DESENVOLVIMENTO DE APLICAÇÃO PARA UTILIZAÇÃO DO COMPUTADOR POR MEIO DE GESTOS COM A MÃO USANDO OPENCV E MEDIAPIPE

Arthur Wagner Alves Rodrigues[1], Marilée Patta[2]

[1]Aluno do Curso de Sistemas de Informação da Unimontes, Montes Claros, Brasil (arthurdk01@gmail.com)
[2]Professora do Curso de Sistemas de Informação da Unimontes, Montes Claros, Brasil (marileep@unimontes.br)

Resumo: Desenvolveu-se aplicação para uso do computador pessoal com gestos da mão, capturados via webcam e interpretados com os frameworks *Python, OpenCV, MediaPipe* e P*yAutoGUI,* uma alternativa de navegação sem dependência de dispositivos tradicionais. Para os testes, foram cadastraram-se três gestos. Em média, os testes atingiram acurácia de 91,5%, precisão de 96,5%, revocação de 85,5% e F1-Score de 0,91, indicando um desempenho satisfatório de reconhecimento de gestos.

Palavras-chave: comandos por gestos; interpretação de gestos; visão computacional.

INTRODUÇÃO

A interação humano-computador vem passando por sucessivas transformações: do teclado e mouse convencionais às interfaces de voz e, mais recentemente, a detecção de gestos. Embora estas novidades alcancem o grande público em jogos eletrônicos, aparelhos de realidade virtual e telefones celulares, seu emprego como recurso de acessibilidade permanece limitado.

Essa limitação ganha contornos ainda mais relevantes quando se observa a dimensão do público que poderia se beneficiar de alternativas de controle.

No mundo inteiro e especificamente no Brasil depara-se com pessoas com várias barreiras físicas que repercutem em menor acesso à educação, trabalho e renda. O IBGE (2023), através da Pesquisa Nacional de Domicílios - PNAD Contínua de 2022, mostra dados alarmantes. Em 2022, havia 14,4 milhões de pessoas com deficiência. Entre as 198,3 milhões de pessoas com dois anos ou mais, 14,4 milhões (ou 7,3%) eram pessoas com deficiência. Assim, oferecer um meio alternativo de navegação sem dependência de dispositivos tradicionais atende a uma demanda social, auxiliando na inclusão social. Além do público com enfermidades neuro-motoras (Parkinson, paralisia cerebral), a solução pode auxiliar idosos, pacientes em reabilitação ou qualquer usuário cujas condições momentâneas dificultem o uso de teclado e mouse.

Neste contexto, desenvolveu-se uma aplicação que permita a utilização de funcionalidades básicas do computador pessoal por meio de gestos com a mão, capturados via webcam e interpretados com o uso dos frameworks *Python OpenCV, MediaPipe* e P*yAutoGUI*, visando promover acessibilidade e inclusão digital. Especificamente pretendeu-se capturar gestos e acionar comandos a serem executados, como cliques, rolagem, alternância de janela e navegação na web e avaliar a taxa de acerto do reconhecimento de gestos.

Os avanços tecnológicos criam a oportunidade de oferecer uma nova via de controle do computador. A visão computacional tem por meta permitir que máquinas “vejam” e entendam cenas do mundo real, extraindo de imagens ou vídeos representações estruturadas que sirvam a tarefas como detecção de objetos, estimação de pose ou reconhecimento de atividades (Szeliski, 2022). Seu avanço foi catalisado por frameworks de código aberto. O OpenCV, lançado em 2000, tornou-se padrão para operações de processamento de imagem (Bradski, 2000), permitindo a captura de vídeo do webcam do computador e repassando essas informações a outras ferramentas para que possam analisar em tempo real o vídeo fornecido.

Para reconhecimento de gestos, métodos de detecção de pontos estimam a posição tridimensional dos dedos a partir de vídeo fornecido por câmeras RGB (*um tipo de câmera comum que captura imagens em três canais de cores: vermelho, verde e azul*). O modelo *MediaPipe Hands* emprega um detector de palmas seguido de uma rede de *landmarks* (pontos de referência anatômicos que representam a posição de partes específicas da mão em uma imagem ou vídeo), permitindo identificar qual posição ou gesto uma mão capturada em webcam estaria em tempo real.

A biblioteca *Open Computer Vision Library*, ou Biblioteca Aberta de Visão Computacional - OpenCV é utilizada em projetos de visão computacional por oferecer ferramentas eficientes para captura, manipulação e exibição de imagens e vídeos em tempo real. Criada por Gary Bradski no ano 2000, a *OpenCV* foi projetada inicialmente para acelerar a adoção da visão computacional na indústria, mantida como um projeto de código aberto. Ela é essencial para alimentar o modelo de detecção do *MediaPipe Hands,* que opera sobre os frames (um frame é uma imagem individual dentro de uma sequência de frames que compõem um vídeo) RGB extraídos pelo OpenCV. Sua eficiência em tempo real e suporte nativo a múltiplas plataformas (*Windows, Linux, Android*) reforçam a escolha do OpenCV como parte central da arquitetura do sistema proposto, garantindo desempenho adequado mesmo em hardware convencional.

A biblioteca funciona de forma eficiente mesmo em computadores comuns, sem precisar de sensores especiais. Por isso, é muito utilizada em aplicações de controle por gestos, realidade aumentada, jogos e ferramentas de acessibilidade.

Dentro desse ecossistema, o *MediaPipe Hands* implementa um *pipeline* (sequência de etapas organizadas para processar dados de forma contínua) de três estágios: 1) Detecção de palmas: o sistema localiza a presença de uma ou duas palmas na imagem. Para isso, utiliza um modelo leve chamado *BlazePalm*, que consegue identificar rapidamente a região da mão com alta precisão. Essa etapa é fundamental para delimitar a área que será analisada nos próximos passos; 2) Estimação de *landmarks* da mão: após identificar a palma da mão, o *MediaPip*e recorta e ajusta essa região e aplica um modelo de rede neural que prevê 21 pontos de referência (*landmarks*) na mão. Esses pontos incluem as articulações dos dedos e o centro do punho, permitindo reconstruir a pose da mão em 3D (com profundidade relativa); 3) Classificação de gestos (opcional): com base nas posições dos *landmarks*, é possível adicionar um classificador de gestos. Essa etapa não faz parte do modelo original por padrão, mas pode ser implementada separadamente para traduzir diferentes configurações da mão em comandos, como "abrir", "fechar", "apontar" ou "clicar".

