ChatGPT

Implementing Thruster-Assisted Cassie Jumping with PPO – Step-by-Step Guide

In this guide, we'll adapt the methodology from the 2023 *Cassie Jumping* paper to a Cassie robot in MuJoCo that has been modified with left and right thrusters. The goal is to enable stable hovering and dynamic jumps using reinforcement learning (PPO). We will cover how to extend the observation/action space for thrusters, set up a multi-stage training curriculum (initial stabilization, jumping, recovery), shape rewards for thruster usage, jump height, and landing stability, modify the simulation for thruster forces, ensure sim-to-real robustness, and integrate these changes into a PPO training loop. Each step includes technical details and code snippets to align with your existing `cassie_env.py`, `cassie.xml`, and PPO pipeline.

Step 1: Extend Observation and Action Spaces for Thrusters

Add Thruster Actions: Your Cassie model originally has 10 motor actuators; with two thrusters, increase the action dimension to 12. In the environment initialization, define the total number of actuators and update any action bound arrays accordingly. For example:

```
# In CassieEnv.__init__
self.num_motor = 10      # existing motors
self.num_thruster = 2    # two new thrusters
self.num_actuator = self.num_motor + self.num_thruster  # now 12 actuators
# Expand action bounds to include thruster limits
self.action_space = gym.spaces.Box(low=-1.0, high=1.0,
shape=(self.num_actuator,), dtype=np.float32)
```

In your provided `cassie_env.py`, we see that `num_actuator` is set to 12 and the action bounds array is updated for 12 dimensions [1]. This ensures the PPO policy outputs a 12-dimensional action vector (10 joint controls + 2 thruster controls).

Thruster Action Range: Decide how thruster commands map to force. If thrusters only provide upward force (no negative thrust), you might define the thruster action range as [0, 1] (or [-1,1] but clamp negatives to zero). For example, an action of 1.0 could correspond to full throttle (some maximum Newtons of thrust). It's often useful to scale actions to physical units. For instance, if maximum thrust is 200 N per thruster, scale the normalized action into that range when applying it in simulation. Ensure the action space reflects this (e.g., normalized -1 to 1 mapped to 0–200 N or similar). You can handle scaling in the environment's step function when applying forces.

Include Thruster State in Observations: To help the policy account for thruster dynamics, augment the observation space with thruster-related state. A simple approach is to include the *last thruster action outputs* in the observation vector. This informs the policy of current thruster levels (since thrusters may have ongoing effect like upward acceleration). For example, if your observation originally is `[joint angles,`

`joint velocities, orientation, etc.]`, you can append `[thruster_left_command, thruster_right_command]`. In code, after processing the usual robot state, do:

```
obs = np.concatenate([base_obs, self.last_thruster_cmd])
```

where `self.last_thruster_cmd` is a 2-length array storing the last applied thruster actions (or filtered thrust values). Your environment already keeps track of `last_acs` (last action) which includes thrusters 2 3, so you can utilize that. Including previous actions is a common trick to provide the policy with a short-term history in a memory-less MLP. This is especially useful for dynamic tasks like jumping where the effect of a thruster action extends over a few time steps.

Sensor Signals: If available, also consider adding any thruster-specific sensor readings (e.g., an IMU might detect upward acceleration when thrusters fire). However, in most cases the existing sensors (joint velocities, body acceleration, orientation) will indirectly reflect thruster effects. No dedicated thruster sensor is strictly necessary if you include the commanded thrust as above.

Verify Spaces: After changes, verify that your `observation_space` and `action_space` in the environment match the new dimensions. For example, the observation length should increase by 2 (for thrusters) if you add those values. You can print out the shape to confirm:

```
print("Observation dim:", env.observation_space.shape, "Action dim:",
env.action_space.shape)
```

This extension of state/action space lays the groundwork for the policy to **control thrusters along with leg motors**.

