ChatGPT

# Guia Técnico: Controle do Humanoide Unitree G1 com Isaac Sim e Isaac GR00T

## 1. Ajuste do URDF (23 DOF) – Fricção, *Damping* e *Armature*

Para melhorar o realismo físico e a estabilidade do robô G1, devemos ajustar o URDF para incluir parâmetros de **fricção nos eixos**, **amortecimento (damping)** e **armature** (inércia do rotor). O URDF do G1 já define 23 DoFs (graus de liberdade) controláveis (pernas, braços e cintura), mas geralmente vêm com mínimos valores padrão de atrito e amortecimento. Podemos editar cada `<joint>` no URDF para adicionar uma tag `<dynamics>` com esses coeficientes. Por exemplo:

```
<joint name="left_knee_joint" type="revolute">
  ...
  <dynamics damping="2.0" friction="0.1"/>
</joint>
```

Esse ajuste adiciona um amortecimento viscoso de 2.0 e atrito estático de 0.1 Nm no joelho esquerdo. Em URDFs, campos padrão como **friction**, **damping** e **armature** são suportados e importados por motores físicos [1]. O **damping** dissipa energia com a velocidade da junta (simula resistência viscosa), enquanto **friction** aplica um torque contrário ao movimento (simula atrito seco). Já **armature** representa a inércia refletida do rotor do motor ou engrenagens no eixo – em simuladores como MuJoCo isso ajuda a suavizar vibrações numéricas. De fato, a documentação do MuJoCo recomenda reduzir o *timestep* ou **"adicionar armature nas juntas relevantes"** para amenizar vibrações, já que **amortecimentos explícitos também podem introduzir feedback instável** [2].

**Dica de implementação:** Comece aplicando amortecimento moderado (p.ex. 1–5 N·m·s/rad) nas juntas das pernas e cintura para reduzir oscilações. Adicione uma pequena inércia (*armature*) às juntas com movimentação rápida (tornozelos, joelhos) – valores baixos (~0.01 kg·m²) costumam estabilizar o controle sem afetar muito a dinâmica. Ajuste também a fricção para evitar deslizamentos não físicos: um atrito de rolamento/torsional de ~0.1–0.2 em cada junta já introduz resistência realista ao movimento [1]. Lembre-se de que no Isaac Sim (PhysX) esses parâmetros são acessíveis via propriedades de *Joint Drive* e *Raw USD Properties*: após importar o URDF para USD, inspecione cada articulção e defina *Joint Friction* e *Armature* conforme os valores escolhidos [3]. Assim, garantimos que o simulador aplique corretamente os torques de atrito e forças de amortecimento em cada passo de tempo.

## 2. Caminhada Natural – Restrição dos Braços e Limites de Juntas

Para obter uma marcha mais **humanoide natural**, convém inicialmente **limitar o movimento dos braços** do G1. Manter os braços abaixados e próximos ao corpo ajuda na estabilidade e simplifica o treinamento do controlador. Tecnicamente, há várias formas de implementar isso:

- **Limites de junta mais restritos:** Podemos reduzir o intervalo de movimento das juntas dos ombros e cotovelos no URDF. Por exemplo, defina os limites de posição do ombro em torno do ponto "braço para baixo". Se originalmente o ombro do G1 podia elevar até, digamos, +90°, podemos limitar para ±10° apenas, restringindo o braço próximo à vertical.

- **Travar juntas com atuador fixo:** Em simuladores, uma abordagem direta é **fixar** os braços usando controladores de posição rígidos. No Isaac Sim, poderíamos temporariamente adicionar uma *Fixed Joint* unindo o braço ao tronco (efetivamente "colando" o braço ao lado do corpo) [4]. Entretanto, isso impede completamente qualquer movimento nos braços – útil para as primeiras tentativas de equilíbrio, mas não ideal para um caminhar natural a longo prazo.

- **Torque ou mola passiva para posição neutra:** Uma solução mais suave é aplicar um **torque passivo que puxe o braço para baixo**. Podemos inserir um termo de controle do tipo `τ = -k * (θ - θ_down)` para cada articulação do ombro, onde θ_down é o ângulo com o braço totalmente baixado. Assim, sempre que o braço se desviar dessa posição, um torque de restauração o traz de volta. Esse comportamento de "mola" pode ser implementado via um **controller** customizado ou até acrescentando um termo de penalidade no aprendizado. Uma técnica citada por praticantes é definir uma configuração articular padrão e **recompensar o robô por manter juntas próximas a essa configuração neutra** [5] – por exemplo, braços apontando para baixo. Essa recompensa faz o agente **aprender a evitar movimentos desnecessários de braço**, servindo como "trela virtual" mantendo-os baixos.

Na nossa abordagem de treinamento, podemos usar um **currículo implícito**: inicialmente **excluímos os braços do espaço de ação**, treinando apenas as 12 juntas das pernas para caminhar. De fato, nos experimentos atuais o *action space* já foi limitado às 12 articulações das pernas [6]. Os braços ficavam parados pela gravidade e um pouco de amortecimento. Confirmamos isso no código: o envio de comandos WASD só afeta `env.commands` lineares e angulares, usadas pelo policy para controlar as pernas, enquanto os braços não recebiam comandos ativos [7]. Com as pernas estáveis, numa segunda fase podemos **liberar gradualmente os braços**: aumentar os limites das juntas ou reduzir os torques de restrição, permitindo ao agente descobrir que pequenos movimentos de braço melhoram o balanço. Essa transição deve ser cuidadosa – por exemplo, reduzindo aos poucos o ganho da "mola" do braço em sessões sucessivas, ou ampliando o range permitido de movimento em etapas. Assim o robô aprende primeiro a andar somente com as pernas e tronco; depois, ao ganhar confiança, passa a **explorar movimentos de braço para melhor equilíbrio** quando liberado. Essa técnica de **curriculum** (do fácil ao complexo) é comum em controle de humanoides. Por exemplo, pesquisadores recomendam inicialmente **punir mudanças bruscas** e **recompensar postura base**, o que naturalmente mantém braços quietos; depois relaxam essas restrições para emergir movimentos naturais [5]. Em resumo, **trave ou restrinja parcialmente os braços nas fases iniciais** (fixação ou forte amortecimento) e **libere-os gradualmente** adicionando graus de liberdade ao controlador assim que a marcha básica estiver estável.

## 3. Mãos no URDF – Massa Inerte agora, Preensão no Futuro

O URDF fornecido do Unitree G1 inclui mãos do tipo “Inspire” (modelo de mãos da Unitree) acopladas aos punhos. Atualmente, **não controlaremos os dedos** – manteremos essas mãos apenas como massas inertes para simular o peso e a inércia dos braços corretamente. Ou seja, as mãos existem no modelo somente para não desequilibrar a dinâmica (mãos ausentes alterariam o centro de massa e o momento de inércia dos braços). Nesse estado, as mãos ficam como objetos rígidos, sem atuadores.