Do ponto de vista de integração, o *MediaPipe Hands* oferece *bindings* (adaptações que permitem usar uma biblioteca em uma linguagem de programação) prontos para C++, *Python*, *Android* e *iOS*. Na prática, isso permite acoplar a biblioteca a aplicações desktop *Windows, Linux, Mac* ou a *apps* móveis, mantendo o mesmo modelo de código.

A *PyAutoGUI* é uma biblioteca *Python* que permite controlar o mouse e o teclado de forma programada, simulando ações como cliques, movimentos, digitação e atalhos de sistema. Ela funciona de maneira multiplataforma (*Windows, Linux e macOS)*, sem necessidade de dependências externas ou permissões elevadas, tornando-se uma ferramenta prática para automação de tarefas no computador.

O design centrado no usuário é reconhecido pela ISO 9241-210 como a abordagem recomendada de uso das pessoas no centro de todas as fases de desenvolvimento (ISO, 2019).

Em acessibilidade digital, considerou-se o *Web Content Accessibility Guidelines* (WCAG). A versão 2.2, publicado em 5 de outubro de 2023, adicionou critérios como *Focus Not Obscured* (o elemento selecionado pelo teclado ou leitor de tela deve estar visível) e *Target Size* (tamanho do alvo, que define dimensões mínimas para botões e links serem facilmente clicáveis), concebidos para usuários com baixa visão ou limitações motoras finas (W3C, 2023). Estes critérios reforçam a necessidade de interfaces que não dependam exclusivamente do clique preciso do mouse, sustentando a motivação deste trabalho.

MATERIAL E MÉTODOS

Iniciou-se com um estudo dos frameworks e bibliotecas que compõem a solução: *OpenCV, MediaPipe Hands* e *PyAutoGUI* de modo a compreender limites de desempenho, dependências e configurações. Foi realizada uma revisão da documentação oficial, análise de estudos correlatos e replicação de exemplos de referência para medir a robustez do uso conjunto das ferramentas que orientaram a realização de configurações, como resolução de captura ou nível de confiança na detecção de palmas, estabelecendo-se requisitos mínimos de hardware e configurações essenciais para uso da aplicação. Posteriormente, foi feito o mapeamento de gestos que consiste em definir um conjunto enxuto de gestos que atendam às necessidades essenciais de navegação, como: Clique; Rolagem; Alternância de janelas; e Movimentação do mouse. Cada gesto foi descrito formalmente pelas relações geométricas dos 21 *landmarks* da mão, definidos pelo *MediaPipe*. A fase inclui a criação de zonas neutras de gestos, visando conter os resultados falsos-positivos. Documentou-se o mapeamento em um manual de usuário que servirá tanto como uma assistência para o usuário quanto um recurso na etapa de validação de usabilidade.

Assim, foram desenvolvidos, em *Python* estruturado em módulos independentes: captura de vídeo pelo *OpenCV*; detecção de mãos e extração de *landmarks* via *MediaPipe*; classificação do gesto com base nas regras estabelecidas; e emissão de comandos no

sistema operacional através da *PyAutoGUI*. Esse fluxo em cadeia foi encapsulado em uma interface gráfica e leve que apresenta, ao usuário, feedback visual imediato, sinalizando o gesto reconhecido e o comando disparado.

Em seguida, fez-se uma avaliação para se apurar a taxa de acerto, conduzida por meio de testes em ambiente controlado, com o objetivo de mensurar a precisão e acurácia geral do sistema de reconhecimento de gestos. Com isso, validaram-se se os gestos mapeados estão corretamente interpretados e traduzidos em comandos pela aplicação, bem como identificar suas eventuais limitações.

Os testes foram realizados em um ambiente interno com iluminação artificial constante, fundo neutro e mínima interferência visual, de modo a reduzir ruídos na detecção dos *landmarks*. Para isso, utilizou-se um computador com configurações típicas de uso doméstico (CPU Intel i3 10100), com webcam USB externa padrão RGB.

Foram cadastrados três gestos. Cada gesto foi executado 30 vezes por três usuários distintos, totalizando 90 tentativas por gesto. Os usuários mantiveram uma distância fixa de 30 centímetros da câmera. Cada repetição seguiu um ciclo estruturado: o participante aproximava a mão da área de alcance do webcam, executava o gesto e, em seguida, retirava a mão do campo de visão do dispositivo antes de iniciar o ciclo seguinte.

Os intervalos de permanência da mão fora do alcance da câmera foram tratados como eventos negativos e monitorados para verificar a ocorrência de reconhecimentos indevidos na ausência de entrada gestual. Dessa forma, cada gesto gerou 90 eventos com mão visível e 90 eventos sem mão visível, totalizando 180 eventos por gesto. Durante a execução, registraram-se: 1) Se o gesto foi corretamente reconhecido (TP); 2) Se o sistema não identificou o gesto quando ele foi executado (FN); 3) Se o sistema reconheceu um gesto quando nenhum estava sendo executado (FP); e 4) Se o sistema não reconheceu corretamente nenhum gesto durante os intervalos sem mão visível (TN).

Assim, nas métricas de avaliação consideram-se os termos: TP (*True Positive* / Verdadeiro Positivo): o sistema reconhece corretamente um gesto quando ele realmente foi executado; TN (*True Negative* / Verdadeiro Negativo): o sistema corretamente identifica que um gesto que não foi executado quando realmente não houve execução desse gesto; FP (*False Positive* / Falso Positivo): o sistema reconhece incorretamente um gesto quando ele não foi executado (falso alarme); FN (*False Negative* / Falso Negativo): o sistema não reconhece um gesto quando ele de fato foi executado (falha de detecção).