Step 2: Implement Thruster Forces in the Simulation (cassie_env.py)

With the action space extended, we need to actually apply thruster forces in the MuJoCo simulation each time step. In your `cassie_env.py`, you will integrate thruster commands into the physics step. Based on your snippet, you plan to apply forces directly to the thruster bodies each control loop iteration. This approach is valid. Here's how to implement it:

Add Thrusters to the XML: In the MuJoCo XML (`cassie.xml`), ensure you have bodies or sites representing the thrusters (e.g., `thruster_left` and `thruster_right`). They might be attached to Cassie's pelvis or another appropriate link. For example, you might have something like:

```
<body name="thruster_left" pos="0 0.1 0.0" mocap="false">
    <geom type="sphere" size="0.05" rgba="0 1 0 0.5"/>
</body>
```

attached to the main body with a weld (so it moves with Cassie). The exact XML depends on how you modeled the thrusters. The key is that these bodies exist and are named, so we can apply forces to them in

the code. (If you have instead defined thrusters as actuators in XML, you could use `sim.data.ctrl` for them – but here we'll assume we apply forces manually, since your snippet shows that approach.)

Apply Forces Each Step: Within the environment's step function (or the low-level simulation loop), after computing the motor commands, apply the thruster forces. From your code, we see you do something like:

```
# In CassieEnv.step or similar:
thruster_actions = actual_pTs_filtered[10:]
if thruster_actions[0] > 0:
    self.sim.apply_force([0, 0, -thruster_actions[0], 0, 0, 0], "thruster_left")
if thruster_actions[1] > 0:
    self.sim.apply_force([0, 0, -thruster_actions[1], 0, 0, 0],
"thruster_right")
```

This matches your snippet [4]. Here, `apply_force([fx, fy, fz, tx, ty, tz], body_name)` applies a force and torque to the given body. We apply a force in the negative z-direction (downwards in world frame) on the thruster bodies, which produces an upward thrust on the robot (equal and opposite reaction). The condition `if thruster_actions[i] > 0` ensures we only apply upward force (no negative thrust). This effectively treats the thrusters as one-directional jet engines.

A few implementation details and tips:

- **Unit Scaling:** Ensure that the values in `thruster_actions` are in Newtons. If your policy outputs normalized values (e.g. -1 to 1), convert them to Newtons by multiplying by the max thrust. For example, if `thruster_actions` comes in as 0.5 (50% throttle) and max thrust is 200 N, you'd apply 100 N. You might handle this in the code that generates `actual_pTs_filtered` or just before calling `apply_force`. Since your code normalizes actions by 100 (suggesting inputs might be scaled), double-check that the thruster portion is scaled appropriately. You might explicitly define a `max_thruster_force` constant and do:

```
force_left = max_thruster_force * np.clip(thruster_actions[0], 0, 1)
```

and then use `force_left` in `apply_force`.

- **Application Frequency:** Apply the force continuously throughout the control step. In your code, you loop `for _ in range(self.num_sims_per_env_step)` and call `sim.step_pd_without_estimation` for motors, then `apply_force` inside that loop [5]. This means the thruster force is applied at each small integration step (e.g. 2000 Hz), effectively making it a continuous force during that control interval. This is good for stable simulation of thrusters. Keep this loop structure – it ensures the thruster force is applied consistently, rather than just once per policy step, which could otherwise act like a short impulse.

- **Thruster Placement and Effect:** Make sure the thruster bodies are positioned to produce the desired torque on the robot. For instance, if the thrusters are mounted symmetrically on the left and right of the torso, equal thrust will generate upward force, and a difference in thrust will create a roll

torque (helping stabilize roll). If the thrusters are placed fore and aft, they'd affect pitch. *Verify the orientations:* If the thrusters are meant to point downward, the force vector should indeed be applied downward in world coordinates (as above). If your simulation uses local coordinates for `apply_force`, you might need to adjust the direction. In MuJoCo's `mj_applyForce` (if using the C API via Python), forces are in world frame by default unless specified otherwise.