No futuro, entretanto, poderemos **utilizar essas mãos para pegar objetos e interagir com o ambiente**. Existem dois cenários principais:

- **Mãos como garras simples (grippers):** Se as mãos do G1 forem do tipo garra (como uma pinça com abertura/fechamento), podemos integrar um controlador para esse grau de liberdade extra. Bastaria adicionar no URDF as juntas das garras com atuadores e, na simulação, enviar comandos de abrir/fechar. Para pegar um objeto, usualmente se detecta contato nas duas “palmas” ou dedos e, ao fechar a garra, aplica-se uma **força de aperto** suficiente. Muitas simulações simplificam usando um **constraint** de fixação quando um objeto está dentro da garra: o objeto fica parentalmente ligado à mão ao detectar o contato. No nosso caso, poderíamos programar que, ao dar um comando de “pegar”, as juntas das mãos fechem e, se um objeto estiver entre elas (detectado via colisão), prendemos ele à mão na simulação.

- **Mãos dexterosas (vários dedos):** O Unitree G1 tem opções de mãos mais complexas (versão “Ultimate” com até 7 DOF por mão, totalizando 43 DOF no robô [8] [9]). Se no futuro instalarmos mãos antropomórficas, o controle fica mais complexo. Nesse caso, poderíamos integrar algoritmos de grasp planning ou até treinar políticas de RL específicas para manipulação. A vantagem é que nosso pipeline com **Isaac GR00T** já foi pensado para manipulação bimanual: o modelo fundacional **GR00T N1** é capaz de coordenar braços e mãos para agarrar, mover objetos com um ou dois braços e até transferir um item de uma mão para a outra [10]. Ou seja, quando quisermos evoluir para interação manual, poderemos aproveitar a capacidade do **System 1** do GR00T (controle motor fino) para gerar movimentos de dedos e braços coordenados de acordo com a tarefa.

Em resumo, por ora **deixamos as mãos “mortas”** – sem comandos, apenas atuando como peso morto – o que não atrapalha nosso objetivo de locomover o robô. Mas já garantimos que a arquitetura suporta mãos: no URDF elas existem, e o modelo GR00T também foi treinado com cenários de preensão. Quando formos habilitar a manipulação, ajustaremos o URDF para ativar os atuadores das mãos (definindo controladores para os dedos), e então poderemos **emitir comandos de agarrar via linguagem ou script**. Por exemplo, poderemos dar uma instrução em linguagem natural como “pegue a caixa da prateleira”, e o modelo fundacional gerará uma sequência de ações de caminhar até a caixa, estender o braço e fechar a mão sobre o objeto [10]. O potencial está pronto – apenas será necessário desenvolver um módulo de detecção do objeto (visão) e acionar adequadamente as juntas das mãos no momento certo. Felizmente, o GR00T N1.5 já inclui entendimento visual e controle multi-passo para essas tarefas, então reutilizaremos esse recurso quando chegarmos à fase de manipulação.

## 4. Simulação Visual no NVIDIA Isaac Sim e Controle WASD

Atualmente validamos nossas policies no MuJoCo (Sim2Sim do Isaac Gym para MuJoCo). O próximo passo é realizar a simulação com visualização realista no **Isaac Sim (Omniverse)**. O Isaac Sim nos permite importar o robô e ambientes 3D com alta fidelidade gráfica, além de usar física PhysX acelerada por GPU.

**Importando o robô G1:** Como ponto de partida, podemos usar o próprio URDF do G1 (fornecido pela Unitree). O Isaac Sim possui uma extensão de importação de URDF/ROS – no menu do Omniverse, é possível selecionar *Create -> Import USD/URDF* e carregar o arquivo URDF do G1. A ferramenta converterá para o formato USD, preservando articulações, geometrias e propriedades físicas básicas. Após a importação, verifique a hierarquia: devemos ter uma primitiva do tipo `ArticulationRoot` com todas as joints (pernas, braços, etc.). **Nota:** Pode ser necessário ajustar materiais e pesos no Isaac Sim. Caso a conversão não atribua corretamente as massas/inércias, utilize os valores do URDF oficial (ex.: o URDF da Unitree define massas de segmentos e tensores de inércia já calibrados para o G1).

**Carregando o ambiente:** Temos um modelo `.obj` de um escritório que gostaríamos de usar como cenário. Podemos importá-lo no Isaac Sim (através do Create -> Import -> Mesh, selecionando o .obj). Certifique-se de **colidir** o ambiente (marcar objetos estáticos como *colliders* no PhysX) para que o robô possa pisar no chão e esbarrar nos móveis. Se o .obj do escritório não estiver devidamente escalado ou texturizado, podemos começar com algo mais simples (um plano de chão e alguns cubos como obstáculos) e depois evoluir para o cenário realista.

**Controle por teclado (WASD):** Para fins de teste e teleoperação direta, implementaremos controles de teclado no Isaac Sim. Diferente do Isaac Gym (que tinha um loop Python direto), o Isaac Sim tem uma GUI e um subsistema de eventos. Há algumas abordagens comprovadas: - **Usar o ActionGraph do Isaac Sim:** O *Action Graph* possui nós que capturam eventos de teclado. Por exemplo, a NVIDIA fornece um grafo de exemplo para o robô Carter onde setas/WASD controlam velocidade. No nosso caso, poderíamos configurar um nó "Keyboard Event" ouvindo `W`, `A`, `S`, `D` e mapeando para comandos desejados (por exemplo, ao apertar `W`, definir uma variável `cmd_vx = +1 m/s`; `S` = -1 m/s; `A` = +rotação; `D` = -rotação). Esses nós podem alimentar um nó de movimento do robô. Se não quisermos usar a interface visual, podemos escrever um **script Python** dentro do Isaac Sim que captura teclado via **omni.isaac.core** ou a API do kit.

- **Interface Python de controle:** Uma maneira flexível é rodar o Isaac Sim em modo "headless" (ou com a GUI ativa) e usar a *Isaac Sim Python API* para controlar o robô. Por exemplo, podemos usar `omni.isaac.core` para acessar a articulação do G1 e aplicar comandos de posição/velocidade. Montamos um laço: a cada tick de simulação, lemos o estado do teclado (p.ex., usando a biblioteca Python `keyboard` ou integrando com a interface do Omniverse) e atualizamos as metas de velocidade linear/angular do robô. Suponha que nosso *policy* (ou controlador) espera comandos de velocidade `(v_x, v_y, w_z)` no formato já utilizado no Isaac Gym. De fato, no código `play.py` do Unitree foi implementado exatamente isso: as teclas WASD ajustam uma variável de comando de velocidade, que entra nas observações do agente. No trecho abaixo (extraído do nosso módulo de teleoperação), vemos essa lógica:

```
# Mapeamento das teclas para comandos de velocidade (play.py)
env.commands[:, 0] = vx_cmd    # Velocidade linear X (frente/trás)
```

```
env.commands[:, 1] = 0.0       # Velocidade lateral Y (zero para humanóide)
env.commands[:, 2] = wz_cmd    # Velocidade angular Z (yaw)
```

Aqui, `vx_cmd` e `wz_cmd` são valores definidos pelas teclas (W=+VX, S=-VX, A=+WZ, D=-WZ). Esse comando é inserido no vetor de observações do agente, que então calcula as ações das 12 juntas das pernas [7]. **No Isaac Sim**, podemos replicar esse comportamento: por exemplo, crie variáveis globais `vx_cmd` e `wz_cmd` que são modificadas em callbacks de tecla, e a cada frame de simulação chame a função de controle (seja uma policy RL ou um simples controlador) passando esses comandos. Assim, quando **W/S** forem pressionados, o robô andará para frente/trás; com **A/D**, ele girará para esquerda/direita – exatamente como pretendido.