As métricas de desempenho adotadas seguem um padrão já desenvolvido na literatura (UDAWI *et al*., 2024), sendo:

1) Acurácia: Proporção de classificações corretas sobre o total de tentativas;

$$Acurácia = \frac{TP+TN}{TP+TN+FP+FN} \quad (1)$$

2) Precisão: Proporção de verdadeiros positivos entre todas as detecções feitas;

$$Precisão = \frac{TP}{TP+FP} \quad (2)$$

3) Revocação: Proporção de verdadeiros positivos entre todas as ocorrências reais do gesto;

$$Revocação = \frac{TP}{TP+FN} \quad (3)$$

4) F1-Score: Média harmônica entre precisão e revocação;

$$F1\ Socore = 2\ x \frac{Precisão\ x\ Revocção}{Precisão+Revocação} \quad (4)$$

5) Taxa de Erro de Classificação: Proporção de classificações incorretas:

$$Tx\ Erro\ Classificação = \frac{FP+FN}{TP+TN+FP+FN} \quad (5)$$

RESULTADOS E DISCUSSÃO

A arquitetura da aplicação foi dividida em módulos independentes. O módulo de câmera é responsável por localizar *webcams* disponíveis, abrir o dispositivo selecionado e capturar os quadros em tempo real. O módulo de gestos recebe os *landmarks* produzidos pelo *MediaPipe* e transforma essas coordenadas em uma representação normalizada da pose. O módulo de automação interpreta o resultado do reconhecimento e aciona o *PyAutoGUI* para mover o cursor, clicar, rolar a tela ou alternar janelas. Já, módulo de interface gráfica reúne os controles utilizados pelo usuário, exibe a imagem da câmera e permite o cadastro dos vínculos entre gestos e ações.

O fluxo de execução começa com a seleção da webcam. Depois disso, um *thread* dedicado captura os frames, converte a imagem para o formato RGB e envia cada quadro ao *MediaPipe Hands*. Quando uma mão é detectada, os 21 *landmarks* são convertidos para objetos internos da aplicação e encaminhados para a etapa de reconhecimento. O resultado é então enviado à interface, que atualiza a pré-visualização e, caso a automação esteja habilitada, dispara a ação correspondente ao gesto reconhecido.

A separação dos módulos reduz o acoplamento entre captura, reconhecimento e automação. Com isso, o

sistema pode continuar exibindo a câmera e registrando gestos mesmo quando a automação está desativada, o que é importante para uso seguro de testes e calibração.

A linguagem escolhida para a implementação foi *Python*, por possuir bibliotecas de visão computacional, automação de interface e criação de aplicações desktop. O gerenciamento de dependências foi feito com o UV, ferramenta utilizada para instalar bibliotecas, executar o projeto, rodar os testes e acionar o comando de compilação da aplicação.

A captura de vídeo foi implementada com *OpenCV* por meio da classe *VideoCapture.* Antes de iniciar o reconhecimento, a aplicação percorre índices de câmera e tenta abrir cada dispositivo encontrado. Quando uma *webcam* válida responde com um frame, suas informações são registradas e apresentadas ao usuário em uma lista de seleção, permitindo, assim, escolher entre diferentes câmeras conectadas ao computador, como webcam interna, webcam USB ou câmera virtual.

Após a seleção, a captura ocorre em um *thread* separado da interface gráfica. Essa decisão impede que a janela principal fique travada durante a leitura dos *frames* e melhora a responsividade do software. A imagem capturada é espelhada horizontalmente, de forma semelhante a um espelho, tornando o controle mais intuitivo para o usuário. Em seguida, o *frame* é convertido do padrão BGR, utilizado pelo *OpenCV*, para RGB, formato esperado pelo *MediaPipe.*

O reconhecimento visual da mão foi realizado com o f*ramework MediaPipe Hands.* A aplicação configura o modelo para detectar uma mão por vez, utilizando processamento em fluxo de vídeo, com rastreamento entre frames. Para reduzir o custo computacional, a complexidade do modelo foi definida como zero e a confiança mínima de detecção e rastreamento em 0,60. No framework, a complexidade igual a zero indica o uso da versão mais leve (*lite model*) disponibilizada pelo MediaPipe, que prioriza a velocidade de inferência e a baixa latência em detrimento da precisão máxima alcançada pela versão completa de nível 1 (ZHANG et al., 2020). Essa escolha é adequada ao objetivo do projeto, pois o sistema precisa operar em tempo real e em computadores de hardware modesto. Já, os valores de confiança mínima iguais a 0,60 funcionam como um filtro para a aceitação dos resultados do modelo: a mão só é considerada detectada ou rastreada quando o *MediaPipe* atribui ao resultado uma confiança mínima de 60%. Dessa forma, busca-se equilibrar desempenho e estabilidade, evitando o excesso de processamento e a aceitação de detecções pouco confiáveis.

Quando a mão é identificada, o *MediaPipe* retorna 21 pontos de referência. Esses pontos representam regiões anatômicas da mão, incluindo punho, articulações intermediárias e pontas dos dedos. Cada ponto contém coordenadas x e y normalizadas em relação à imagem e uma coordenada z relativa, que indica profundidade aproximada em relação ao plano da mão. A aplicação converte esses pontos para uma estrutura própria, mantendo apenas as informações necessárias para o reconhecimento e o controle do cursor.

O algoritmo implementado não compara pixels, verificando, portanto, a geometria da mão. Isso torna o processamento mais leve e reduz a influência de variações de fundo, cor da pele e iluminação, embora a qualidade da câmera e a estabilidade da detecção ainda influenciam no resultado final.

O cadastro de um gesto personalizado ocorre a partir da pose da mão no momento do registro. Mesmo quando a interface exibe 'Nenhum gesto', a aplicação ainda pode salvar a posição atual da mão, caso os *landmarks* estejam disponíveis. Isso acontece porque 'Nenhum gesto' indica apenas que nenhuma pose previamente cadastrada foi reconhecida, e não que o sistema deixou de enxergar a mão. Se houver *landmarks* válidos, a pose pode ser transformada em um novo gesto personalizado.

