CINEMÁTICA INVERSA COMO PROBLEMA DE OTIMIZAÇÃO: IMPLEMENTAÇÃO VIA GRADIENTE DESCENDENTE E ESTRATÉGIAS HÍBRIDAS

Marcos Vinícius da Silva[1]

RESUMO

A cinemática inversa (inverse kinematics, ou IK) é um problema fundamental na engenharia da computação e robótica, consistindo na determinação dos parâmetros articulares necessários para posicionar um efetuador final em um alvo desejado. Este trabalho propõe uma abordagem baseada em algoritmos de otimização para resolver o problema de IK, afastando-se das soluções analíticas geométricas em favor de métodos iterativos baseados em cálculo. O objetivo principal é demonstrar como o algoritmo de gradiente descendente (gradient descent), fundamentado no cálculo de derivadas parciais através da matriz jacobiana, pode ser aplicado para minimizar o erro de posição de um manipulador robótico. A metodologia compreende a formulação matemática da função de custo, a derivação do método do jacobiano transposto e a implementação de um simulador de braço robótico 2D utilizando a linguagem Rust e o motor gráfico Bevy. Os resultados experimentais mostram que a abordagem de otimização é eficaz, permitindo que o manipulador converja suavemente para alvos dentro de seu alcance e mantenha um comportamento estável em situações de singularidade ou alvos inalcançáveis, validando a aplicação prática dos conceitos teóricos de otimização.

Palavras-chave: Otimização de algoritmos. Cinemática inversa. Cálculo. Gradiente descendente. Simulação robótica.

INTRODUÇÃO

A otimização de algoritmos é um pilar fundamental da engenharia da computação e da matemática aplicada, consistindo na seleção da melhor solução dentro de um conjunto de alternativas disponíveis (YANG, 2011). Esta área serve como a força motriz por trás de avanços significativos, desde a alocação eficiente de recursos até o treinamento de modelos complexos de inteligência artificial, onde algoritmos como o gradiente descendente são amplamente utilizados (RUDER, 2016).

Neste contexto, surge o problema da cinemática inversa (IK, do inglês *inverse kinematics*). Enquanto a cinemática direta (FK, do inglês *forward kinematics*) trata do cálculo da posição final de um sistema articulado dados os ângulos de suas juntas, a cinemática inversa

[1] Graduando em Engenharia da Computação pela UEMG Divinópolis. E-mail: marcos.1695704@discente.uemg.br

apresenta o desafio oposto: determinar quais devem ser os ângulos das juntas para alcançar uma posição-alvo desejada (BUSS, 2009). Resolver a cinemática inversa analiticamente é muitas vezes computacionalmente custoso ou matematicamente complexo, especialmente para sistemas com muitos graus de liberdade (BUSS, 2009; ROKBANI; ALIMI, 2013).

A abordagem moderna, portanto, é reformular o IK, não como um problema puramente geométrico, mas como um problema de otimização (VOSS; KOPP, 2025). O objetivo passa a ser encontrar o conjunto de ângulos que minimiza uma função de custo, definida tipicamente como a distância euclidiana entre a posição atual do efetuador final e o alvo (ROKBANI; ALIMI, 2013). Esta conexão abre a porta para a aplicação do cálculo, especificamente através do algoritmo de gradiente descendente (*gradient descent*), que ajusta iterativamente os parâmetros para minimizar o erro (ALI; AHMED, 2021; RUDER, 2016).

Neste contexto, o presente trabalho se propõe a demonstrar a eficácia do algoritmo de gradiente descendente na solução do problema de cinemática inversa para um sistema articulado. O objetivo principal é ilustrar a formulação matemática da função de custo e a aplicação do cálculo (especificamente através do jacobiano transposto) como um método robusto para minimizar o erro de posição do efetuador final. Para tanto, este artigo detalhará a fundamentação teórica de cinemática direta e inversa, apresentará a formulação do IK como um problema de otimização e a derivação do gradiente descendente, descreverá a metodologia e os resultados da simulação, e, por fim, sumarizará as conclusões do estudo.

REVISÃO DE LITERATURA

O desenvolvimento de controladores cinemáticos eficientes depende intrinsecamente da aplicação de conceitos avançados de otimização e cálculo diferencial (YANG, 2011). Ao reformular a movimentação de sistemas articulados como um problema de minimização de funções de custo, supera-se as limitações das soluções analíticas tradicionais, permitindo uma resposta dinâmica e robusta em ambientes virtuais (ROKBANI; ALIMI, 2013; VOSS; KOPP, 2025). A base para tal abordagem encontra-se na análise rigorosa de algoritmos iterativos, como o gradiente descendente (RUDER, 2016; ALI; AHMED, 2021), e na utilização da matriz jacobiana como elo fundamental entre o espaço das juntas e o espaço cartesiano (BUSS, 2009).

Otimização e funções de custo

A otimização é um campo central da matemática aplicada e da ciência da computação, preocupada com a eficiência na resolução de problemas complexos. Segundo Yang (2011), a

escolha do algoritmo de otimização correto é crucial, pois a eficiência computacional e a precisão da solução dependem diretamente da adequação do método ao problema.

Em essência, um problema de otimização consiste em selecionar o "melhor elemento" de um conjunto de alternativas disponíveis, minimizando ou maximizando um critério específico. Este critério é matematicamente definido como uma função de custo (ou função objetivo), que mapeia os parâmetros do problema para um número real representando o "erro" ou a "qualidade" da solução. Em problemas de engenharia contínua, busca-se frequentemente minimizar essa função.

Ali e Ahmed (2021) destacam que, em contextos como regressão linear ou treinamento de redes neurais, a função de custo mais comum é o erro quadrático (SSE, do inglês *sum of squared errors*). Esta função é particularmente útil porque é diferenciável e convexa em muitos contextos, penalizando desvios grandes de forma mais severa do que pequenos desvios, o que guia o otimizador suavemente em direção ao alvo.