- **Uso do ROS 2 (alternativa):** O Isaac Sim integra nativamente com ROS2. Caso queiramos uma solução robusta de teleop, poderíamos lançar um nó ROS2 de teleoperação (por exemplo, `teleop_twist_keyboard`) que publica mensagens de velocidade (`geometry_msgs/Twist`). Dentro do Isaac Sim, habilitaríamos a extensão ROS e inscreveríamos o robô nesses tópicos, aplicando as velocidades via um controlador de base. Esta abordagem é mais complexa de configurar, mas é *"comprovadamente funcional"* em setups de simulação robótica [11]. Dado que já temos a solução de RL integrada, talvez não seja necessário acoplar ROS por enquanto – podemos ficar com o script Python direto ou ActionGraph para WASD.

**Sincronização e validação:** Depois de implementar o controle, faça testes no Isaac Sim: coloque o robô em posição inicial de pé e **habilite a simulação de física**. Pressione **Play** para iniciar; então use as teclas para comandar. Verifique se o robô responde apropriadamente – se usar a policy treinada, ele deve tentar andar. Pode ser necessário ajustar a escala dos comandos: por exemplo, se `VX_BASE = 1.0 m/s` estiver muito rápido visualmente, reduza para 0.5. No briefing original, diagnosticamos que o parâmetro angular `WZ_BASE = 1.5 rad/s` estava alto demais e causava instabilidade no giro [12]. Portanto, no Isaac Sim podemos já aplicar o valor ajustado (no **Guia de Equilíbrio**, sugerimos `tracking_ang_vel` menor, equivalente a ~0.5 rad/s para A/D [13]).

Resumindo, **é possível controlar o G1 via WASD no Isaac Sim** implementando um loop de leitura de teclado que atualiza comandos de velocidade, tal como fizemos no Isaac Gym. O ponto-chave é que no Isaac Sim temos que **manualmente avançar a simulação e aplicar as ações** – seja chamando `sim.step()` em Python a cada iteração do loop, seja confiando no *timeline* do editor e usando callbacks temporizados. Uma implementação bem estruturada é integrá-la ao **Isaac Lab** (framework de aprendizado da NVIDIA): ele possui dispositivos de teclado que publicam comandos para robôs [14]. Porém, dada a urgência de um protótipo funcional, um simples script dentro do Isaac Sim já serve. Assim teremos um cenário onde vemos o G1 no ambiente 3D (escritório) e conseguimos **dirigi-lo com WASD** para frente, trás e giros – útil para depurar equilíbrio e coletar demonstrações.

*(Nota: Nossa meta final não é ficar pilotando por teclado, mas essa etapa garante que a simulação física e controle básico estão OK antes de avançar para comandos de alto nível.)*

# 5. Integração do Isaac GR00T – Controle Autônomo do G1 em Simulação

O foco principal será agora implementar o controle do nosso humanoide G1 usando o **NVIDIA Isaac GR00T**, o modelo fundacional generalista. O **Isaac GR00T N1.5** (lançado em 2025) é um "cérebro universal" para robôs humanoides, capaz de entender instruções em linguagem natural e traduzir em movimentos corporais coordenados [15] [16]. Diferentemente do nosso controlador PPO específico, o GR00T traz **habilidades generalizadas pré-treinadas** – por exemplo, equilíbrio, andar, girar, e até manipular objetos – aprendidas a partir de um enorme dataset de demonstrações humanas e simulações. Vamos então integrar esse modelo para controlar o G1 no Isaac Sim.

## 5.1 Preparação do Ambiente de Software (Ubuntu 24.04, RTX 4070)

Para rodar o GR00T N1.5 em nosso sistema, precisamos de um ambiente Python com suporte a inferência de redes neurais de grande porte. Siga uma abordagem **passo a passo** comprovada:

- **Instalar PyTorch e dependências:** O GR00T N1.5-3B contém ~3 bilhões de parâmetros e utiliza Transformers de visão e linguagem. Requer o PyTorch ≥ 2.0 com CUDA. Como nosso sistema já tem CUDA 12 e drivers atualizados (RTX 4070 confirmada com driver 575 e CUDA Runtime 12.9 [17]), instale o PyTorch adequado. Por exemplo, no new ambiente Conda Python 3.10, faça: `pip install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cu118`. (Use cu118/cu121 conforme a versão suportada – PyTorch 2.0 e 2.1 suportam Ada Lovelace.)

- **Clonar o repositório Isaac GR00T:** NVIDIA disponibiliza o código no GitHub: `git clone https://github.com/NVIDIA/Isaac-GR00T.git`. Esse repositório contém utilitários para carregar o modelo, scripts de inferência e exemplos de configuração. Entre no diretório e instale em modo editável: `pip install -e .`. Isso resolverá as dependências listadas (bibliotecas HuggingFace, OpenCV, etc.). Alternativamente, use o Docker fornecido pela NVIDIA – há um *Dockerfile* oficial que configura tudo. Mas vamos supor que faremos manualmente via pip para flexibilidade.

- **Baixar o modelo pré-treinado:** O modelo GR00T N1.5 3B está hospedado no HuggingFace Hub, nome `nvidia/GR00T-N1.5-3B`. Podemos baixar automaticamente ao rodar o código (a primeira inferência faz *download*). Porém, para evitar latência, pode-se usar `huggingface-cli` para login (o modelo é aberto, licença NSCL v1) e depois `git lfs` clone ou `wget` do arquivo `.safetensors`. O modelo pesa alguns gigabytes – assegure-se de ter espaço em disco e ~10–12 GB de VRAM livre para carregá-lo (a RTX 4070 possui 12GB, suficiente para inferência de um sample por vez [18]).

- **Configurar o Embodiment (morfologia do robô):** O GR00T suporta múltiplos "*embodiment heads*", isto é, ele pode adaptar a diferentes robôs. Para nosso G1, precisamos escolher o cabeçalho que corresponda. Pelas definições: `EmbodimentTag.GR1` destina-se a humanoides com mãos **dedicadas/destro (dexterous)**, enquanto `EmbodimentTag.AGIBOT_GENIE1` é para humanoides com **grippers simples** [19]. O Unitree G1 básico **não tem mãos altamente articuladas**, apenas uma espécie de *end-effector* simples (sem dedos independentes). Portanto, **usaremos o EmbodimentTag.AGIBOT_GENIE1**, que é exatamente para "humanoide com grippers usando

controle em espaço de juntas" [19]. Esse perfil abrange um corpo bípede com braços de 5 DOF cada e possivelmente um atuador de preensão simplificado – alinhado ao G1 de 23 DOF. Selecionar o *embodiment* correto é importante para que o modelo interprete corretamente o vetor de estado do robô (propriocepção) e gere ações compatíveis (23 comandos de juntas). *(Observação: Se eventualmente ativarmos mãos mais complexas, poderíamos trocar para EmbodimentTag.GR1, mas por ora AGIBOT_GENIE1 é apropriado.)*