Para armazenar o gesto, o algoritmo utiliza os 21 *landmarks* retornados pelo *MediaPipe*. As coordenadas são normalizadas em relação ao punho, que funciona como origem local da mão. Em seguida, a escala é ajustada pela maior distância entre o punho e os demais pontos. O resultado é um vetor com 63 valores, composto pelas coordenadas x, y e z de cada *landmark*. Essa normalização reduz a influência da posição absoluta da mão na imagem e do tamanho aparente causado pela distância entre a mão e a câmera.

Cada gesto personalizado recebe um identificador interno, um nome exibido ao usuário e o vetor normalizado da pose. O usuário pode então atrelar esse gesto a uma das ações disponíveis, como clique esquerdo, clique direito, clique duplo, movimentação do cursor, rolagem para cima, rolagem para baixo ou alternância de janela (ver Figura 1).

```python
def normalize_landmarks(landmarks: Sequence[Landmark]) ->
tuple[float, ...]:
    if len(landmarks) < 21:
        return ()

    wrist = landmarks[0]
    scale = max(_distance(wrist, landmark) for landmark in
landmarks)
    scale = max(scale, 1e-6)
    normalized: list[float] = []
    for landmark in landmarks[:21]:
        normalized.extend(
            (
                (landmark.x - wrist.x) / scale,
                (landmark.y - wrist.y) / scale,
                (landmark.z - wrist.z) / scale,
            )
        )
    return tuple(normalized)
```

Figura 1. Função de normalização de *landmarks*.

A identificação de gestos personalizados é feita por comparação de similaridade. A cada novo *frame* com uma mão detectada, a aplicação normaliza a pose atual e calcula a distância média entre seus *landmarks* e os l*andmarks* de cada gesto salvo. O gesto com menor distância é considerado o candidato mais próximo. Para que esse candidato seja aceito, sua distância precisa estar abaixo de um limiar de similaridade configurável. A aplicação utiliza valor padrão de 0,18, limitado entre 0,06 e 0,30. Valores menores tornam o reconhecimento mais rígido, exigindo que o usuário repita a pose originalmente registrada com maior precisão. Valores maiores tornam o reconhecimento mais tolerante, mas aumentam o risco de confundir gestos parecidos.

O mesmo limiar também é utilizado no momento de registrar novos gestos. Se a nova pose estiver dentro da tolerância de um gesto já existente, a aplicação bloqueia o cadastro para evitar sobreposição. Esse mecanismo reduz a chance de dois gestos visualmente muito semelhantes dispararem comandos diferentes, o que poderia gerar cliques ou atalhos inesperados durante o uso (ver Figura 2).

```python
def find_closest_custom_gesture(
    vector: Sequence[float],
    custom_gestures: dict[str, CustomGesture],
    *,
    match_threshold: float = CUSTOM_MATCH_THRESHOLD,
    skip_id: str | None = None,
) -> CustomGestureMatch | None:
    threshold = clamp_match_threshold(match_threshold)
    best_gesture: CustomGesture | None = None
    best_distance = float("inf")

    for custom_gesture in custom_gestures.values():
        if custom_gesture.id == skip_id or
len(custom_gesture.vector) != len(vector):
            continue

        distance = _vector_distance(vector,
custom_gesture.vector)
        if distance < best_distance:
            best_distance = distance
            best_gesture = custom_gesture

    if best_gesture is None or best_distance > threshold:
        return None

    confidence = 1.0 - (best_distance / threshold) * 0.45
    return CustomGestureMatch(best_gesture, best_distance,
max(0.55, min(1.0, confidence)))
```

Figura 2. Função que percorre os gestos salvos, calcula a distância e retorna o gesto mais próximo.

O controle do cursor foi implementado a partir de um ponto de referência da mão. Visando estabilidade, é utilizado o ponto mais baixo da mão na imagem, isto é, o *landmark* com maior coordenada y entre os 21 pontos detectados. Esse ponto tende a ficar próximo à região do punho ou da base da mão, tornando o movimento menos sensível a pequenas mudanças na posição dos dedos.

As coordenadas normalizadas desse ponto são convertidas para as dimensões da tela. O sistema também possui um controle de sensibilidade do mouse, limitado entre 0,40x e 2,50x. A sensibilidade atua como um ganho em relação ao centro da imagem: valores menores reduzem o deslocamento do cursor, enquanto valores maiores ampliam o movimento produzido pela mão.

Para diminuir movimentos bruscos foi aplicada uma suavização temporal. Quando já existe uma posição anterior do cursor, a nova posição é calculada combinando parte da posição anterior com parte do alvo atual. Esse filtro reduz oscilações causadas por pequenas variações dos *landmarks* entre frames consecutivos.

A opção 'Mover cursor com qualquer gesto' foi implementada de forma independente do reconhecimento de gestos personalizados. Quando essa opção está habilitada, a mão deve ser detectada para que o cursor possa ser movido. Portanto, o movimento do cursor não depende do limiar de similaridade nem da classificação de uma pose já registrada (ver Figura 3).

```python
    def _move_cursor(self, point: tuple[float, float] | None,
sensitivity: float) -> None:
        if point is None:
            return

        pyautogui = self._load_pyautogui()
        screen_width, screen_height = pyautogui.size()
        target_x = _apply_sensitivity(point[0], sensitivity) *
screen_width
        target_y = _apply_sensitivity(point[1], sensitivity) *
screen_height

        if self._last_cursor_position is None:
            next_x, next_y = target_x, target_y
        else:
            previous_x, previous_y = self._last_cursor_position
            next_x = previous_x * 0.65 + target_x * 0.35
            next_y = previous_y * 0.65 + target_y * 0.35

        pyautogui.moveTo(next_x, next_y, duration=0.02)
        self._last_cursor_position = (next_x, next_y)
```

Figura 3 - Função que aplica a movimentação do cursor.

A automação das ações foi implementada com *PyAutoGUI*. Quando a automação está ativada e um gesto é reconhecido com confiança suficiente, a aplicação consulta a tabela de vínculos cadastrada pelo usuário e executa a ação correspondente. Caso a automação esteja desativada, o sistema continua reconhecendo e exibindo gestos, mas não envia comandos ao sistema operacional.