O Algoritmo de gradiente descendente

Para minimizar funções de custo diferenciáveis, o algoritmo de gradiente descendente é uma das técnicas mais populares e fundamentais. Ruder (2016) explica que, embora existam variações complexas (como o *stochastic gradient descent* ou métodos com *momentum*), a versão clássica do algoritmo permanece como a base intuitiva para a otimização: para encontrar o ponto mais baixo em uma superfície (o mínimo da função de custo), deve-se dar passos iterativos na direção da descida mais íngreme.

O cálculo diferencial fornece a ferramenta matemática para determinar essa direção: o gradiente (∇). O gradiente de uma função multivariável é um vetor que contém todas as suas derivadas parciais ($\partial C/\partial \theta_i$). Uma propriedade fundamental do gradiente é que ele aponta na direção de maior crescimento da função. Consequentemente, o vetor oposto, o gradiente negativo ($-\nabla$), aponta na direção de maior decrescimento.

A regra de atualização iterativa do algoritmo, conforme detalhada por Ali e Ahmed (2021), é dada por:

$$\theta_{novo} = \theta_{antigo} - \alpha \cdot \nabla C(\theta)$$

Onde θ representa o vetor de parâmetros a ser otimizado e α é a taxa de aprendizado. Ruder (2016) alerta que a escolha de α é crítica: se for muito pequeno, a convergência é lenta e computacionalmente cara; se for muito grande, o algoritmo pode oscilar ou divergir, falhando em encontrar o mínimo.

Cinemática inversa como otimização

Tradicionalmente, a Cinemática Inversa (IK) era resolvida por métodos analíticos geométricos, que tentam isolar os ângulos das juntas em equações fechadas. No entanto, Rokbani e Alimi (2013) argumentam que, para sistemas articulados complexos ou redundantes, métodos de otimização numérica baseados em inteligência computacional (como gradiente) oferecem maior flexibilidade. Nesta abordagem, o problema deixa de ser "resolver a geometria" e torna-se "minimizar o erro".

Voss e Kopp (2025) reforçam essa visão moderna, definindo o IK como a minimização de uma função de erro J(θ) através de métodos numéricos, sendo essencial para a geração de movimentos plausíveis em tempo real para personagens virtuais. A função de custo C(θ) é tipicamente definida como a distância euclidiana ao quadrado entre a posição atual do efetuador s(θ) e a posição do alvo t:

$$C(\theta) = ||t - s(\theta)||^2$$

Esta formulação permite lidar não apenas com o objetivo de alcançar um ponto, mas também facilita a inclusão de restrições adicionais (como limites de juntas ou evasão de obstáculos) simplesmente adicionando novos termos de penalidade à função de custo (ROKBANI; ALIMI, 2013).

O método do jacobiano transposto

A aplicação do gradiente descendente à robótica exige o cálculo de como a posição do efetuador muda em relação aos ângulos das juntas. Buss (2009) introduz a matriz jacobiana (J) como a ferramenta central para essa linearização. O jacobiano é a matriz de derivadas parciais que relaciona as velocidades das juntas ($\dot{\theta}$) às velocidades do efetuador final ($\dot{s}$) no espaço cartesiano:

$$\dot{s} = J\dot{\theta}$$

Enquanto métodos clássicos exigiriam o cálculo da inversa do Jacobiano (J^{-1}), Buss (2009) observa que a inversão de matrizes é computacionalmente custosa e sofre de instabilidade numérica próxima a singularidades. Como alternativa, Pisculli et al. (2014) demonstram a eficácia da estratégia baseada na transposta do jacobiano. Em vez de inverter a matriz, utiliza-se a sua transposta (J^T) para mapear o vetor de erro cartesiano de volta para o espaço das juntas.

Embora robusto e livre das instabilidades de inversão, o método da transposta pode apresentar convergência lenta ou oscilações se não for devidamente controlado. Buss (2009)

ressalta que, para garantir estabilidade e evitar comportamentos erráticos em configurações singulares, é frequentemente necessário aplicar técnicas de amortecimento (*damping*) ou limitação do passo de atualização, estratégias essenciais para a viabilidade de simulações em tempo real.

METODOLOGIA DA PESQUISA

Este trabalho adota uma metodologia de pesquisa aplicada com desenvolvimento experimental, visando a validação de algoritmos de otimização em um ambiente de simulação controlada. O percurso metodológico foi dividido em três etapas: modelagem matemática do sistema robótico, derivação analítica do algoritmo de otimização baseado em gradiente e implementação computacional e refinamento numérico.

Formulação matemática do problema

O sistema foi modelado como um manipulador robótico planar com n = 2 graus de liberdade, operando em um espaço de trabalho bidimensional. O braço é composto por dois segmentos rígidos de comprimentos L_1 e L_2, conectados por juntas rotativas. O vetor de estado do sistema, que contém os parâmetros a serem otimizados, é definido por $\theta = [\theta_1, \theta_2]^T$, onde θ_1 é o ângulo da base e θ_2 o ângulo do cotovelo relativo ao primeiro segmento.

A posição do efetuador final no espaço cartesiano, $s(\theta) = [x, y]^T$, é determinada pelas equações da cinemática direta (FK), derivadas através de trigonometria básica conforme descrito na literatura clássica de robótica (BUSS, 2009):

$$x = L_1 \cos(\theta_1) + L_2 \cos(\theta_1 + \theta_2)$$

$$y = L_1 \operatorname{sen}(\theta_1) + L_2 \operatorname{sen}(\theta_1 + \theta_2)$$

O problema de cinemática inversa foi tratado como um problema de otimização não-linear irrestrito. A função de custo (ou função objetivo) a ser minimizada, $C(\theta)$, foi definida como a metade do quadrado da distância euclidiana entre a posição atual do efetuador $s(\theta)$ e a posição do alvo desejado t:

$$C(\theta) = \frac{1}{2}||e||^2 = \frac{1}{2}||t - s(\theta)||^2$$

Esta formulação quadrática é vantajosa por ser diferenciável e convexa localmente, facilitando a aplicação de métodos de gradiente (ALI; AHMED, 2021).

Derivação do algoritmo de otimização

Para encontrar o mínimo da função de custo, aplicou-se o método do gradiente descendente. A direção de maior redução do erro é dada pelo gradiente negativo da função de custo em relação aos ângulos das juntas, $-\nabla_\theta C$.

