이 3개밖에 없지만 각도가 무수히 많아져도 시간 복잡도의 걱정없이 사용할 수 있다. 만약 관절이 무수히 많아짐에도 불구하고 3중 for문을 사용한다면 아예 작동도 안할 것이다.

이런 장점만 있어 보이는 경사 하강법에도 단점은 존재한다. 우선 U자 협곡이 여러 개(고차함수)일 때 우리가 찾은 최하점이 전역의 최하점이 아닌 로컬의 최하점일 수 있다는 한계가 있다. 그렇기 때문에 값을 그냥 차이를 구하는 것이 아닌 Learning Rate를 곱해서 협곡을 하강하는 보폭을 조절해야한다. 이 값이 너무 커버리면 값을 지나쳐서 설정한 반복 횟수 이내에 원하는 지점에 수렴

을 못하거나 너무 작아져 버려도 조금씩 하강하여 수렴하는데 매우 오래걸린다. 그래서 여러 차례 프로그램 실행 후 적절한 Learning Rate값을 찾아야 한다. 동시에 적정 반복 횟수도 정한다.

추가 : 디자인적 요소

과제가 특정 지점을 따라가는 것이라 Robot Arm의 끝부분이 실제로 그 부분을 가리키면 어떨까 생각을 해봤다. 어떤 것을 가리킬 때는 손가락만큼 정확한 것이 없다고 생각을 했고 조금은 복잡하지만 사람의 손모양같이 디자인해보려고 했다. 손을 구현할 때 처음에는 하드코딩으로 했고 그 이후에는 겹치는 부분을 stack을 사용하여 간소화했다.

Hand 부분 외에도 Upper Arm와 Lower Arm을 가운데 보강대를 넣어 H 모양으로 디자인하여 튼튼한 느낌을 주려고 해봤다. 덕분에 판의 크기가 얇음에도 불구하고 중간에 있는 보강대로 인해 견고한 느낌을 준다. 추가적으로 관절에 심을 하나씩 추가하여 기계적인 느낌도 줬다.

시행착오 및 해결 과정

빨간색 점의 위치

MyTarget.h의 GetPosition()함수를 사용해서 빨간색 점의 좌표를 구하고 이 좌표와 Hand 사이 거리를 구하고 출력했을 때 화면으로 보이는 거리와 출력된 거리가 일치하지 않아 보이는 문제가 있었다. 해결하기 위해 각각의 좌표를 직접 출력해보니 Hand의 좌표는 이상이 없었고 빨간색 점의 좌표가 화면상의 원점과 Y축의 좌표가 달랐다. MyTarget.h의 코드를 보니 0.2의 offset이 있었고 이 때문에 값이 약간 달랐다고 판단하여 main.cpp에서 0.4만큼 Y축 좌표에서 빼주니 값이 정상적으로 나왔다.

단위 라디안과 각도

가장 처음에 알고리즘을 완성했을 때 작동이 이상하게 되는 가장 근본적인 원인이었다. Hand의 위치 변화율을 구할 때 분모에 라디안이 아닌 각도를 사용했고 이 때문에 값이 엄청 튀는 문제가 발생했다. 라디안이랑 각도는 차이가 엄청나게 크기 때문이다. 분모를 라디안으로 변환하니 전처럼 값이 튀는 문제가 사라지고 안정화됐다.

learningRate와 maxIterations값 설정

의외로 가장 단순하지만 오래 걸렸던 과정이다. 이 두 파라미터는 속도와 안정성을 결정하며 서로 상충 관계에 있다는 결론을 얻었다. 처음에 learningRate를 1.5로 설정했을 때 Robot Arm이 목표 지점을 정확히 찾지 못하고 빨간색 점 주변에서 약간 떠는 현상이 있었다. 반대로 0.1로 현저히 낮추니까 정해진 반복문 횟수 내에 목표 지점에 도달하지 못하고 뭔가 뒤따라가는 느낌을 주었다.

그렇게 여러시도를 한 결과 learningRate의 타협점을 0.5로 정했고 반복문의 반복 횟수를 20으로 정했다. 결과적으로 learningRate를 낮춰 안정성을 확보했고 안정성으로 낮아진 속도를 보완하기 위해 maxIterations를 20으로 늘려 균형점을 찾았다.

전체적으로 Arm이 약간 늦게 따라가 서로 뚫고 가는 문제

앞서 파라미터들을 정했음에도 불구하고 Robot Arm이 가끔 본체를 뚫고 지나가 어색해 보이는 문제가 생겼다. 이때 자세히 관찰해보니 첫번째 관절이 뭔가 늦게 따라가는 듯한 양상을 보였다. 그래서 떠올린 생각이 "ang1의 기울기에 가중치를 주면 어떨까?"였다. ang1에 가중치 4를 곱하여 더 민감하게 반응하도록 하여 팔이 전체가 더 빠르게 목표를 추적할 수 있게 했다.