- **Include Thruster Dynamics (Optional):** If you want more realism, you can model thruster dynamics such as a slight lag in thrust (e.g., first-order lag or ramp). A simple way is to low-pass filter the thruster command. It looks like you are already filtering actions with `self.action_filter.filter(actual_pTs)` [6]. Ensure this filter also affects thruster commands (it appears it does, since you split thruster actions *after* filtering). The Butterworth filter you imported will smooth rapid changes [7]. This prevents unrealistic instant thrust changes and helps training stability. You can also manually limit the rate of change of thruster output if needed.

- **Update Environment State:** If you maintain any internal state for thrusters (like fuel or temperature), update it each step. In most cases we can ignore these complexities, but you might simply reset thruster state in environment reset. For example, set `self.last_thruster_cmd = np.zeros(2)` on reset, as seen in your code where `last_acs[10:] = 0.0` on reset [2].

With these modifications, the simulation will correctly respond to thruster actions. **Test the thrusters in simulation** to ensure they behave as expected: for example, apply a constant thruster action and verify Cassie's COM accelerates upward and possibly tilts if one thruster is stronger. Tuning of maximum thrust may be required. A too-powerful thruster can launch Cassie uncontrollably, while too weak might not aid jumping. Aim for thrusters that can meaningfully assist the jump (perhaps providing a lift equal to a significant fraction of Cassie's weight, e.g. each thruster ~30-50% body weight), but not so strong that the policy learns to rely solely on thrusters to "fly" (unless that's intended). You can adjust this by scaling the action-to-force mapping or capping the thruster action inputs.

Step 3: Structure a Multi-Stage Training Curriculum (Stabilization, Jump, Recovery)

Training Cassie to jump and land is challenging, so following a **multi-stage curriculum** is crucial [8] [9]. We will break training into stages, each with its own focus and progressively harder objectives:

Stage 1 – Initial Stabilization (Hover/Balancing): The first stage focuses on basic stability. Here, the robot learns to stand upright (and potentially hover slightly) without falling. We *do not* ask for a full jump yet. The goal is to have a policy that can maintain balance, handle minor perturbations, and become comfortable with the thrusters in a low-stakes setting.

- **Environment Setup:** Initialize the robot in a nominal standing pose. No jump command is given (if your environment uses a goal `c`, set it to zero movement: e.g., target location = same spot, target height = 0). You may disable or heavily penalize thruster usage in this stage so that the policy first learns to use the legs (and only minor thruster corrections). Alternatively, allow small thruster outputs to see if it uses them for balance.

- **Rewards:** Reward staying upright and penalize falling. For example, you can give a reward at each step for "pelvis height roughly equals standing height" (encouraging the robot not to crouch or collapse) and for keeping pelvis orientation level (minimal roll/pitch angle). Also provide a small **alive bonus** each step the robot remains standing, to encourage longer runs. Penalize large control inputs to discourage thruster overuse or jitter. Essentially, the policy should learn a **stable hover/stand**. If thrusters are powerful enough to lift the robot's weight, you might allow a slight hover (feet barely off ground) and reward maintaining a certain height, but this is optional – the main point is balance.

- **Early Termination:** End the episode if the robot falls over (e.g., if pelvis height drops below a threshold or if orientation deviates too much). This prevents the agent from getting used to lying on the ground. In Stage 1, since no jumping is required, you can set a relatively long episode length (e.g., 5–10 seconds) so the robot practices prolonged balancing.

By the end of Stage 1, you should have a policy that keeps Cassie upright in place, using mainly leg actuators (and maybe slight thruster firings if needed for balance). This will serve as initialization for the jump learning. Cassie Jumping researchers found that learning to jump *from scratch* is very hard – a single-stage approach often failed to even achieve takeoff [10]. The multi-stage strategy mitigates that.

Stage 2 – Jump Execution (Takeoff and Flight): In the second stage, we train the policy to perform the jump itself. We leverage the stability learned in Stage 1 and now introduce the **jumping task**. Initially, start with a simple jump (e.g., jump in place vertically) before attempting more complex jumps.