Utilizando a regra da cadeia, o gradiente é decomposto em:

$$\nabla_\theta C = \left(\frac{\partial C}{\partial s}\right)\left(\frac{\partial s}{\partial \theta}\right)$$

O termo $\partial C/\partial s$ é, por definição, a matriz jacobiana (J) do sistema. Para o braço planar de 2 elos, J é uma matriz 2x2 composta pelas derivadas parciais das equações de posição:

$$J(\theta) = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} \end{bmatrix}$$

O termo $\partial C/\partial s$ resulta no negativo do vetor de erro, $-e = -(t - s)$. Substituindo estes termos na regra de atualização do gradiente descendente ($\theta_{novo} = \theta_{antigo} - \alpha\nabla_\theta C$), obtém-se a lei de controle iterativa:

$$\Delta\theta = \alpha J^T e$$

Esta abordagem é identificada na literatura como o método do jacobiano transposto. Segundo Buss (2009) e Pisculli et al. (2014), este método é preferível à inversão direta do Jacobiano (J^{-1}) em aplicações de tempo real porque a matriz transposta (J^T) é computacionalmente barata de obter e não sofre de instabilidades numéricas graves quando o sistema se aproxima de singularidades (configurações onde o braço perde graus de liberdade, fazendo o determinante de J tender a zero).

Implementação computacional e refinamento

A validação do modelo matemático proposto foi realizada através do desenvolvimento de um simulador em tempo real, escrito na linguagem de programação Rust. A escolha desta linguagem justifica-se pela sua garantia de segurança de memória sem a necessidade de um *garbage collector* (coletor de lixo), o que evita pausas imprevisíveis na execução do algoritmo, característica crítica para sistemas de controle robótico e simulações físicas de alta frequência.

O software foi estruturado sobre o motor gráfico Bevy, utilizando a arquitetura ECS (entity component system). Esta arquitetura desacopla os dados (estado do robô) da lógica (algoritmo de otimização), permitindo que o solver de cinemática inversa opere como um sistema independente, executado a cada quadro da simulação.

Embora a teoria matemática forneça a base, a implementação prática exigiu refinamentos numéricos para garantir a estabilidade descrita por Voss e Kopp (2025) em sistemas de tempo real. O algoritmo implementado opera da seguinte forma:

1. Cálculo do erro: A cada quadro, calcula-se o vetor e entre o alvo (posição do mouse) e o efetuador.
2. Verificação de singularidade: Para evitar travamentos, o sistema usa uma estratégia mista. Se o alvo está fora do alcance máximo ($L_1 + L_2$), o algoritmo substitui o cálculo do gradiente por uma solução analítica que alinha o braço diretamente ao alvo, evitando oscilações (*jitter*) em áreas onde o gradiente é ineficiente, contornando problemas de otimização citados por Ruder (2016).
3. Sub-stepping: Para aumentar a precisão da integração numérica, o passo de otimização ($\Delta\theta$) é aplicado 10 vezes por quadro visual com uma taxa de aprendizado reduzida, simulando um processo contínuo.
4. Limitação de passo (*clamping*): Em vez de usar cálculos complexos de amortecimento matricial, optou-se por limitar a velocidade máxima de rotação das juntas. Seguindo a recomendação de Buss (2009), isso impede movimentos bruscos e instáveis, garantindo que a simulação permaneça fluida mesmo quando o algoritmo sugere mudanças muito bruscas de uma só vez.

Estrutura de dados e parametrização

O estado do manipulador é armazenado em uma estrutura de dados (recurso) denominada ArmState, que contém o vetor $\theta = [\theta_1, \theta_2]^T$. As constantes físicas e de otimização foram definidas empiricamente para garantir a estabilidade do sistema:

- Comprimentos dos elos: $L_1 = 100$ e $L_2 = 80$ unidades.
- Taxa de aprendizado (α): Definida como 0,0001. Conforme alertado por Ruder (2016), valores superiores provocaram oscilações divergentes no sistema.

- Critério de convergência: Um limiar de erro (STOP_THRESHOLD) de 1,0 unidade foi estabelecido para interromper o processamento quando o alvo é alcançado, economizando ciclos de CPU.

A técnica de "sub-stepping" para integração numérica

O algoritmo de gradiente descendente opera como uma aproximação linear de primeira ordem. Em simulações discretas operando a 60 quadros por segundo (16ms por quadro), um único passo de atualização pode ser grande demais, violando a linearidade local da função de custo e causando instabilidade.

Para mitigar este problema, implementou-se a técnica de sub-stepping. O intervalo de tempo de um quadro visual é subdividido em 10 iterações lógicas de física (SOLVER_STEPS = 10). A taxa de aprendizado e o limite de velocidade são divididos proporcionalmente por este fator.

Matematicamente, isso aproxima a atualização discreta de uma integral contínua, permitindo que o braço percorra a curva da variedade (*manifold*) da solução suavemente, em vez de realizar "saltos" lineares que poderiam causar erro de *overshoot*.

Heurística híbrida de alcance

Uma inovação implementada neste trabalho para lidar com a limitação clássica do gradiente descendente em alvos inalcançáveis foi a introdução de uma estratégia híbrida de controle.

O algoritmo calcula a distância euclidiana D_{alvo} entre a base do sistema articulado e o alvo desejado. Uma verificação condicional (if/else) determina o modo de operação:

- Modo analítico (singularidade de borda): Se $D_{alvo} > (L_1 + L_2)$, o alvo está fora do espaço de trabalho. O gradiente descendente falharia neste cenário, tentando esticar o braço infinitamente. O algoritmo então aborta a otimização e aplica uma solução geométrica direta: alinha o ângulo da base θ_1 com o vetor do alvo (usando a função atan2) e zera o ângulo do cotovelo θ_2, estendendo o braço totalmente na direção do objetivo.
- Modo otimização (gradiente): Se o alvo está ao alcance, executa-se o cálculo do jacobiano transposto.