As ações implementadas incluem "mover o cursor, clique esquerdo, clique direito, clique duplo, rolagem para cima, rolagem para baixo e alternar janela". A ação de alternar janela executa o atalho Alt+Tab por meio do *PyAutoGUI,* simulando a troca para a próxima janela aberta no sistema operacional.

Para impedir que um único gesto mantido diante da câmera dispare comandos repetidamente em alta velocidade, foram definidos intervalos de espera entre execuções. Cliques usam um intervalo de 0,75 segundo, comandos de rolagem usam 0,25 segundo e a alternância de janela aguarda 1,20 segundo. A quantidade de rolagem também é configurável pela interface, com valores entre 1 e 20 passos, sendo 5 o valor padrão (ver Figura 4).

```python
    def handle(
        self,
        gesture: GestureResult,
        bindings: dict[str, str],
        *,
        automation_enabled: bool,
        move_any_gesture: bool,
        mouse_sensitivity: float = DEFAULT_MOUSE_SENSITIVITY,
        scroll_amount: int = DEFAULT_SCROLL_AMOUNT,
    ) -> str | None:
        if not automation_enabled:
            return None

        try:
            if move_any_gesture:
                self._move_cursor(gesture.cursor_point,
mouse_sensitivity)

            if not gesture.is_known or gesture.confidence < 0.55:
                return None

            action_value = bindings.get(gesture.name)
            action = Action(action_value) if action_value in
Action._value2member_map_ else None
            if action == Action.MOVE_CURSOR and not
move_any_gesture:
                self._move_cursor(gesture.cursor_point,
mouse_sensitivity)

            if action is not None and action !=
Action.MOVE_CURSOR:
                self._run_action_when_ready(gesture.name, action,
scroll_amount)
        except Exception as exc:  # pragma: no cover - depends on
host OS permissions.
            return f"Automação indisponível: {exc}"

        return None

    def _run_action_when_ready(self, gesture_name: str, action:
Action, scroll_amount: int) -> None:
        now = time.monotonic()
        key = f"{gesture_name}:{action.value}"
        cooldown = _cooldown_for(action)
        if now - self._last_action_at.get(key, 0.0) < cooldown:
            return

        pyautogui = self._load_pyautogui()
        if action == Action.LEFT_CLICK:
            pyautogui.click(button="left")
        elif action == Action.RIGHT_CLICK:
            pyautogui.click(button="right")
        elif action == Action.DOUBLE_CLICK:
            pyautogui.doubleClick()
        elif action == Action.SCROLL_UP:
            pyautogui.scroll(clamp_scroll_amount(scroll_amount))
        elif action == Action.SCROLL_DOWN:
            pyautogui.scroll(-clamp_scroll_amount(scroll_amount))
        elif action == Action.ALT_TAB:
            pyautogui.hotkey("alt", "tab")

        self._last_action_at[key] = now
```

Figura 4. Lógica de automação e execução de comandos com PyAutoGUI

A interface gráfica foi desenvolvida com a biblioteca *Tkinter* e é dividida em dois painéis principais. O painel direito ocupa a maior parte da janela e exibe a pré-visualização em tempo real da *webcam*. A mão detectada é sobreposta por um esqueleto colorido composto por pontos de referência (*landmarks*) e linhas de conexão entre as articulações, gerados pela biblioteca *MediaPipe*. Cada dedo é representado por uma cor distinta, facilitando a visualização da pose reconhecida pelo modelo (ver figura 5).

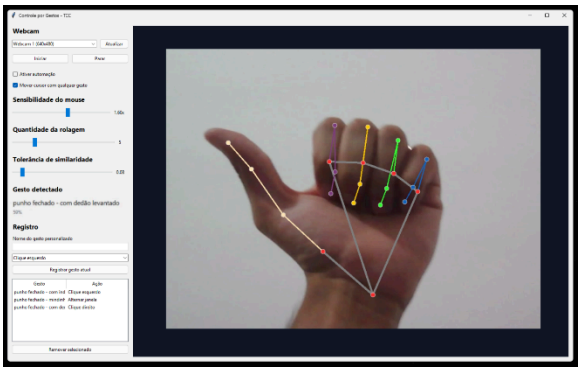

Figura 5. Interface Gráfica

O painel esquerdo concentra todos os controles da aplicação, organizados em seções verticais. Na seção *Webcam*, o usuário pode selecionar o dispositivo de captura por meio de um menu suspenso, que exibe, por exemplo, "*Webcam* 1" e acionar os botões "Iniciar e Parar" para controlar a captura de vídeo, além do botão "Atualizar" para recarregar os dispositivos disponíveis.

São apresentados dois controles de comportamentos globais, como caixas de seleção: 1) Ativar automação, que habilita a execução das ações vinculadas aos gestos; e 2) Mover cursor com qualquer gesto, que permite o rastreamento do ponteiro mesmo quando nenhum gesto específico é reconhecido. Quando essa opção é ativada, apenas reconhecer uma mão pelo vídeo da webcam já é suficiente para iniciar o movimento do cursor. Em seguida, três controles deslizantes permitem a calibração do sistema: 1) Sensibilidade do mouse, que regula a escala de movimento do cursor em relação ao deslocamento da mão; 2) Quantidade da rolagem, que define quantos passos de *scroll* são executados a cada acionamento dos comandos de rolar para cima ou rolar para baixo; e 3) Tolerância de similaridade, que define o limiar de aceitação para o reconhecimento de gestos personalizados. Valores menores na tolerância tornam o reconhecimento mais restrito, enquanto valores maiores tornam o sistema mais permissivo a variações da pose.

A seção "Gesto detectado" exibe em tempo real o nome do gesto identificado pelo classificador (ver na Figura 1), "punho fechado - com dedão levantado" foi o nome dado pelo usuário para o gesto que está sendo identificado, acompanhado de um indicador percentual de confiança da detecção, de 59%.

A seção "Registro" permite ao usuário cadastrar novos gestos personalizados. Um campo de texto recebe o nome do gesto, um menu suspenso lista as ações disponíveis, sendo elas: mover cursor; clique direito; clique esquerdo; clique duplo; rolar para cima; rolar para baixo; alternar a janela.