- **Teste de inicialização:** Com tudo instalado, escreva um pequeno script para validar. Por exemplo:

```
from gr00t.model.policy import Gr00tPolicy
from gr00t.data.embodiment_tags import EmbodimentTag

policy = Gr00tPolicy(
    model_path="nvidia/GR00T-N1.5-3B",   # HuggingFace model reference
    embodiment_tag=EmbodimentTag.AGIBOT_GENIE1,
    device="cuda"
)
```

Ao rodar, o código acima deve baixar e carregar o modelo na GPU. Isso confirmará que nossa instalação está OK. O tempo de carregamento inicial pode ser de alguns segundos a um minuto. Após carregado, uma única inferência (ação) normalmente é rápida o bastante para controle em tempo quase-real – a NVIDIA relatou que para uma amostra por vez, até GPUs diferentes (L40 vs RTX4090) têm performance similar, indicando que o modelo foi otimizado e pode rodar a alguns Hz sem problemas [18].

### 5.2 Controle do G1 via Groot – Execução na Simulação

Com o modelo carregado, precisamos **conectá-lo ao nosso robô no Isaac Sim**. A arquitetura GR00T usa uma *policy* neuronal que mapeia *observações multimodais -> ações contínuas*. As entradas incluem: **visão** (imagens de câmera do robô), **linguagem** (instrução textual) e **propriocepção** (estado das juntas do robô). A saída são vetores de ação (ângulos ou torques das articulações). Para integração:

- **Configurar observações do robô:** No Isaac Sim, inserimos pelo menos uma câmera presa ao robô (por exemplo, na cabeça ou tronco) para fornecer visão ao modelo. Podemos usar uma câmera RGB frontal de resolução 224x224 para coincidir com os requisitos do modelo [20] [21]. Também coletamos o estado atual: posições articulares, velocidades e possivelmente forças de contato – esses seriam nosso vetor de **propriocepção**. O GR00T N1.5 espera esses dados em um formato específico (*LeRobot* schema): por exemplo, um vetor com ângulos normalizados e talvez orientação do tronco. Precisaremos mapear do estado do Isaac Sim para o formato de *input* do modelo. O repositório fornece exemplos de *ModalityConfig* para diferentes robots; no nosso caso usaríamos a configuração já existente para GR1/AGIBOT.

- **Enviar instrução ao modelo:** Agora, decidimos como comandar o robô. Duas opções de entrada de alta nível: via **WASD** (traduzido em texto simples, ou bypassando linguagem) ou via **linguagem natural direta**. Como o GR00T aceita uma string de instrução, podemos emular WASD enviando comandos textuais como *"andar para frente"* quando W está pressionado, *"vire à esquerda"* para A, etc. Entretanto, acionar o modelo a cada pequena variação poderia ser ineficiente – possivelmente

melhor é dar um objetivo mais constante, como *"ande continuamente para frente"* enquanto W está pressionado, e enviar *"pare"* quando soltar. Uma alternativa simplificada é tratar o modelo fundacional como controlador de alto nível de velocidade: há trabalhos que condicionam a política em um *waypoint* ou *velocity command*, mas o GR00T em si foi treinado para seguir instruções sem exigir um vetor de comando separado. Para começar simples, podemos de fato usar frases fixas correspondendo às teclas. (Lembre que isso ainda exercita o entendimento de linguagem do modelo, o que é desejável). Por exemplo: definimos que `W` = instrução `"walk forward"`, `A` = `"turn left in place"`, `S` = `"walk backward"`, `D` = `"turn right"`. Essas strings serão alimentadas no modelo sempre que uma tecla for detectada. O modelo então planejará movimentos para cumprir aquela ordem.

- **Chamada de inferência:** Com **imagem**, **estado** e **instrução** prontos, montamos um `data_sample` e passamos para a política GR00T. Usando a API do repo:

```
# Supondo já carregado Gr00tPolicy como `policy`
obs = {
   "image": camera_image,    # tensor [N_cameras x 224 x 224 x 3]
   "proprio": state_vector,  # tensor do estado do robô
   "instruction": "walk forward"
}
action = policy.get_action(obs)
```

A função `get_action` do **Gr00tPolicy** realizará internamente o processo de inferência: o **System 2** do modelo (Visão-Linguagem) raciocina sobre a instrução e cenário, e o **System 1** gera movimentos contínuos (juntas) correspondentes [16] [22]. Em implementação, o GR00T usa um *transformer de fluxo* que produz um **chunk de ações** (uma sequência de ações ao longo de alguns passos) ao invés de apenas um instante [23]. Provavelmente o modelo N1.5 produz, digamos, 1 segundo de movimentos discretizados. O código indica que ele *denoiseia* ações iterativamente para atingir a meta [24]. Portanto, `policy.get_action()` pode retornar um array de ações para vários sub-passos futuros. Precisaremos aplicar essas ações no robô: podemos **executá-las em sequência** no simulador. Por exemplo, se recebemos 10 comandos de articulação para os próximos 10 frames, aplicamos um por frame. Antes de acabar, podemos já consultar se a instrução mudou (tecla pressionada diferente ou nova instrução de usuário) e então chamar `get_action` novamente com a nova instrução e estado atualizado.

- **Ciclo de controle no Isaac Sim:** Integramos tudo em um loop principal. Cada iteração (por exemplo, ~20 Hz para ter controle suave):
- Captura imagem da câmera do robô e estado atual das juntas.
- Verifica a instrução do usuário (por ex., mantemos a última até mudar).
- Alimenta o modelo GR00T e obtém um conjunto de ações (ou uma única ação de curto prazo).
- Aplica o(s) comando(s) às juntas do robô no simulador usando um controlador de posição ou torque. No Isaac Sim, podemos usar o **Joint Velocity or Position targets** via *ArticulationController*. Dado que o GR00T aprendeu em simulações, ele provavelmente espera controlar a posição das juntas (como alvo de um PD interno). De fato, no paper KungfuBot usando G1, a policy gerava **target positions para um PD** [25]. Podemos espelhar isso: usar *joint position drives* no Isaac Sim com stiffness alto para seguir as posições recomendadas.
- Avança a simulação alguns steps para executar o movimento.

• Repete o ciclo.

Esse esquema deve fazer o G1 se mover de forma autônoma conforme a ordem dada. Por exemplo, se enviarmos a instrução *"walk forward 2 meters"* (ou simplesmente "walk forward" continuamente), o robô começará a andar para frente no ambiente do Isaac Sim. Se houver objetos no caminho e fornecermos visão, o modelo pode tentar evitá-los ou parar – embora, sem refinamento, inicialmente ele pode esbarrar pois não foi especificamente mandado desviar. Contudo, o modelo fundacional tem noções gerais de obstáculo graças ao treino extenso, então pode mostrar comportamentos emergentes.