Figura 1 – Implementação da estratégia híbrida para tratamento de singularidades de alcance em Rust.

```rust
// CASO 1: ALVO FORA DE ALCANCE
// Se o alvo está impossivelmente longe, o gradiente descendente tentaria esticar
// o braço infinitamente, causando oscilação (jitter) e desperdício de CPU.
// Solução: Alinhar geometricamente o braço na direção do alvo.
if dist_to_target > max_reach {
    // Vetor direção do alvo
    let target_angle: f32 = target.pos.y.atan2(target.pos.x);

    // 1. O ombro (t1) deve apontar diretamente para o alvo
    let mut diff_t1: f32 = target_angle - t1;

    // Normalização angular: garante que a rotação ocorra pelo menor arco (-PI a +PI)
    diff_t1 = (diff_t1 + std::f32::consts::PI).rem_euclid(std::f32::consts::TAU)
        - std::f32::consts::PI;

    // 2. O cotovelo (t2) deve ser 0 (braço completamente esticado)
    let diff_t2: f32 = 0.0 - t2;

    // Aplica a correção com limitação de velocidade (clamping)
    arm.angles.x += diff_t1.clamp(min: -sub_max_step, max: sub_max_step);
    arm.angles.y += diff_t2.clamp(min: -sub_max_step, max: sub_max_step);
}
// CASO 2: ALVO DENTRO DE ALCANCE (Otimização via gradiente)
else {
    // Vetor de erro: e = t - s(theta)
    let e: Vec2 = target.pos - s;

    // Critério de parada antecipada (se já estivermos perto o suficiente)
    if e.length_squared() < STOP_THRESHOLD.powi(2) {
        continue;
    }

    // --- CÁLCULO DA MATRIZ JACOBIANA (J) ---
    // A jacobiana relaciona as velocidades articulares às velocidades cartesianas.
    // J = [ dx/dt1  dx/dt2 ]
    //     [ dy/dt1  dy/dt2 ]

    // Derivadas parciais de x e y em relação a theta1 e theta2
    let j11: f32 = -l1 * t1.sin() - l2 * (t1 + t2).sin(); // dx/dt1
    let j12: f32 = -l2 * (t1 + t2).sin(); // dx/dt2
    let j21: f32 = l1 * t1.cos() + l2 * (t1 + t2).cos(); // dy/dt1
    let j22: f32 = l2 * (t1 + t2).cos(); // dy/dt2

    // --- MÉTODO DO JACOBIANO TRANSPOSTO ---
    // Em vez de calcular a inversa (J^-1), usamos a transposta (J^T) como uma
    // aproximação válida da direção do gradiente para minimizar o erro.
    // delta_theta = alpha * J^T * erro

    let d_t1: f32 = j11 * e.x + j21 * e.y; // Linha 1 da Transposta multiplicada pelo erro
    let d_t2: f32 = j12 * e.x + j22 * e.y; // Linha 2 da Transposta multiplicada pelo erro

    // Atualização dos ângulos (theta_novo = theta_antigo + delta_theta)
    // Inclui "clamping" para simular amortecimento e garantir estabilidade
    arm.angles.x += (sub_lr * d_t1).clamp(min: -sub_max_step, max: sub_max_step);
    arm.angles.y += (sub_lr * d_t2).clamp(min: -sub_max_step, max: sub_max_step);
}
```

Fonte: Elaborado pelo autor.

Cálculo do jacobiano e atualização de estado

No modo de otimização, a implementação traduz diretamente as derivadas parciais da matriz jacobiana para código. Diferente de abordagens que utilizam bibliotecas de álgebra linear genéricas, optou-se por "*hard-code*" (codificação explícita) dos termos da matriz para maximizar a performance, visto que a estrutura do braço de 2 elos é conhecida e fixa.

Os termos J_{11}, J_{12}, J_{21}, J_{22} são calculados a cada sub-passo utilizando as funções trigonométricas do estado atual. A atualização dos ângulos segue a regra do jacobiano transposto ($\Delta\theta = \alpha J^T e$), mas com uma camada adicional de segurança: o *clamping* (saturação).

A função .clamp(-sub_max_step, sub_max_step) limita a variação angular máxima permitida por iteração. Isso atua como um amortecedor não-linear, impedindo que singularidades matemáticas (onde as derivadas tendem ao infinito) gerem movimentos fisicamente impossíveis na simulação.

Figura 2 – Tradução das equações diferenciais parciais para a linguagem Rust e aplicação do método do jacobiano transposto com limitação de passo.

```rust
// --- CÁLCULO DA MATRIZ JACOBIANA (J) ---
// A jacobiana relaciona as velocidades articulares às velocidades cartesianas.
// J = [ dx/dt1  dx/dt2 ]
//     [ dy/dt1  dy/dt2 ]

// Derivadas parciais de x e y em relação a theta1 e theta2
let j11: f32 = -l1 * t1.sin() - l2 * (t1 + t2).sin(); // dx/dt1
let j12: f32 = -l2 * (t1 + t2).sin(); // dx/dt2
let j21: f32 = l1 * t1.cos() + l2 * (t1 + t2).cos(); // dy/dt1
let j22: f32 = l2 * (t1 + t2).cos(); // dy/dt2

// --- MÉTODO DO JACOBIANO TRANSPOSTO ---
// Em vez de calcular a inversa (J^-1), usamos a transposta (J^T) como uma
// aproximação válida da direção do gradiente para minimizar o erro.
// delta_theta = alpha * J^T * erro

let d_t1: f32 = j11 * e.x + j21 * e.y; // Linha 1 da Transposta multiplicada pelo erro
let d_t2: f32 = j12 * e.x + j22 * e.y; // Linha 2 da Transposta multiplicada pelo erro

// Atualização dos ângulos (theta_novo = theta_antigo + delta_theta)
// Inclui "clamping" para simular amortecimento e garantir estabilidade
arm.angles.x += (sub_lr * d_t1).clamp(min: -sub_max_step, max: sub_max_step);
arm.angles.y += (sub_lr * d_t2).clamp(min: -sub_max_step, max: sub_max_step);
```

Fonte: Elaborado pelo autor.