O botão "Registrar gesto atual" salva a associação entre a pose atual da mão e a ação selecionada. Os vínculos cadastrados são exibidos em uma tabela com as colunas Gesto e Ação, permitindo ao usuário consultar e gerenciar os mapeamentos existentes. O botão Remover possibilita excluir uma associação previamente cadastrada.

As configurações são persistidas em um arquivo *JSON* no perfil do usuário, armazenando os gestos personalizados, as ações vinculadas, a sensibilidade do mouse e o limiar de similaridade. Com isso, o usuário não precisa reconfigurar o sistema a cada nova execução da aplicação.

Foram implementados testes automatizados com a biblioteca *unittest*. Os testes cobrem regras centrais do sistema, como normalização e comparação de gestos, aplicação do limiar de similaridade, escolha do ponto de referência do cursor, controle da sensibilidade do mouse, quantidade de rolagem, carregamento de configurações e comando de empacotamento.

O projeto também possui um comando específico para compilação, executado por meio do UV. O comando utiliza *PyInstaller* para gerar uma versão distribuível da aplicação. Essa etapa é importante porque permite avaliar o software não apenas como script em ambiente de desenvolvimento, mas como uma aplicação desktop que pode ser executada de forma mais próxima ao uso final pretendido.

Dessa forma, o desenvolvimento da aplicação contempla tanto a implementação técnica do *pipeline* de visão computacional quanto aspectos práticos de usabilidade, calibração, segurança de acionamento e distribuição. O resultado é uma base funcional para avaliar a viabilidade do controle de computador por gestos manuais captados por webcam em um ambiente de hardware convencional.

As avaliações foram conduzidas com a colaboração de três voluntários, identificados como U1, U2 e U3, selecionados para compor perfis heterogêneos quanto à proximidade com recursos tecnológicos.

Tabela 1. Perfil dos participantes da pesquisa

Participante	**Idade**	**Experiência tecnológica (auto-avaliada)**
U1	24	8/10
U2	24	6/10
U3	19	4/10

Antes do início dos testes, cada participante recebeu uma orientação verbal sobre os gestos mapeados e a forma correta de posicionamento da mão em relação à webcam. Os participantes foram instruídos a

aproximar a mão da área de alcance da webcam, executar o gesto, retirar a mão da área de captura e repetir o ciclo. Optou-se por não realizar treinamento exaustivo, visando mimetizar a curva de aprendizado espontânea de um usuário iniciante.

A avaliação baseou-se na versão distribuível da aplicação, compilada via PyInstaller, com o intuito de aproximar o cenário experimental das condições reais de utilização final. Manteve-se o distanciamento de 30 centímetros entre o voluntário e o dispositivo de captura. Com cada ciclo de avaliação perdurando cerca de 10 minutos por participante. Os usuários receberam a aplicação com os gestos já cadastrados e foram instruídos a tentar replicá-los a fim de ativar a automação.

Três gestos foram previamente cadastrados pelo desenvolvedor antes da realização dos testes, e os participantes receberam a aplicação já configurada, sendo instruídos a replicar cada gesto de forma a ativar a automação correspondente. Essa decisão foi adotada para garantir uniformidade metodológica entre os participantes, permitindo que os resultados de cada usuário fossem comparáveis entre si. Tal abordagem é viável dado que a função de normalização de *landmarks* da aplicação ajusta a representação vetorial de cada pose em relação ao punho e à escala máxima da mão, tornando o sistema indiferente a diferenças de tamanho entre mãos distintas. Para cada um dos 3 gestos, realizaram-se 30 repetições por participante, totalizando 90 ensaios por gesto e 270 tentativas no conjunto completo. Os gestos avaliados foram: Punho fechado; Punho fechado com o dedo indicador levantado; Punho fechado com o dedo mínimo levantado.

Durante a coleta, foram catalogadas as ocorrências pela interface, distinguindo entre verdadeiros positivos (TP), falsos negativos (FN) e falsos positivos (FP). Situações de inatividade gestual corretamente ignoradas pelo software foram registradas como verdadeiros negativos (TN), seguindo o protocolo de procedimentos de testes.

Os resultados dos testes do gesto 1, punho fechado (ver Figura 6), gesto 2, punho fechado com dedo indicador levantado (ver Figura 7) e gesto 3, punho fechado com dedo mínimo levantado (ver figura 8), punho fechado, identificaram a grande maioria das ocorrências de *True Positive* (Tabelas 2,4,6), com alta precisão (Tabelas 3,5,7).