- **Environment and Goal:** Configure the environment to command a jump. For instance, if you have a goal vector `c = [c_x, c_y, c_z, c_ϕ]` (as in the paper), for Stage 2 you might set `c_x = c_y = c_ϕ = 0` (no horizontal displacement or turning) and only `c_z` (vertical target) to a certain height. Essentially, **jump in place** is the first objective [11]. You can choose a moderate target height (e.g., 0.3 m or 0.5 m above the ground) or simply reward "jump as high as possible" if no explicit target. Providing a target height as part of the observation (goal) is recommended so the agent knows what it should achieve [12].

- **Policy Initialization:** Initialize this stage's policy with the weights from Stage 1 (the balanced standing policy). If you're using the same PPO training code, load the Stage 1 model parameters so that learning starts from a stable behavior.

- **Rewards for Jumping:** The reward design is critical to encourage a proper jump. In this stage, **task completion** should be emphasized more. Key reward components:

- **Takeoff (Jump Height)**: Reward the robot for achieving upward motion. You can use the robot's center-of-mass or pelvis height: for example, give a reward proportional to the maximum height achieved or for exceeding a threshold height. If you have a target height (e.g. `c_z`), reward the proximity to that height at the apex. A simple shaping is `r_height = exp(-|achieved_height - target_height|)` or even a binary success bonus if above threshold. This drives the policy to actually leave the ground.
- **Foot Clearance**: Alternatively or additionally, ensure the feet leave the ground. The Cassie paper used a reference foot trajectory and would terminate the episode if the foot didn't reach a minimum clearance [13]. You can implement a term that measures the distance of each foot off the ground

during flight and reward foot clearance above some value (or early-terminate if not achieved by a certain time). This prevents the policy from cheating by shuffling without true jump.

- **Orientation Control (Flight Stability)**: Reward maintaining a good orientation during the jump. For instance, penalize large body tilt or rotation unless it's part of the task. A level body in flight is generally desired for safe landing. As in the paper, even if the task doesn't explicitly require a certain orientation, adding a reward for keeping roll and pitch near zero can help stabilize the pelvis [14].
- **Thruster Usage**: In Stage 2, thrusters become important to achieve height and stability. Encourage the use of thrusters **up to a point**. You should still include a **penalty on excessive thruster use** (to mimic fuel/energy cost), but the weight can be lower than in Stage 1. For example, add a term like `r_thruster = -α * (F_left^2 + F_right^2)` where F are the instantaneous thrust forces and α is a small coefficient. This is analogous to penalizing motor torques for energy efficiency [15], extended to thrusters. The penalty ensures the agent uses thrusters judiciously rather than maxing them out constantly, but it won't outright forbid them. Calibrate α so that using thrusters is not worse than failing the jump – the agent should still prefer to fire thrusters to achieve the height if needed. (You can gradually decrease α over training iterations if you find the agent is too thruster-averse.)

- **Smooth Control**: Jumping is an explosive action, but we still want to avoid extremely jerky control that could destabilize the robot. Introduce an **action smoothing** reward component: penalize large changes in action between consecutive timesteps. For example, `r_smooth = -β * ||a_t - a_{t-1}||^2`. This should include thruster actions as well. Cassie's jumping policy included a "change of action" penalty in later stages to smooth out aggressive maneuvers [16]. This will help prevent the thrusters from flickering between max on/off in successive steps and similarly smooth the leg motion.

- **Phased Reward Scheduling:** During a jump, the priorities before takeoff vs after landing differ. The Cassie paper handled this by splitting the episode into two phases: **pre-landing (jump phase)** and **post-landing (stabilization phase)** with different reward weightings [17]. You can do the same:

- *Before landing* (while the robot is in the stance and flight phase up to the apex and just before touchdown), emphasize rewards for powerful takeoff (tracking target height/distance, foot clearance) and less on staying still (because we want it to move).
- *After landing* (once feet hit ground), emphasize stability: reward the robot for coming to a stop, standing upright, minimal oscillation, etc., and heavily penalize falling. For instance, after landing, increase the weight on the thruster penalty and joint velocity/torque penalties to encourage a gentle touchdown and quiet stance [15]. The Cassie reward had a high weight on minimizing torques and joint velocities after T_J (a preset jump duration) to enforce a stationary landing pose [16].