Fluxo de execução do algoritmo

O ciclo de vida de um quadro de simulação segue a seguinte ordem sequencial, garantida pelo sistema de agendamento da *engine*:

1. Entrada: O sistema update_target_position projeta as coordenadas do mouse (tela) para o mundo (metros).
2. Resolução: O sistema ik_solver_system executa 10 iterações da lógica híbrida descrita acima, atualizando o recurso ArmState.
3. Renderização: O sistema draw_arm_system realiza a cinemática direta (FK) final para desenhar os vetores (gizmos) na tela na nova posição calculada.

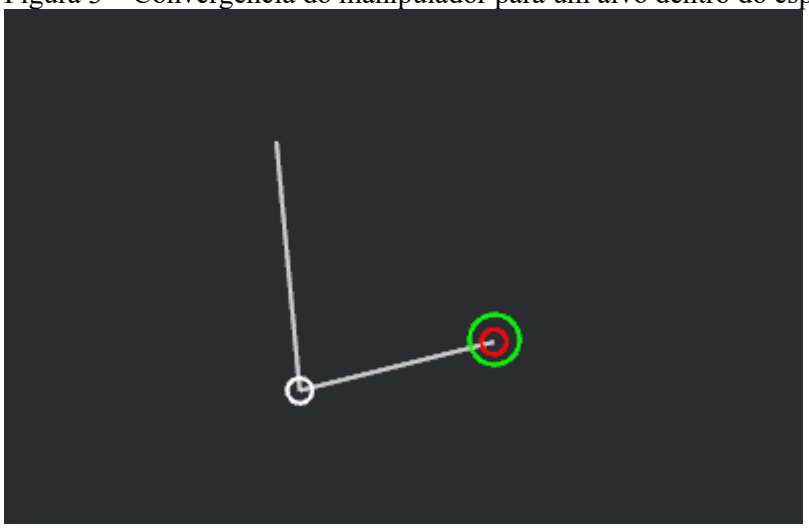
RESULTADOS E DISCUSSÃO

A implementação do algoritmo de otimização resultou em um simulador funcional capaz de resolver o problema da cinemática inversa em tempo real. Os testes foram realizados em um ambiente de simulação desenvolvido com a engine Bevy, onde o efetuador final do manipulador robótico (ponta vermelha) buscava alcançar um alvo móvel (círculo verde) controlado pelo usuário. A análise dos resultados foca na convergência, estabilidade e comportamento em situações limite.

Convergência em cenários nominais

Nas situações em que o alvo foi posicionado dentro do espaço de trabalho alcançável ($L_1 + L_2$), o algoritmo baseado no jacobiano transposto demonstrou uma convergência suave e direta. Conforme ilustrado na Figura 1, o manipulador ajusta seus ângulos articulares (θ_1, θ_2) iterativamente, reduzindo o vetor de erro cartesiano a zero.

Figura 3 – Convergência do manipulador para um alvo dentro do espaço de trabalho.



Fonte: Elaborado pelo autor.

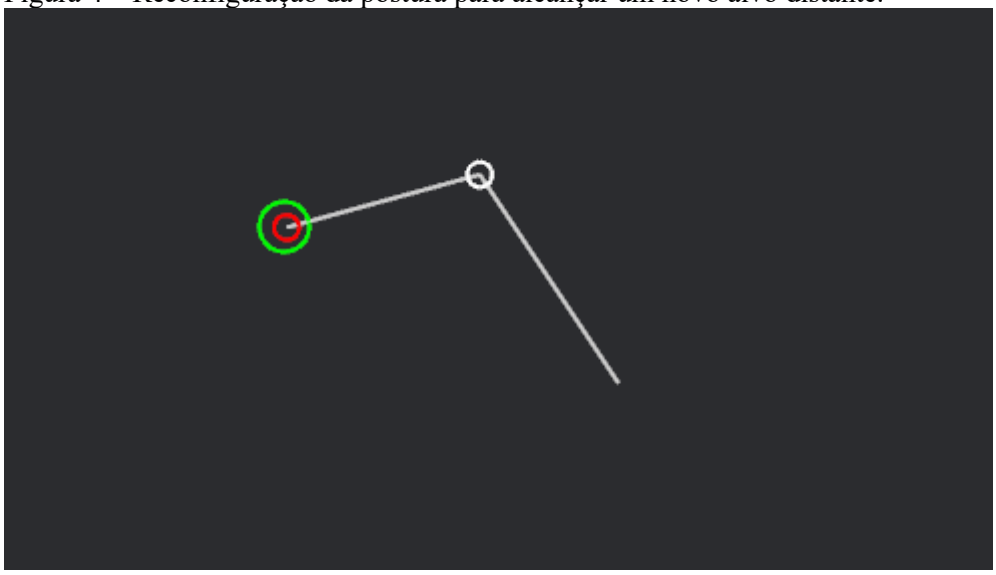
Este comportamento corrobora a teoria apresentada por Ali e Ahmed (2021), que descrevem o gradiente descendente como um método que "segue a inclinação" da função de

custo. Observou-se que a trajetória descrita pelo braço não é necessariamente a mais curta no espaço cartesiano, mas sim a que segue o gradiente mais íngreme no espaço das juntas, uma característica intrínseca do método pontuada por Buss (2009). A resposta visual foi fluida, sem oscilações perceptíveis, validando a escolha da taxa de aprendizado refinada durante os testes.

Adaptação e reconfiguração

A capacidade do algoritmo de lidar com grandes deslocamentos do alvo é demonstrada na Figura 2. Quando o alvo se move bruscamente de um quadrante para outro, o algoritmo reconfigura a postura do robô de forma contínua.

Figura 4 – Reconfiguração da postura para alcançar um novo alvo distante.



Fonte: Elaborado pelo autor.