Uma forma alternativa de integração, recomendada pela NVIDIA, é usar o **Inference Service** fornecido no repositório. Podemos rodar o servidor de políticas em separado:

```
# Terminal 1 – inicia o servidor GR00T
python scripts/inference_service.py --model-path nvidia/GR00T-N1.5-3B --server
```

Isso carregará o modelo e abrirá um serviço (provavelmente sobre gRPC ou WebSocket). Então, do lado do Isaac Sim (ou outro cliente Python), podemos mandar requisições de estado e receber ações:

```
# Terminal 2 – cliente envia pedido de inferência
python scripts/inference_service.py --client
```

No modo cliente, você adaptaria para enviar o seu próprio pacote de observação. A vantagem desse design é **desacoplar a simulação da pesada inferência** – o Isaac Sim pode rodar num processo, e o modelo noutro, comunicando via rede local [26]. Isso evita que possíveis latências travem a simulação gráfica. Muitos desenvolvedores seguem essa arquitetura (um backend de IA servidor e frontend simulador/real mandando percepções). Em nosso caso, para um protótipo, integrar no mesmo processo é aceitável, mas vale notar essa opção escalável.

**Validação na prática:** Assim que o pipeline estiver montado, faremos um teste simples: instrução "**Stand**" (ficar de pé parado). O robô deve permanecer equilibrado sem cair – o modelo fundacional foi treinado para noções de equilíbrio, então espera-se que mantenha postura. Depois, damos "**Walk forward**" – o G1 deve iniciar uma marcha. Monitoraremos se a marcha é estável e se corresponde a uma caminhada humana (o GR00T aprendeu gaits humanos de seu dataset; espera-se um andar mais natural que o nosso PPO inicial). Vamos observar os braços: inicialmente podemos ter deixado as restrições, mas agora, **permitiremos os braços se moverem**, pois o GR00T provavelmente tentará balançá-los para balancear (isso pode melhorar a estabilidade). Caso ainda tenhamos braços parcialmente presos, podemos liberar para ver o comportamento emergente – lembrando que o modelo foi *pré-treinado* em robôs com braços livres, então para tirar máximo proveito, devemos deixá-los soltos agora.

Em suma, **implementar o Isaac GR00T N1.5** envolve preparar o ambiente de inferência e conectar as entradas (câmeras, estados, comandos) às saídas (ações nas juntas) na simulação. Seguimos as referências da NVIDIA, que fornecem notebooks tutoriais no repositório (por exemplo, *1_gr00t_inference.ipynb* mostra como montar todo o pipeline de inferência [27]). Vamos nos basear nesses exemplos para garantir que estamos usando chamadas corretas. O core é: carregar a policy com o `EmbodimentTag` certo e chamar

`policy.get_action()` passando nosso formato de observação [28] – isso **já traz o poder do modelo fundacional ao nosso robô**, sem precisarmos treinar do zero.

## 6. Comparativo de Modos de Comando: Teleop WASD vs Linguagem Natural

Vale discutir as duas modalidades de controle que teremos disponível e como cada uma funciona tecnicamente no sistema:

**Teleop WASD (baixo nível):** Aqui o humano atua como controlador, enviando comandos instantâneos de velocidade. Implementamos isso antes com sucesso parcial – os comandos W/S (frente/trás) funcionavam, mas A/D (giros) tinham problema de estabilidade [29]. O esquema do WASD é **reativo e contínuo**: a cada frame, lê-se a entrada do teclado e define-se uma velocidade desejada (ex.: `vx_cmd = 1 m/s` para W). Essa velocidade entra no *observation* do nosso policy RL (`env.commands` no código) [7], e a rede neural (treinada via PPO) calcula as 12 ações das pernas que levarão o robô a seguir aquela velocidade. Em outras palavras, nosso modelo PPO funcionava como um **controlador de rastreamento de velocidade** (velocity-tracking) sintonizado pelos parâmetros de *reward*. Por exemplo, tínhamos um peso `tracking_lin_vel = 1.0` e `tracking_ang_vel = 2.5` inicialmente [30], o que levou a oscilações – depois ajustamos para 0.5 no angular [13] para reduzir a agressividade no giro. Essa interface de comando já estava **integrada** no treino e inferência: a Fig. do Guia de Equilíbrio mostra o fluxo `WASD Input → [vx, wz] → env.commands → observations → policy → joint_actions` [31]. Ou seja, **teclas definem metas de velocidade**, o *policy* de baixo nível (PPO) controla as juntas para atingir essas metas e o robô obedece rapidamente.

Vantagens do WASD: é simples, de **baixa latência** (resposta imediata) e totalmente determinístico – o robô faz exatamente o que a pessoa pedir (dentro do possível). Desvantagens: requer que uma pessoa esteja sempre no controle e limita o robô a movimentos predefinidos (não há compreensão de objetivo ou obstáculo – se você mandar andar, ele anda mesmo que haja parede).

**Comandos de Linguagem Natural (alto nível):** Neste modo, queremos que o **robô interprete instruções complexas** e atue por conta própria para cumpri-las. Com o Isaac GR00T, podemos dar ordens textuais como *"Caminhe até a mesa vermelha e pegue a xícara"*. O modelo fará o planejamento internamente: primeiro identificar a mesa vermelha na imagem (usando o encoder visual e a instrução), depois gerar uma série de movimentos – andar até lá, parar, esticar o braço, fechar a mão na xícara, etc. Tudo isso sem a gente especificar manualmente cada passo. Tecnicamente, o GR00T possui um **Vision-Language Model (VLM)** congelado que entende a cena e a tarefa, e um **Action Transformer** que produz a sequência motora [16] [32]. Nosso trabalho, então, é só fornecer a instrução certa e o input sensorial. No exemplo, teríamos uma câmera no ponto de vista do robô vendo a mesa e a xícara; damos o comando textual; o modelo notará *"mesa vermelha"* na visão, calcula que precisa andar ~3 metros, então gera ações conjuntas para pernas avançarem, olhando possivelmente para a xícara, etc.

Em código, em vez de ler teclas a cada frame, teríamos talvez uma interface de **texto** (o operador digita ou fala a frase, convertida para string). Passamos essa string ao modelo apenas **uma vez por tarefa**. Podemos, se necessário, atualizar a instrução dinamicamente – mas idealmente, damos uma instrução de cada vez e deixamos o robô terminá-la. Durante a execução, o modelo recebe feedback visual contínuo, então ele se ajusta: por exemplo, se a xícara se move ou cai, o modelo replanejaria em tempo real (dentro

da capacidade de seu transformer). Esse tipo de controle **não exige intervenção constante** do humano – é **auto-dirigido**. O humano atua mais como um supervisor de alto nível: "vá ali", "faça isso", e o robô decide *como* fazer.

No sistema, precisamos expor uma interface para enviar essas instruções. Podemos implementar um simples prompt no terminal ou GUI: o usuário digita a frase e pressiona Enter, então nosso loop de controle atualiza a variável `current_instruction = "texto do usuário"` e reseta o estado do GR00T (podemos rodar uma nova inferência com a nova instrução). Como o GR00T foi projetado para **seguir comandos em linguagem com alta taxa de sucesso (93% de sucesso em comandos de manipulação no benchmark GR-1)** [33], esperamos que instruções claras sejam atendidas corretamente.