In practice, you can hard-code a switch at a certain timestep T_J (the notional time by which a well-executed jump should have landed). For example, if a typical jump lasts ~1 second in simulation, set $T_J = 1.0$ sec (or corresponding timesteps). For $t < T_J$, focus on jump goals; for $t > T_J$, focus on balancing. This does not need to be perfectly tuned – the policy can handle variable actual flight times (the paper notes the fixed phase time still worked across different jump distances [18]). The key is to ensure the reward function "shifts gears" from *explosiveness* to *stability* at landing.

- **Episode Termination:** End the episode if the robot falls or if it diverges too far from the target. For example, if it was supposed to jump in place but lands far off, or if it's clearly out of control (lying on ground). Also consider early termination if the robot hasn't initiated a jump after some time (to avoid

wasting time in local optima of not jumping). For instance, if within the first 1–2 seconds the feet haven't left the ground by a small threshold, terminate with a poor reward. This forces the policy to attempt the jump rather than just standing (which by Stage 2 it might otherwise do since it learned standing is safe). The paper used a foot height error tolerance to trigger early resets [13] – effectively, *jump or get zero*. This kind of tough love helps overcome the inertia of doing nothing.

Stage 2 training will likely be the most involved. Over training, the agent should learn to coordinate a jump: bending legs, firing thrusters at the right moment to boost upward, tucking legs if needed, and preparing for landing. Monitor progress by looking at metrics like max height achieved and whether the robot sticks the landing upright. If it often falls on landing, you may need to tune rewards to emphasize stability more (or move to Stage 3).

Stage 3 – Landing Recovery and Robustness: The third stage aims to refine the landing and make the policy robust to variations (including those needed for sim-to-real transfer). Here we introduce more randomness and challenge the policy to recover from imperfect conditions.

- **Environment Variations:** Enable **domain randomization** and perturbations in this stage (more on this in Step 5). Randomize physical properties (mass, inertia, friction, motor strengths, thruster power, etc.) each episode [19], and optionally apply random external pushes or disturbances at landing. The idea is to expose the policy to a variety of conditions so that it learns to generalize and can handle unexpected events (like a slightly slippery floor or a thruster that's a bit weaker than expected). In the Cassie jumping project, Stage 3 added extensive dynamics randomization while continuing the multi-goal training [20].

- **Multiple Jump Goals:** At this point, you can also expand the variety of jump tasks if desired. Instead of only jumping in place, randomize the goal c each episode (or even within an episode). For example, sample different jump distances, directions, and heights. One episode the robot might need to do a long forward jump, the next a high in-place hop, the next a small sideways hop. The policy is conditioned on the goal vector c, so it should adjust its behavior when you vary these inputs [12] [21]. In training, you can randomize c at the start of each episode (Stage 2 was single-goal, Stage 3 is multi-goal). This *multi-goal training* forces the policy to become versatile and not overfit to one jump type [11]. It also tends to improve robustness: as shown in the paper, training on a diverse set of maneuvers let the policy reuse those maneuvers to recover from perturbations or bad landings [22] [23]. (If multi-goal training is too complex to implement, you can still do Stage 3 with just the in-place jump but with heavy randomization. However, incorporating at least a couple of different jump targets is beneficial.)