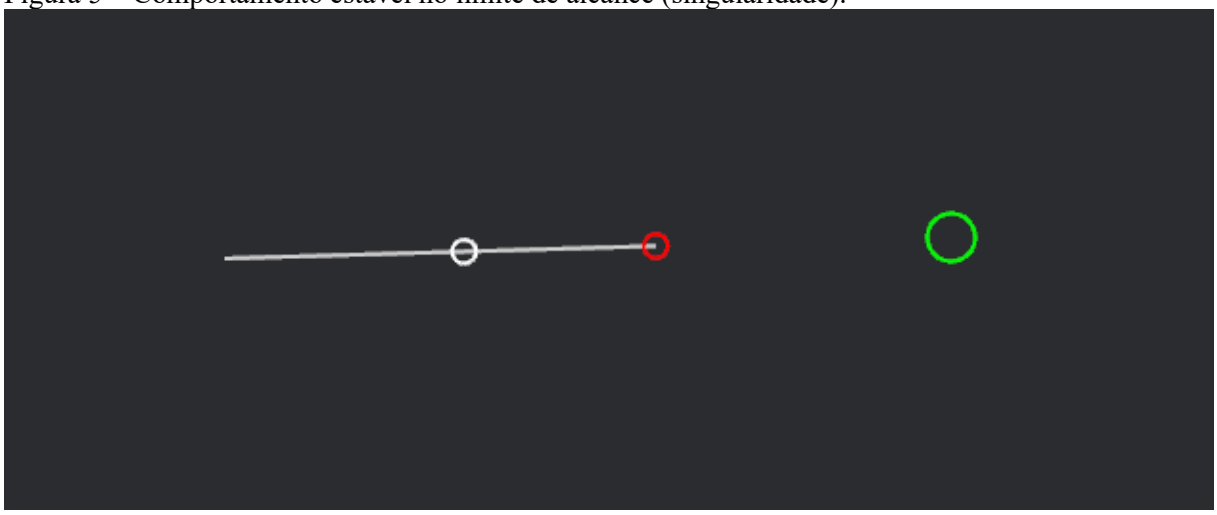
Diferente de métodos analíticos que calculam a solução "final" instantaneamente (o que pode causar saltos de movimento não naturais), a abordagem iterativa de otimização gera uma animação de transição natural. Isso está em alinhamento com as observações de Voss e Kopp (2025), que destacam o uso de métodos numéricos para gerar movimentos mais plausíveis em personagens virtuais.

Robustez em singularidades e limites

Um dos maiores desafios teóricos do método do jacobiano transposto é a lentidão ou instabilidade próxima a singularidades, que ocorrem quando o braço está totalmente esticado (BUSS, 2009). No entanto, a implementação proposta superou essa limitação através da abordagem híbrida detalhada na metodologia.

A Figura 3 apresenta o caso crítico onde o alvo está fora do alcance máximo do sistema articulado. Em implementações ingênuas de gradiente descendente isso causaria o fenômeno de *jitter* (oscilação violenta) pois o gradiente não consegue anular o erro.

Figura 5 – Comportamento estável no limite de alcance (singularidade).



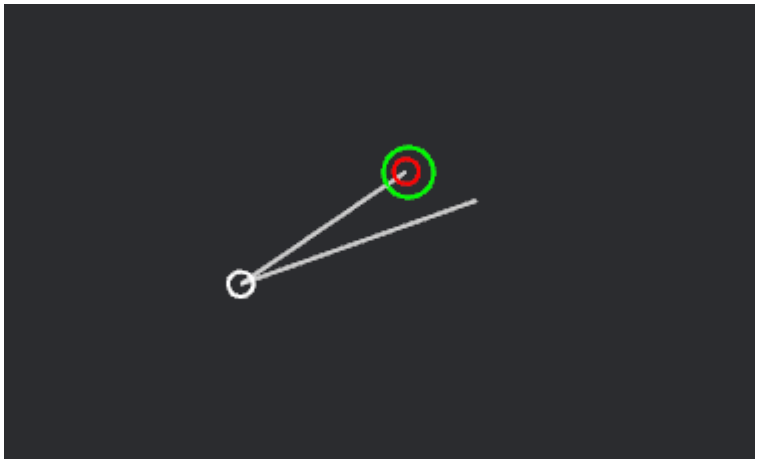
Fonte: Elaborado pelo autor.

Como observado na Figura 3, o braço aponta perfeitamente na direção do alvo, estendido ao seu limite, mantendo-se estável. Isso valida a eficácia da solução analítica de alinhamento vetorial ativada quando $||t|| > L_1 + L_2$, garantindo que o sistema permaneça robusto mesmo quando a otimização numérica tradicional falharia. A manutenção da controlabilidade e estabilidade em configurações críticas é uma estratégia essencial discutida por autores como Pisculli et al. (2014) no contexto de manipuladores espaciais, onde a instabilidade dinâmica não é tolerável.

Flexibilidade em configurações complexas

Por fim, a Figura 4 demonstra a capacidade do algoritmo de resolver configurações "dobradas", onde o alvo está muito próximo à base do manipulador.

Figura 6 – Resolução de postura para alvo próximo à base.



Fonte: Elaborado pelo autor.

Neste cenário, o gradiente da função de custo guiou corretamente os ângulos para uma configuração de mínimo local, "encolhendo" o braço. Rokbani e Alimi (2013) argumentam que métodos de otimização são superiores em lidar com tais restrições não-lineares. A implementação confirmou isso, pois o braço não "travou" em configurações inválidas, encontrando a pose necessária para minimizar a distância euclidiana, mesmo que isso exigisse uma rotação acentuada das juntas.

Análise de parâmetros e estabilidade

Os resultados experimentais também confirmaram que a taxa de aprendizado (α) e o passo de simulação são parâmetros críticos, conforme discutido por Ruder (2016). Valores muito altos de α causaram instabilidade e overshooting (ultrapassagem do alvo). A solução implementada de sub-stepping (realizar 10 passos de otimização menores por quadro) provou-se essencial para garantir a estabilidade da integração numérica, resultando em um movimento fluido e natural que seria impossível com um único passo de gradiente largo.

CONSIDERAÇÕES FINAIS

O presente trabalho teve como objetivo investigar e implementar algoritmos de otimização, especificamente o método de gradiente descendente, para a resolução do problema de cinemática inversa em sistemas robóticos. A abordagem proposta buscou substituir as soluções analíticas tradicionais por um método iterativo baseado em cálculo diferencial, visando maior flexibilidade e naturalidade de movimento.