**Código nos repositórios:** Para WASD, já vimos o trecho de código no `play.py` do Unitree Gym [7] – ele mostra exatamente como injetar comandos no ambiente. Para comandos de linguagem com GR00T, o repositório Isaac GR00T traz exemplos de dataset e inferência. Um exemplo de chamada foi mencionado:

```
policy = Gr00tPolicy(..., embodiment_tag=EmbodimentTag.AGIBOT_GENIE1, ...)
action_chunk = policy.get_action(dataset[0])
```

No *notebook* de inferência, eles montam um `dataset[0]` contendo `images, state, text` correspondentes a uma tarefa de pick-and-place e obtêm a saída [34]. Nós faremos similar, mas construindo o *dataset sample* manualmente do estado atual. O importante é que o `Gr00tPolicy` **encapsula toda a complexidade** – nós chamamos `get_action()` e recebemos ações já no espaço de juntas do nosso robô, sem precisar programar cada submovimento [28]. Esse encapsulamento mostra como a comunidade está usando: muitos exemplos na comunidade utilizam exatamente essa classe *policy* para ligar no controlador do robô, seja real ou simulado. A NVIDIA menciona que desenvolvedores já conectaram o GR00T N1 a robôs reais como 1X Neo e Agility Digit [35] [36], o que valida a **funcionalidade do código** – podemos confiar que seguindo a API, as ações produzidas serão compatíveis e eficazes para o G1.

**Conclusão sobre os modos:** O WASD será útil para **teleoperar e coletar dados/demonstrações** ou verificar comportamento de baixo nível (por exemplo, validar se a policy RL ou o modelo fundacional está estável quando segue pequenas ordens). Já o controle por linguagem natural é o objetivo final para **autonomia** – permitir comandos abstratos (*"ande até aquela porta"*) e o robô cumprir de maneira inteligente. No nosso desenvolvimento, **faremos primeiro o WASD funcionar no Isaac Sim** (para assegurar que o robô no PhysX está calibrado, não cai sozinho, responde a comandos básicos). Em seguida, **ativaremos o GR00T com comandos textuais simples** (que podem até imitar WASD como citado, mas via linguagem). Assim teremos as duas opções: podemos sempre tomar controle manual se algo der errado, mas preferencialmente deixaremos o modelo conduzir o robô.

Tecnicamente, manteremos ambos os pipelines disponíveis no código, talvez através de uma chave de modo. Poderíamos ter um loop onde, se um teclado estiver ativo, ele tem prioridade (teleop manual interrompe o auto); se o usuário não apertar nada, o robô espera uma instrução textual para agir. Esse é um esquema comum de *shared control*.

# 7. Implementação do Isaac GR00T N1.5 – Setup e Controle via *Groot* no Ubuntu 24.04

(Dando ênfase, pois é a tarefa maior)

Conforme mencionado, já instalamos as dependências e preparamos o modelo. Aqui detalharemos a **implementação prática passo-a-passo** de todo o sistema de controle G1+Groot em nosso ambiente, consolidando os pontos anteriores:

**Passo 1 – Configurar dependências e workspace:**
No Ubuntu 24.04, dentro do nosso ambiente Conda (por exemplo, `unitree-groot` separado do `unitree-rl` para evitar conflitos de versão), instalamos: - PyTorch 2.x com suporte CUDA.
- Biblioteca HuggingFace Transformers, caso o GR00T use internamente (instalada via `pip install transformers huggingface_hub` se não veio pelo repo).
- Biblioteca *timm* (Torch image models) e *opencv-python*, possivelmente usadas para processamento de imagem no dataset.
- Qualquer extra do `requirements.txt` do repo (por exemplo, `pip install einops safetensors accelerate` etc., conforme listado).

**Passo 2 – Obter o modelo GR00T:**
Como indicado, usar `Gr00tPolicy` com `model_path="nvidia/GR00T-N1.5-3B"` baixa automaticamente o modelo da internet na primeira execução [34]. Certifique-se de ter feito login no HuggingFace (se necessário, embora o modelo seja público). Alternativamente, baixe manualmente o peso *safetensors* e coloque num diretório local, usando então `model_path="path_local/GR00T-N1.5-3B"` para carregá-lo localmente. O repositório indica que o N1.5 3B está disponível publicamente e pronto para uso comercial/pesquisa [37].

**Passo 3 – Montar a cena no Isaac Sim:**
Carregue o USD do G1 (importado do URDF). Posicione-o em pé no chão. Configure a física (gravidade ligada, solver de PhysX com substeps adequados, talvez 60 Hz). Adicione a câmera no robô (attach a `rgb_camera` prim no link da cabeça ou peito, resoluções como 224x224). Importe ou crie o ambiente (chão, etc). Salve a cena. Podemos automatizar isso via um script Python para garantir reprodutibilidade. Exemplo: um Python script usando `omni.isaac.core` API para criar mundo, adicionar robot from URDF, etc., e depois entrar no loop de controle. O *Isaac Sim* permite executar um script no startup (extensão) – faremos isso para rodar nossa lógica.

**Passo 4 – Loop de controle principal:**
Este loop faz: - Ler estado do robô: usando `Articulation.get_joint_positions()` e `get_joint_velocities()`, compor vetor propriocepção. Opcionalmente normalizar ângulos (por exemplo, GR00T pode esperar ângulos em radianos normalizados -1 a 1 conforme limites).
- Capturar imagem da câmera: via `sensor.get_rgb()` ou usando *Viewport capture* do Isaac Sim. Converter para tensor PyTorch e normalizar (modelo espera 224x224 uint8 ou normalizado 0-1).
- Obter instrução: aqui integramos se há uma nova instrução do usuário. Podemos ter uma variável global atualizando por: - Teleop: se detectamos tecla, definimos instruction = correspondente texto simples. - Natural: se o usuário digitou algo, usamos isso. - Ou podemos scriptar uma sequência de testes (ex: automaticamente mandar "andar para frente por 5s"). - **Inferência do modelo:** Montar um objeto de dados

com `modality_config`. O repositório fornece utilitários para isso. Por exemplo, eles usam `ComposedModalityConfig` para preparar transforms (redimensionamento de imagem, etc.) e empacotam numa classe `LeRobotSingleDataset` [38] [39]. Para simplificar inicialmente, podemos chamar `policy.get_action()` passando dicionários ou tensores brutos, já que temos apenas um robô. Se a API exigir um dataset, podemos instanciar um `LeRobotSingleDataset` de tamanho 1 com nossos dados atualizados a cada passo (isso é meio contorno – talvez haja método mais direto). De qualquer forma, chamamos a policy e obtemos `action_chunk`.
- **Aplicar ação:** O `action_chunk` pode ser, por exemplo, uma matriz `[H x N_dof]` (H passos, N_dof=23). Podemos pegar a primeira linha como comando imediato, ou bufferizar as H ações e aplicá-las sucessivamente. NVIDIA sugere que o *flow matching model* produz um chunk que você pode aplicar inteiro até pedir outro [40] [24]. Uma estratégia segura: solicite um novo chunk a cada, digamos, 0.5 segundos, aplicando suas ações a 20 Hz – assim se nada mudar, o robô continua aquela sequência; se o usuário der nova ordem, podemos descartar o restante do chunk e pedir um novo imediatamente.
- Step da simulação: chamamos `simulation_context.step()` ou equivalente para avançar um tick com as ações aplicadas. - Repetir.