- **Rewards:** Use a reward structure similar to Stage 2 but now with even more weight on *successful landing* and *recovery*. The task completion aspect now covers potentially different goals (reward reaching the commanded distance/direction). Continue to penalize thruster overuse and encourage smooth actions. One specific addition: you might give a bonus for a **clean landing** – e.g., if the robot manages to land within some tolerance of the target point and remain upright for a certain time, give an extra reward. Also, **time extension bonus** can be used: if the robot is still up and balanced at the end of the episode (which might be a few seconds after landing), that itself is a reward (since it means success). Essentially, surviving without falls *is* a reward in episodic return because the robot accumulates alive bonuses. Make sure the alive bonus or stability reward after landing is significant enough that the agent prefers a controlled landing over a high-but-crashy jump. In practice, you

might increase the weight on orientation and zero-velocity after T_J further in this stage, because domain randomization might induce oscillations [16].

- **Episode Design:** It can be useful to allow *multiple jumps per episode* at Stage 3, to really test recovery. For example, let the robot land from one jump, stand for a random delay (say 1–5 seconds), then provide a new jump command and have it jump again in the same episode. This was done in the paper: they ran very long episodes (up to 2500 time steps, ~76 seconds) with random jump commands given intermittently [24]. The robot had to *stand, jump, land, stand, jump, …* repeatedly. This trains the policy to not only land once, but to regain composure and be ready for the next move, which really enforces robust landings. To implement this, your environment needs to handle multiple goal commands in one episode. A simple way: sample a random number of jumps per episode or a time for the next jump, and when one jump is done (landed), if time remains, issue a new `c` goal. If the robot falls, terminate early. This is an advanced option; if it complicates your training code too much, you can instead simulate multiple jumps by just randomizing the initial state a bit on each episode (e.g., sometimes start already in a slight hop or after a small previous jump). But if feasible, multiple jumps per episode are great for testing recovery.

By the end of Stage 3, you should have a policy that can handle **full jump sequences**: steady standing, a strong takeoff with thrusters, and a stable landing, under various conditions. Importantly, this curriculum (Stage 1 → Stage 2 → Stage 3) mirrors the approach from the Cassie jumping paper [10] [20]: first learn a single jump (in place), then learn to generalize to many jumps, and finally add domain randomization for robustness.

Step 4: Reward Design for Thruster-Assisted Jumping

We've touched on reward terms in describing each stage. Here we summarize the key reward shaping components you should implement, which address thruster usage, jump height control, and landing stability:

1. Thruster Usage Penalty: We want the agent to use the thrusters effectively but not excessively. Introduce a penalty term for thruster activation to represent energy consumption or limited fuel. For continuous thruster actions, a common choice is a quadratic penalty on the force magnitude. For example:

```
# In CassieEnv.__get_reward (or similar)
thruster_left = self.last_action[10]   # or current action command
thruster_right = self.last_action[11]
thruster_cost = (thruster_left / max_thruster_force)**2 + (thruster_right /
max_thruster_force)**2
reward -= alpha * thruster_cost
```

Here `alpha` is a weight tuning how harsh the penalty is. (We divide by `max_thruster_force` to non-dimensionalize if actions are actual Newtons; if using normalized actions 0–1, you can drop that.) A smaller $α$ (like 1e-3 or 1e-4) means using full thrusters subtracts only a little from the reward, whereas a large $α$ (0.1 or 1.0) heavily discourages thruster use. Start with a modest penalty. You can schedule this weight per stage: e.g., in Stage 1, set a higher $α$ to basically teach the policy to not rely on thrusters for balancing; in

Stage 2 and 3, lower $α$ so it will use thrusters for jumping. This staged adjustment mirrors how you'd treat "torque penalties" in a curriculum – early on you might restrict actuations to force creativity, later you allow more actuation for the complex task. The Cassie jumping reward included a torque penalty term $r(τ,0)$ for energy efficiency [15] ; our thruster penalty is analogous.

Additionally, consider penalizing *thruster usage on the ground*. If you find the policy foolishly firing thrusters while Cassie is standing (which could be wasted energy or even destabilizing), you can add a condition: when contact forces indicate the robot is on the ground, increase the thruster penalty. This encourages thrusters to be primarily used during flight or the boost phase, not when idling on the ground (since real thrusters would likely be used sparingly due to fuel).