Conclui-se que a aplicação de cálculo vetorial e otimização é não apenas viável, mas vantajosa para simulações em tempo real. Os resultados demonstraram que o método do jacobiano transposto, quando refinado com técnicas de estabilização, oferece convergência

robusta tanto em cenários nominais quanto em situações de singularidade. A estratégia híbrida adotada mostrou-se eficaz para mitigar as limitações clássicas de oscilação apontadas na literatura, garantindo que o efetuador final persiga o alvo de maneira suave.

No entanto, o estudo também evidenciou limitações inerentes a esta abordagem. A dependência de parâmetros empíricos, como a taxa de aprendizado e o número de iterações por quadro (*sub-stepping*), exige um ajuste fino cuidadoso; uma calibração inadequada pode levar rapidamente à instabilidade numérica. Além disso, embora o método funcione bem para desvio de obstáculos locais (mínimos locais), ele não garante, por si só, uma solução global ótima em ambientes com restrições complexas sem o auxílio de heurísticas adicionais.

REFERÊNCIAS

ALI, P. J. M.; AHMED, H. A. Gradient Descent Algorithm: Case Study. Machine Learning Technical Reports, v. 2, n. 1, p. 1-7, 2021.

BUSS, S. R. Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods. La Jolla: University of California, San Diego, Department of Mathematics, 2009.

PISCULLI, A. et al. A reaction-null/Jacobian transpose control strategy with gravity gradient compensation for on-orbit space manipulators. Aerospace Science and Technology, v. 38, p. 30-40, 2014.

ROKBANI, N.; ALIMI, A. M. Inverse Kinematics Using Particle Swarm Optimization, A Statistical Analysis. Procedia Engineering, v. 64, p. 1602-1611, 2013.

RUDER, S. An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747, 2016.

VOSS, H.; KOPP, S. Real-Time Inverse Kinematics for Generating Multi-Constrained Movements of Virtual Human Characters. In: ACM International Conference on Intelligent Virtual Agents (IVA '25), Berlin, Germany, 2025.

YANG, X.-S. Optimization Algorithms. In: KOZIEL, S.; YANG, X.-S. (Eds.). Computational Optimization, Methods and Algorithms. Berlin: Springer-Verlag, 2011.