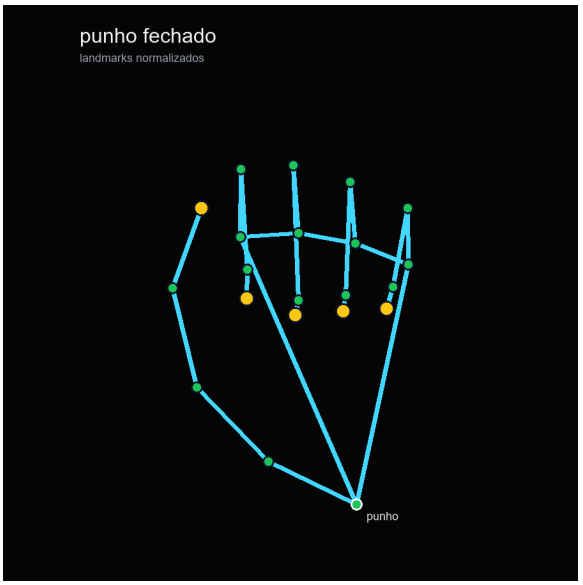

Figura 6. Punho fechado

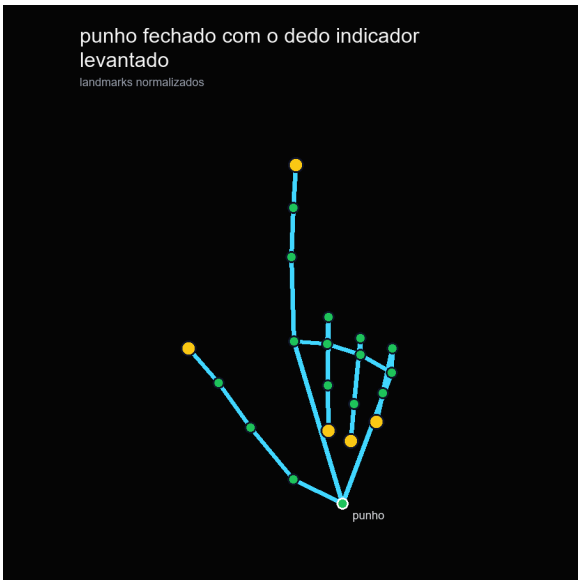

Figura 7. Punho fechado com dedo indicador levantado

Figura 8. Punho fechado com dedo mínimo levantado

Tabela 2. Dados brutos do Gesto 1

Usuário	TP	FP	FN	Total
U1	28	1	1	30
U2	18	2	10	30
U3	24	0	6	30

Tabela 3. Índices desempenho do Gesto 1

Precisão	95,9 %
Revocação	80,5 %
F1-Score	0,88
Taxa de Erro	11,1%

Tabela 4. Dados brutos do Gesto 2

Usuário	TP	FP	FN	Total
U1	29	0	1	30
U2	25	1	4	30
U3	27	1	2	30

Tabela 5. Índices desempenho do Gesto 2

Precisão	97,6 %
Revocação	92,0 %
F1-Score	0,95
Taxa de Erro	5,0%

Tabela 6. Dados brutos do Gesto 3

Usuário	TP	FP	FN	Total
U1	29	0	1	30
U2	17	2	11	30
U3	27	1	2	30

Tabela 7 – Índices desempenho do Gesto 3

Precisão	96,1 %
Revocação	83,9 %
F1-Score	0,90
Taxa de Erro	9,4%

O protocolo adotado incluiu, entre cada tentativa, um intervalo em que o participante retirava a mão do campo de visão da *webcam*. Esses momentos foram monitorados para verificar a ocorrência de reconhecimentos indevidos na ausência de mão detectada. Em nenhuma das 270 transações registradas o sistema disparou reconhecimento em falso, resultando em TN = 90 por gesto e TN total = 270.

Adicionalmente, em nenhuma das tentativas com mão visível o sistema confundiu um gesto com outro: os falsos positivos registrados corresponderam exclusivamente a situações em que o sistema reconheceu o gesto avaliado quando a mão ainda não havia assumido a pose correta, e não a reconhecimentos cruzados entre gestos distintos.

Apresenta-se a comparação entre as métricas.

Tabela 8. Comparativo consolidado de métricas

Gesto	Acurácia	Precisão	Revocação	F1-Score	Taxa de Erro de Classificação

PUNHO FECHADO	88,9%	95,9%	80,5%	0,88	11,1%
PUNHO FECHADO COM O DEDO INDICADOR LEVANTADO	95,0%	97,6%	92,0%	0,95	5,0%
PUNHO FECHADO COM O DEDO MÍNIMO LEVANTADO	90,6%	96,1%	83,9%	0,90	9,4%
Média geral	91,5%	96,5%	85,5%	0,91	8,5%

Assim, o sistema demonstrou desempenho satisfatório, atingindo acurácia média de 91,5%, precisão média de 96,5%, revocação média de 85,5% e F1-Score médio de 0,91 no conjunto dos três gestos avaliados.

O gesto de maior desempenho foi o punho fechado com o dedo indicador levantado, com F1-Score de 0,95 e acurácia de 95,0%. O punho fechado simples apresentou o menor desempenho, com F1-Score de 0,88, acurácia de 88,9% e revocação de apenas 80,5%, o valor mais baixo entre os três gestos, indicando que o sistema deixou de reconhecer o gesto em aproximadamente 1 a cada 5 tentativas. Esse resultado se reflete também no maior número de falsos negativos do conjunto (17 no total). Uma hipótese para esse comportamento é que o punho completamente fechado exige maior precisão na replicação da pose cadastrada, pois pequenas variações no ângulo ou na tensão dos dedos alteram os *landmarks* de forma mais sensível do que gestos com dedos em posição estendida, que constituem referências geométricas mais estáveis.

Em relação à variação entre participantes, U2 apresentou consistentemente os piores resultados nos três gestos, com destaque para os Gestos 1 e 3, nos quais registrou 10 e 11 falsos negativos, respectivamente. Esse padrão sugere que as características individuais de postura ou a forma de executar os gestos de U2 distanciavam-se da referência cadastrada de forma mais acentuada do que os demais participantes, evidenciando que variações individuais de execução constituem um fator relevante para o desempenho do sistema em cenários com múltiplos usuários. Em um cenário real de uso, esse problema tenderia a ser reduzido com a realização do cadastro de gestos pelo próprio usuário, dado que a referência armazenada corresponderia à sua própria forma de execução.

CONCLUSÃO

Desenvolveu-se uma aplicação capaz de permitir a utilização de funcionalidades básicas do computador por meio de gestos com a mão, capturados via webcam e interpretados com o auxílio dos frameworks *OpenCV*, *MediaPipe Hands* e *PyAutoGUI*. A proposta se ancorou na premissa de que ferramentas de visão computacional de código aberto, combinadas com hardware convencional, são capazes de oferecer uma via alternativa de interação humano-computador com potencial de promover acessibilidade e inclusão digital.

Os resultados obtidos na avaliação experimental indicam que o sistema atingiu acurácia média de 91,5%, precisão média de 96,5%, revocação média de 85,5% e F1-Score médio de 0,91 no conjunto dos três gestos avaliados, demonstrando que o *MediaPipe Hands*, mesmo operando com o modelo de menor complexidade computacional e confiança mínima de 60%, é capaz de reconhecer poses da mão com alto grau de confiabilidade em condições controladas de uso.