**Passo 5 – Testes unitários e depuração:**
Antes de algo complexo, testamos comandos simples: - Instrução "*stand*" (ficar parado): o robô deve permanecer de pé estável. Se cair, pode ser falta de ganho no controlador – aumente a rigidez das juntas ou verifique se a policy entendeu o comando (talvez "stand still" seja melhor frase). - "*walk forward*": robô deve dar passos para frente. Veja se os pés interagem bem com o chão (pode ser preciso ajustar a fricção do pé com o solo no Isaac Sim – aumente o atrito estático do material do pé para ~1.0 para evitar escorregar). Se a marcha estiver muito lenta ou rápida, lembre-se: GR00T foi treinado com vários comprimentos de passo, talvez devamos escalar a instrução ("walk forward quickly" vs "slowly"). Podemos experimentar variações na frase para ver qual produz marcha estável. - "*turn left*": valida giro parado. - *Combinações*: eventualmente, "walk forward and turn left" para ver se ele consegue fazer curva (o modelo deve conseguir interpretar conjunção de tarefas graças ao entendimento de linguagem).

**Passo 6 – Iterar melhorias:**
Com o básico funcionando, refinamos. Por exemplo, integrar múltiplas câmeras (GR00T aceita visão multi-view concatenada [41], mas se o G1 só tem uma câmera frontal, tudo bem). Incluir detecção de conclusão de tarefa: o modelo em tese executa continuamente, então para saber se "chegou no destino", talvez precisamos de algum critério externo (ou confiar que a instrução "até a mesa" ele pára perto da mesa sozinho). Nesse aspecto, possivelmente integraremos futuramente um nível de planejamento externo ou monitoração.

**Resumo:** Ao final dessa implementação, teremos no nosso Ubuntu 24.04 o Isaac Sim rodando com o G1, e um script de controle que ouve **WASD** *ou* comandos de texto, alimenta o **Isaac GR00T** e aplica as ações no robô. Utilizamos abordagens já testadas: o repositório oficial da NVIDIA com `Gr00tPolicy` (citado acima) e o nosso já validado esquema de env.commands do WASD. Assim combinamos **teleop de baixo nível** e **autonomia de alto nível** no mesmo sistema.

## 8. *Fine-tuning* do GR00T para o G1 – Abordagem Técnica e Conceitual

Embora não façamos isso imediatamente, é relevante delinear **como ajustar o modelo GR00T para nossos propósitos específicos** no futuro (fine-tuning). O GR00T N1.5 que carregamos é um modelo **genérico**,

treinado para vários robôs e tarefas. Funciona *out-of-the-box*, mas para obter **performance ótima no Unitree G1** em tarefas particulares, podemos realizar um **pós-treinamento (post-training)** com dados adicionais do G1.

**Por que finetunar?**
Considere que o GR00T N1.5 foi treinado com humanoides diferentes (Agility Digit, Fourier GR-1, 1X Neo etc. estavam no conjunto) [36] [42]. Cada um tem pequenas diferenças de tamanho, massa e dinâmica. O Unitree G1 é semelhante, mas não idêntico – por exemplo, o comprimento de perna ou torque máximo podem diferir. Finetuning permite **especializar** o modelo para o G1, garantindo que as ações geradas sejam 100% compatíveis com os limites e características dele. Além disso, se quisermos que o robô realize uma tarefa específica do nosso interesse (por exemplo, *subir escadas*, ou *dançar capoeira*), poderemos fornecer dados dessas tarefas e refinar o modelo para aprendê-las mais precisamente.

**Como finetunar tecnicamente:**
O repositório oferece scripts automatizados para isso. Em especial, o `gr00t_finetune.py` suporta treinar o modelo em novos dados [43]. A pipeline típica seria: 1. **Coletar dataset de demonstração** no contexto desejado. Pode ser **dados sintéticos** do Isaac Sim: por exemplo, teleoperamos o G1 (usando WASD ou um joystick) para andar até certos pontos, ou usamos a própria policy treinada PPO para gerar trajetórias variadas (essa é uma ideia – usar nossas simulações como geradoras de dados). Salvamos sequências de (vídeo, estados, ações) com anotações de instrução. Por exemplo, gravamos um episódio onde o robô anda em frente 2 metros, e etiquetamos com a instrução "walk forward 2 meters". Precisamos fazer isso para várias situações, incluindo possivelmente pegar objetos se for do nosso escopo futuro.
- O formato deve seguir o **LeRobot compatible schema** que o GR00T usa [44]. Basicamente, pastas com frames de câmera, logs de estados (juntas) e ações aplicadas, mais um arquivo JSON/YAML descrevendo a instrução textual e segmentos. Podemos converter nossos dados para esse formato. O repo fornece um exemplo em `demo_data/robot_sim.PickNPlace` com um *pick-and-place* de referência.

1. **Configurar o treinamento de *fine-tune*:** Escolhemos um *EmbodimentTag*. Se estivermos focando só no G1 e já há um head pré-treinado (AGIBOT_GENIE1), podemos **continuar treinando esse head** com nossos dados (chama-se *post-training* nesse caso). Alternativamente, poderíamos definir `EmbodimentTag.NEW_EMBODIMENT` caso quiséssemos adicionar, por exemplo, o G1 com mãos totalmente ativas (29 DOF) – aí seria um novo head não pré-treinado, que aprenderia a mapear para essa nova morfologia [45]. O repo tem um notebook específico *3_0_new_embodiment_finetuning.md* [46] [47] para essa situação. Provavelmente não precisaremos, pois o G1 se encaixa nos existentes.

2. **Executar o script de fine-tune:** Com os dados prontos e config ajustada, rodamos: `python scripts/gr00t_finetune.py --dataset-path ./dados_meus/G1_set --num-gpus 1 --embodiment AGIBOT_GENIE1` (por exemplo). Existe uma série de argumentos – convém rodar com `--help` primeiro [43]. No nosso caso, usando uma RTX 4070 (12GB), provavelmente precisamos ativar opções de economia de memória. O repositório já menciona: *"Se for treinar em uma 4090, use* `--no-tune_diffusion_model` *para evitar OOM"* [48]. Certamente para a 4070 isso também se aplica. Essa flag congela a parte de difusão (flow model) e treina só camadas de ajuste, reduzindo memória. Outra técnica é usar **LoRA (Low-Rank Adaptation)**, e de fato eles citam que usaram LoRA com 2 GPUs A6000/4090 para melhores resultados [18]. Com uma GPU, podemos ainda fazer fine-tune completo mas com batch pequeno. A recomendação da NVIDIA é **aumentar o batch size ao máximo e treinar ~20k steps** para convergir [49]. Talvez na 4070 um batch de 1-2 seja o máximo – compensaremos treinando mais steps ou reduzindo a taxa de aprendizado.

3. **Verificar resultados:** Após o fine-tune, obteremos um modelo ajustado (pesos atualizados). Podemos salvar como um novo checkpoint (por exemplo, *GR00T-N1.5-G1-custom*) e usar esse no `Gr00tPolicy` em vez do original. Esperamos ver melhorias: movimentos mais calibrados, sucesso maior em tarefas específicas testadas.