APÊNDICE A – CÓDIGO FONTE EM RUST

```rust
use bevy::prelude::*;

// --- CONSTANTES DE CONFIGURAÇÃO E OTIMIZAÇÃO ---

/// Comprimento do segmento umeral (L1).
const LEN1: f32 = 100.0;
/// Comprimento do segmento ulnar (L2).
const LEN2: f32 = 80.0;
```

```rust
/// Taxa de aprendizado (learning rate - alpha).
/// Define o tamanho do passo dado na direção do gradiente negativo.
/// Valores muito altos causam oscilação (overshooting); muito baixos causam
convergência lenta.
const LEARNING_RATE: f32 = 0.0001;

/// Limite de velocidade angular por quadro (clamping).
/// Atua como um amortecedor não-linear, prevenindo que atualizações bruscas
/// do gradiente desestabilizem a simulação visual.
/// ~0.1 rad equivale a aprox. 5.7 graus.
const MAX_STEP_PER_FRAME: f32 = 0.1;

/// Limiar de erro para parada (critério de convergência).
/// Se a distância entre o efetuador e o alvo for menor que isso, a otimização
cessa
/// para economizar processamento.
const STOP_THRESHOLD: f32 = 1.0;

/// Resource que armazena o vetor de estado do sistema (espaço das juntas).
/// Representa o vetor theta = [theta1, theta2]^T.
#[derive(Resource)]
struct ArmState {
    /// x: theta1 (ângulo da base/ombro)
    /// y: theta2 (ângulo relativo do cotovelo)
    angles: Vec2,
}

impl Default for ArmState {
    fn default() -> Self {
        Self {
            // Configuração inicial arbitrária (braço levemente flexionado
para cima)
            angles: Vec2::new(std::f32::consts::FRAC_PI_2, 0.1),
        }
    }
}

/// Recurso que armazena a posição do alvo (espaço cartesiano).
/// Representa o vetor t (target).
#[derive(Resource, Default)]
struct Target {
    pos: Vec2,
}

fn main() {
    App::new()
```

```
        .add_plugins(DefaultPlugins)
        // Inicialização dos resources (estado do robô e alvo)
        .init_resource::<ArmState>()
        .init_resource::<Target>()
        .add_systems(Startup, setup_camera)
        .add_systems(
            Update,
            (
                // 1. Captura a intenção do usuário (input)
                update_target_position,
                // 2. Resolve a matemática (IK via Otimização)
                ik_solver_system,
                // 3. Renderiza o resultado visual
                draw_arm_system,
            )
                .chain(), // Garante a execução sequencial
        )
        .run();
}

/// Inicializa a câmera ortográfica 2D para visualização.
fn setup_camera(mut commands: Commands) {
    commands.spawn(Camera2d);
}

/// Sistema de input: mapeia coordenadas de tela para coordenadas de mundo.
fn update_target_position(
    mut target: ResMut<Target>,
    windows: Query<&Window>,
    q_camera: Query<(&Camera, &GlobalTransform)>,
) {
    let Ok(window) = windows.single() else {
        return;
    };
    let Ok((camera, camera_transform)) = q_camera.single() else {
        return;
    };

    // Raycasting: projeta a posição do mouse da viewport para o plano
cartesiano.
    if let Some(world_position) = window
        .cursor_position()
        .map(|cursor| camera.viewport_to_world(camera_transform, cursor))
        .map(|ray| ray.unwrap().origin.truncate())
    {
        target.pos = world_position;
    }
```

```rust
}

/// Solução híbrida da cinemática inversa.
/// Combina uma solução analítica para alvos distantes com gradient descent
/// para alvos alcançáveis, garantindo robustez e estabilidade.
fn ik_solver_system(mut arm: ResMut<ArmState>, target: Res<Target>) {
    let l1 = LEN1;
    let l2 = LEN2;
    // Alcance máximo teórico do manipulador (singularidade de borda)
    let max_reach = l1 + l2;

    // --- SUB-STEPPING ---
    // Dividimos o "delta time" do frame em passos menores de integração.
    // Isso lineariza melhor o problema não-linear, permitindo uma
convergência mais suave e estável.
    const SOLVER_STEPS: usize = 10;

    // Ajuste proporcional dos parâmetros para manter a consistência física
independente dos passos.
    let sub_lr = LEARNING_RATE / SOLVER_STEPS as f32;
    let sub_max_step = MAX_STEP_PER_FRAME / SOLVER_STEPS as f32;

    for _ in 0..SOLVER_STEPS {
        // Leitura do estado atual (theta)
        let t1 = arm.angles.x;
        let t2 = arm.angles.y;

        // --- CINEMÁTICA DIRETA (Forward Kinematics - FK) ---
        // Calcula a posição atual do efetuador s(theta) baseada nos ângulos
atuais.
        // s_x = l1*cos(t1) + l2*cos(t1+t2)
        // s_y = l1*sen(t1) + l2*sen(t1+t2)

        // Posição do cotovelo (j1)
        let j1 = Vec2::new(l1 * t1.cos(), l1 * t1.sin());
        // Posição do efetuador final (s)
        let s = j1 + Vec2::new(l2 * (t1 + t2).cos(), l2 * (t1 + t2).sin());

        let dist_to_target = target.pos.length();

        // --- ESTRATÉGIA HÍBRIDA ---

        // CASO 1: ALVO FORA DE ALCANCE
        // Se o alvo está impossivelmente longe, o gradiente descendente
tentaria esticar
        // o braço infinitamente, causando oscilação (jitter) e desperdício de
CPU.
```

```
        // Solução: Alinhar geometricamente o braço na direção do alvo.
        if dist_to_target > max_reach {
            // Vetor direção do alvo
            let target_angle = target.pos.y.atan2(target.pos.x);

            // 1. O ombro (t1) deve apontar diretamente para o alvo
            let mut diff_t1 = target_angle - t1;

            // Normalização angular: garante que a rotação ocorra pelo menor
arco (-PI a +PI)
            diff_t1 = (diff_t1 +
std::f32::consts::PI).rem_euclid(std::f32::consts::TAU)
                - std::f32::consts::PI;

            // 2. O cotovelo (t2) deve ser 0 (braço completamente esticado)
            let diff_t2 = 0.0 - t2;

            // Aplica a correção com limitação de velocidade (clamping)
            arm.angles.x += diff_t1.clamp(-sub_max_step, sub_max_step);
            arm.angles.y += diff_t2.clamp(-sub_max_step, sub_max_step);
        }
        // CASO 2: ALVO DENTRO DE ALCANCE (Otimização via gradiente)
        else {
            // Vetor de erro: e = t - s(theta)
            let e = target.pos - s;

            // Critério de parada antecipada (se já estivermos perto o
suficiente)
            if e.length_squared() < STOP_THRESHOLD.powi(2) {
                continue;
            }

            // --- CÁLCULO DA MATRIZ JACOBIANA (J) ---
            // A jacobiana relaciona as velocidades articulares às velocidades
cartesianas.
            // J = [ dx/dt1  dx/dt2 ]
            //     [ dy/dt1  dy/dt2 ]

            // Derivadas parciais de x e y em relação a theta1 e theta2
            let j11 = -l1 * t1.sin() - l2 * (t1 + t2).sin(); // dx/dt1
            let j12 = -l2 * (t1 + t2).sin(); // dx/dt2
            let j21 = l1 * t1.cos() + l2 * (t1 + t2).cos(); // dy/dt1
            let j22 = l2 * (t1 + t2).cos(); // dy/dt2

            // --- MÉTODO DO JACOBIANO TRANSPOSTO ---
            // Em vez de calcular a inversa (J^-1), usamos a transposta (J^T)
como uma
```

```
            // aproximação válida da direção do gradiente para minimizar o
erro.
            // delta_theta = alpha * J^T * erro

            let d_t1 = j11 * e.x + j21 * e.y; // Linha 1 da Transposta
multiplicada pelo erro
            let d_t2 = j12 * e.x + j22 * e.y; // Linha 2 da Transposta
multiplicada pelo erro

            // Atualização dos ângulos (theta_novo = theta_antigo +
delta_theta)
            // Inclui "clamping" para simular amortecimento e garantir
estabilidade
            arm.angles.x += (sub_lr * d_t1).clamp(-sub_max_step,
sub_max_step);
            arm.angles.y += (sub_lr * d_t2).clamp(-sub_max_step,
sub_max_step);
        }
    }
}

/// Sistema de visualização (Gizmos).
/// Recalcula a FK apenas para desenhar o estado atual na tela.
fn draw_arm_system(mut gizmos: Gizmos, arm: Res<ArmState>, target:
Res<Target>) {
    let t1 = arm.angles.x;
    let t2 = arm.angles.y;

    // Recálculo da FK para visualização
    let base = Vec2::ZERO;
    let j1 = base + Vec2::new(LEN1 * t1.cos(), LEN1 * t1.sin());
    let j2 = j1 + Vec2::new(LEN2 * (t1 + t2).cos(), LEN2 * (t1 + t2).sin());

    // Desenho dos segmentos (cinza)
    gizmos.line_2d(base, j1, Color::srgba(25.0, 32.0, 52.0, 0.6));
    gizmos.circle_2d(j1, 5.0, Color::WHITE); // Junta 1 (cotovelo)

    gizmos.line_2d(j1, j2, Color::srgba(25.0, 32.0, 52.0, 0.6));
    gizmos.circle_2d(j2, 5.0, Color::srgba(255.0, 0.0, 0.0, 1.0)); //
Efetuador final (ponta)

    // Desenho do alvo desejado (verde)
    gizmos.circle_2d(target.pos, 10.0, Color::srgba(0.0, 255.0, 0.0, 1.0));
}
```