2. Jump Height / Distance Reward: To achieve *jump height control*, incorporate terms that reward reaching the desired apex or distance: - *Vertical Height:* If the task is to reach a certain height (e.g., jumping onto a platform of height `c_z`), you can reward the pelvis or foot reaching that height. One simple shaping is to use the difference between current height and initial height. For example, `r_height = k * (pelvis_z - nominal_standing_z)` during the flight phase, which gives a positive reward the higher the robot jumps. However, this unbounded reward could make it want to jump endlessly high (limited by thruster power). So it's better to center it around the goal: `r_height = -|pelvis_z_apex - target_height|` (negated error), perhaps scaled into (0,1] via an exponential or something. You might detect the apex height of the jump (e.g., when vertical velocity changes sign) and assign a reward then. If that's complex, rewarding height at the time of peak can be approximated by rewarding height at a fixed time after jump start (assuming that's near apex for a correct jump). - *Horizontal Distance:* If you incorporate forward/back/lateral jumps, similarly add `r_distance = -||(landing_position_x,y) - (target_x,y)||`. Or break it into components and reward closeness in X and Y separately [25] . In practice, giving a dense reward for forward distance traveled can also work (to encourage pushing off). For example, you could give a small reward each step for forward velocity if the goal is forward, etc., and a final reward based on how close to the target the robot landed. The Cassie paper specifically had a reward component comparing the final horizontal displacement to the commanded (c_x, c_y) and the final orientation to $c_ϕ$ [25] . Implementing this requires tracking the landing or episode end position. - *Accuracy vs. Height:* Decide if hitting the exact height is required or just exceeding a minimum. For platform jumps, *overshooting* can be as bad as undershooting (you might crash). So you want the robot to learn to tune its jump impulse. That means a peaked reward around the target. For example, reward could drop off if it jumps too high as well. If target height control is crucial, using a well-shaped error reward or even an imitation of a reference trajectory will help.

3. Landing Stability: This encompasses a few aspects: - *Upright Posture:* Encourage the robot to land with minimal tilt. After landing, reward small roll and pitch (e.g., `r_orientation = - (roll_angle^2 + pitch_angle^2)` or use a cosine of those angles to reward being close to 0). The paper explicitly added a reward for minimizing pelvis roll/pitch to 0, even though it wasn't in the task spec, because it helps stabilize the robot [14] . - *Soft Landing:* Penalize harsh impacts. If you can measure contact forces or acceleration on landing, add a penalty if they exceed a threshold. For instance, if the peak foot impact force is very high, subtract some reward. Alternatively, penalize high downward velocity just before contact, which correlates with impact. This will encourage use of legs (and possibly thrusters) to cushion the landing. For example: `r_impact = -γ * max(0, v_downwards_before_impact - safe_velocity)^2`. A simpler approach is to penalize big drops in pelvis height at landing (which indicate a hard impact compression). - *No Bouncing or Falling:* If the robot lands and then falls over within a short time, that's a failure. You should already terminate on falling, which zeroes out future rewards, but you can also explicitly penalize it. Many

implementations give a big negative reward (or zero final reward) if the robot falls. You might set up the episode so that a successful landing means the robot stands until episode end – thereby accumulating more reward – whereas a fall ends the episode with no further reward. This differential is enough. You can also give a discrete success bonus if the robot remains standing for, say, 2 seconds after landing (to really ensure it's stable). For example, `if not fallen in final 2s: reward += 50` (some significant number relative to other scales). This can dramatically focus the policy on **"stick the landing"**. - *Minimal Thruster on Landing:* We touched on this – if you want the robot to rely on leg damping rather than thruster to settle, penalize thruster usage after touchdown. This could be done by increasing the thruster penalty weight once contact is detected.

4. Other Shaping Terms: Retain general terms that keep the motion reasonable: - Joint position/velocity tracking if using reference trajectories (in Stage 1, you might have used imitation of a nominal jump trajectory [26], and possibly still use some in Stage 2). - Joint velocity and torque penalties to discourage flailing. For example, penalize very high joint speeds or excessive torque commands (similar to thruster penalty, you likely have these for motors already). - "Change of action" penalty as discussed to smooth control inputs [16]. - Alive bonus or time penalty: a small constant reward each timestep to motivate longer survival, which indirectly pushes for completing the jump without failing.