Um exemplo conceitual: suponha que queira que o G1 ande com passos mais curtos porque nosso escritório tem espaços apertados. Poderíamos gerar demonstrações de passo curto no sim e treinar o modelo a preferir esse estilo para comandos de andar. Assim, ao dizer "ande até a porta", o modelo tunado no G1 talvez use passos contidos, enquanto o modelo genérico talvez desse passos largos (pensando num robô maior como Digit). Fine-tuning ajusta essas sutilezas.

**Considerações finais de fine-tuning:**
- **Continual Learning:** Poderemos iterar: conforme coletemos experiências do robô (inclusive falhas ou sucessos), podemos incorporar ao dataset e continuar refinando. É importante não "esquecer" as habilidades gerais – técnicas como *low-rate fine-tuning* ou LoRA ajudam a **preservar o conhecimento prévio** enquanto ajustam a nova habilidade.
- **Realismo vs Simulação:** Podemos inicialmente finetunar totalmente em simulação (*sim2sim*), mas para transferência ao robô real (no futuro), idealmente incluiremos alguns dados reais (mesmo que poucos, usando *domain randomization* e *post-training* com real data). O GR00T foi projetado exatamente para absorver **pequenas quantidades de dados reais** e adaptar a sim->real [50] [51]. Portanto, no longo prazo, capturar logs do G1 real e finetunar o modelo com eles seria o passo derradeiro para desempenho ótimo no hardware.

Em conclusão, o fine-tuning do GR00T para nossos propósitos envolve preparar um dataset representativo das situações que nos importam, usar as ferramentas do repositório (scripts/notebooks) para treinar com parâmetros adequados à nossa GPU, e depois validar o ganho de desempenho. Conceitualmente, **é um processo de especialização de um modelo fundacional** – similar a ajustar um modelo de linguagem grande para um domínio específico. E graças à arquitetura do GR00T, isso é viável mesmo com recursos limitados: eles demonstram que com poucas demonstrações e algumas horas de treinamento, um humanoide pode aprender tarefas novas adaptadas [35].

Para o escopo imediato, não realizaremos o fine-tune – utilizaremos o modelo pré-treinado como está, que já deve melhorar muito nosso controle de caminhada e equilíbrio. Mas mantivemos o pipeline preparado para, assim que quisermos dar o próximo passo (por exemplo, ensinar o G1 a chutar uma bola ou subir escada), coletarmos dados e refinarmos o "cérebro" GR00T do nosso robô.

---

**Referências Utilizadas:**


- Unitree G1 – Especificações de DOFs e configurações do robô [9] [52].
- Documentação NVIDIA Isaac Sim e Isaac Lab (Tutorial de configuração de atuadores e captura de eventos) [53] [14].
- Códigos e guias do projeto G1 (WASD Teleop e Equilíbrio) para integração de comandos e parâmetros de RL [7] [13].
- NVIDIA Isaac GR00T – Anúncios, repositório e manual de uso do modelo N1.5 (embodiments suportados, pipeline de inferência e fine-tuning) [34] [19] [54].

- Discussões de implementação de física (MuJoCo, Pinocchio) confirmando suporte a parâmetros de dinâmica em URDF [1] [2].
- Dicas da comunidade para controle de humanoides (ex: restringir movimentos espúrios das juntas) [5].
- Capabilidades do modelo fundacional em tarefas de manipulação e mobilidade [10] [33].

---


[1] Future Plans for Motor Friction Parameters (friction, damping, rotorInertia, rotorGearRatio, armature)? · stack-of-tasks pinocchio · Discussion #2435 · GitHub
https://github.com/stack-of-tasks/pinocchio/discussions/2435

[2] Modeling - MuJoCo Documentation
https://mujoco.readthedocs.io/en/stable/modeling.html

[3] [4] [53] Tutorial 13: Rigging a Legged Robot for Locomotion Policy — Isaac Sim Documentation
https://docs.isaacsim.omniverse.nvidia.com/latest/robot_setup_tutorials/tutorial_rig_legged_robot.html

[5] Reinforcement learning based walking on our open source humanoid : r/robotics
https://www.reddit.com/r/robotics/comments/1lrr3tw/reinforcement_learning_based_walking_on_our_open/

[6] [12] [29] [30] Briefing_Especialista_G1_WASD_Groot.md
file://file-2hnJ22VyZqw4jyn6t2cQRx

[7] [13] [31] Guia_Treinamento_Equilibrio_G1.md
file://file-1BukfwNbLBoKKUDdLwXbbq

[8] Unitree G1 EDU Ultimate A Robotic Humanoid (U3) - RoboStore
https://robostore.com/products/unitree-g1-edu-ultimate-a-robotic-humanoid?srsltid=AfmBOoomXJQEfunzF9WRA08uq4nm027oUkPxxT7gGy6ypjsAkthPV8Cx

[9] Unitree G1 Basic - Toborlife AI Robots
https://toborlife.ai/g1/?srsltid=AfmBOorUhCW7jCFBjdInx4NGH2vlN5_8MBKhgv8zxslvZKnRHasNCs9t

[10] [15] [16] [22] [35] [36] NVIDIA Announces Isaac GR00T N1 — the World's First Open Humanoid Robot Foundation Model — and Simulation Frameworks to Speed Robot Development | NVIDIA Newsroom
https://nvidianews.nvidia.com/news/nvidia-isaac-gr00t-n1-open-humanoid-robot-foundation-model-simulation-frameworks

[11] Control Your Robot Using Command Line with NVIDIA Isaac Sim ...
https://medium.com/@kabilankb2003/control-your-robot-using-command-line-with-nvidia-isaac-sim-and-ros-2-6b315c84a7da

[14] isaaclab.devices — Isaac Lab Documentation
https://isaac-sim.github.io/IsaacLab/main/source/api/lab/isaaclab.devices.html

[17] Unitree_G1_Simulacao_Guia.md
file://file-LP7gmwAtav3a4oL9aPETb1

[18] [19] [26] [27] [28] [32] [33] [34] [38] [39] [42] [43] [44] [45] [46] [47] [48] [49] [54] GitHub - NVIDIA/Isaac-GR00T: NVIDIA Isaac GR00T N1.5 is the world's first open foundation model for generalized humanoid robot reasoning and skills.
https://github.com/NVIDIA/Isaac-GR00T

[20] [21] [23] [24] [37] [40] [41] nvidia/GR00T-N1.5-3B · Hugging Face
https://huggingface.co/nvidia/GR00T-N1.5-3B

25 KungfuBot: Physics-Based Humanoid Whole-Body Control for Learning Highly-Dynamic Skills
https://arxiv.org/html/2506.12851v1

50 51 Enhance Robot Learning with Synthetic Trajectory Data Generated by World Foundation Models | NVIDIA Technical Blog
https://developer.nvidia.com/blog/enhance-robot-learning-with-synthetic-trajectory-data-generated-by-world-foundation-models/

52 Unboxing the FUTURE! Unitree G1 Humanoid Robot - YouTube
https://www.youtube.com/watch?v=j34pfoXzoA8