Entre os gestos avaliados, o punho fechado com o dedo indicador levantado apresentou o melhor desempenho, com *F1-Score* de 0,95 e acurácia de 95,0%, enquanto o punho fechado simples foi o gesto de menor desempenho, com acurácia de 88,9% e revocação de 80,5%. Essa diferença indica a hipótese de que gestos com dedos em posição estendida oferecem referências geométricas mais estáveis para o classificador, ao passo que poses completamente fechadas são mais sensíveis a pequenas variações individuais de execução. A ausência total de falsos alarmes durante os 270 intervalos sem mão visível é um resultado relevante do ponto de vista da usabilidade, pois indica que o sistema não tende a disparar comandos indevidos quando o usuário não está interagindo ativamente.

A variação de desempenho entre os participantes evidenciou um fator limitante importante: quando os gestos são cadastrados por uma pessoa e replicados por outra, diferenças individuais de postura e execução impactam negativamente na taxa de reconhecimento. O participante U2, que apresentou os piores resultados nos três gestos, ilustra esse comportamento. Em um cenário real de uso, pode-se atenuar este problema com a realização do cadastro de gestos pelo usuário que vai usá-los, o que

representa tanto uma limitação do protocolo de testes adotado quanto uma recomendação prática para implantações futuras do sistema.

Do ponto de vista técnico, a arquitetura modular adotada demonstrou ser adequada aos objetivos propostos. A separação entre os módulos de captura, reconhecimento, automação e interface gráfica permitiu que o sistema operasse com estabilidade em hardware doméstico, viabilizando seu uso sem dependência de sensores especializados ou equipamentos de alto custo. A persistência das configurações em arquivo JSON e a disponibilização do software em formato executável compilado via PyInstaller aproximam a aplicação de um cenário de uso real.

Como limitações do trabalho, destacam-se a restrição dos testes a um ambiente com iluminação controlada e fundo neutro e a amostra composta por apenas três participantes jovens com experiência tecnológica acima da média. Essas condições favoráveis podem não se reproduzir em cenários de uso cotidiano, especialmente para o público-alvo de pessoas idosas ou com limitações motoras, para quem a variabilidade de execução dos gestos tende a ser maior. Além disso, o uso prático da aplicação apresenta inconveniências inerentes ao paradigma de controle por gestos que vão além da taxa de reconhecimento. Tarefas que exigem entrada contínua de texto, por exemplo, não são contempladas pela solução proposta, que, apesar de poderem ser contornadas com o uso de ferramentas como o teclado virtual, apresentam uma usabilidade de baixa qualidade e não prática para textos longos, o que restringe sua aplicabilidade a situações de navegação e controle pontual. Esses fatores indicam que a aplicação é mais adequada como recurso complementar de acessibilidade do que como substituto integral do mouse e teclado em sessões longas de uso.

Portanto, recomenda-se ampliar o conjunto de gestos reconhecidos, incorporar testes com usuários pertencentes ao público-alvo de acessibilidade, avaliar o desempenho do sistema em condições variadas de iluminação e fundo, e investigar abordagens de aprendizado de máquina supervisionado como alternativa ao classificador por similaridade geométrica atualmente implementado, com vistas a aumentar a robustez do reconhecimento frente à diversidade de usuários.

.

AGRADECIMENTOS

Á Unimontes.

REFERÊNCIAS

BRADSKI, G. The OpenCV Library. Dr. Dobb's Journal of Software Tools, v. 25, n. 11, p. 120-125, 2000. Disponível em: https://opencv.org/links/ Acesso em Jun. 2025.

BRASIL. Lei nº 13.146, de 6 de julho de 2015. Estatuto da Pessoa com Deficiência. Diário Oficial da União, Brasília, 7 jul. 2015. Disponível em: https://www.planalto.gov.br/ccivil_03/_ato2015-2018/2015/lei/l13146.htm. Acesso em: 1 jun. 2025.

CHEEMA, N.; et al. Predicting Mid-Air Interaction Movements and Fatigue Using Deep Learning. Proceedings of the CHI Conference on Human Factors in Computing Systems, 2020. Disponível em: https://dl.acm.org/doi/10.1145/3313831.3376701. Acesso em: 1 jun. 2025.

INSTITUTO BRASILEIRO DE GEOGRAFIA E ESTATÍSTICA. Pesquisa Nacional por Amostra de Domicílios Contínua: Pessoas com deficiência: 2022. Rio de Janeiro: IBGE, 2023. 15 p. Disponível em: https://biblioteca.ibge.gov.br/visualizacao/livros/liv102013_informativo.pdf. Acesso em: 12 jun. 2025.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO 9241-210:2019 – Ergonomics of human-system interaction – Human-centred design for interactive systems, 2019. Disponível em: https://www.iso.org/standard/77520.html. Acesso em: 1 jun. 2025.

NIELSEN, J.; MOLICH, R. Heuristic evaluation of user interfaces. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 1990. Disponível em: https://dl.acm.org/doi/10.1145/97243.97281. Acesso em: 1 jun. 2025.

ORGANIZAÇÃO PAN-AMERICANA DA SAÚDE. Deficiência. Brasília: OPAS, 2023. Disponível em: https://www.paho.org/pt/topicos/deficiencia. Acesso em: 12 jun. 2025.

SZELISKI, R. Computer Vision: Algorithms and Applications. 2. ed. Cham: Springer, 2022. Disponível em: https://szeliski.org/Book/. Acesso em: 1 jun. 2025.

UDAWI, N.; et al. Innovative healthcare solutions: robust hand gesture recognition of daily life routines using 1D CNN. Frontiers in Bioengineering and Biotechnology, v. 12, 2024. Disponível em: https://www.frontiersin.org/journals/bioengineering-and-biotechnology/articles/10.3389/fbioe.2024.1401803/full. Acesso em: 4 jul. 2025.

WORLD WIDE WEB CONSORTIUM. Web Content Accessibility Guidelines (WCAG) 2.2, 2023. Disponível em: https://www.w3.org/TR/WCAG22/. Acesso em: 1 jun. 2025.

ZHANG, F. et al. MediaPipe Hands: On-device real-time hand tracking. arXiv:2006.10214, 2020. Disponível em: https://arxiv.org/abs/2006.10214. Acesso em: 1 jun. 2025.