Designing a reward function is a bit of an art – you want to guide the policy to the desired behavior without over-constraining it. The Cassie jumping reward was a weighted sum of reference motion tracking, task completion, and smoothing terms [27]. In our case, we replace "reference motion tracking" with "jump height/distance tracking" since we may not have a predefined motion for thruster-assisted jumping, and emphasize task completion (achieving the commanded jump) as well as smoothness/efficiency.

Finally, verify that your reward terms are properly implemented in code (`__get_reward` in your env). Use a `reward_dict` (as you have in info [28]) to log each component for debugging. This helps to see if, say, the thruster penalty is too large or if the height reward is kicking in at the right time. You can adjust weights based on these observations during training.

Step 5: Ensure Robustness and Sim-to-Real Transfer (Domain Randomization)

If you plan to eventually deploy the learned policy on the real Cassie (or just want a robust policy), it's vital to bridge the reality gap. The Cassie jumping paper achieved **zero-shot sim-to-real transfer** by massive domain randomization and a specialized policy architecture [29] [30]. We will focus on domain randomization, as it's the most accessible technique.

Domain Randomization: The idea is to randomize simulation parameters within reasonable bounds so that the policy doesn't overfit to one "perfect" model of the world. In Stage 3 of training, enable randomization each episode [20]. Possible parameters to randomize: - *Robot Mass and Inertia:* Vary Cassie's mass distribution a bit. For example, ±10% mass of each link [19]. This makes the policy tolerant to modeling errors in weight and weight distribution. - *Joint Friction and Damping:* Randomize the damping in the joints or any friction in the simulator. Real robots have unmodeled friction; introducing a range (e.g., half to double the nominal damping) can help. - *Ground Friction:* Randomize ground friction coefficient each episode [31] [32]. E.g., one episode the ground is a bit slippery, another it's sticky. This affects landing and takeoff grip. - *Motor Strength:* Randomize motor torque scale or control noise. For instance, you could scale

the motor torques by 0.9 to 1.1 randomly, to simulate slight actuator strength differences or voltage sag. - *Thruster Power:* Critically, randomize the thrusters' effective force output. The real thrusters might produce less thrust than expected or have variance. You can model this by scaling the applied thruster force by a factor sampled from, say, [0.8, 1.2] each episode. That means sometimes the thrusters are weaker (policy must compensate by jumping harder with legs) or stronger (policy must not over-boost). - *Latency and Noise:* Add sensor noise to observations (small Gaussian noise to IMU readings, joint angles, etc.). Also consider a slight action latency or delay. PPO in simulation typically has none, but real systems have a control delay. You might not simulate this fully, but a small noise in actions or a 1–2 step delay in applying commands can make the policy more robust. Given your environment uses a Butterworth filter on actions, some smoothing is already in place, which is good. - *External Perturbations:* Optionally, apply random pushes or impulses to the robot during flight or right at landing. For example, a brief lateral force on the pelvis in mid-air (simulating wind) or a bump upon touchdown (uneven ground). Your environment's `PerturbationGenerator` likely handles that if enabled [7]. You can schedule a perturbation event randomly in some episodes to test recovery. The Cassie policy demonstrated recovery from external pushes during real tests [22], likely due to being trained in many varied scenarios.

To implement domain randomization in code, you can integrate with your environment reset routine. It seems you have an `EnvRandomizer` class (from `env_randomizer.py`) to sample random properties [31] [32]. Make sure to call it on reset during Stage 3. For example:

```
def reset(self):
    # ... normal reset ...
    if self.randomization_enabled:
        self.sim.set_body_mass(self.env_randomizer.get_rand_mass())
        self.sim.set_body_inertia(self.env_randomizer.get_rand_inertia())

self.sim.set_geom_friction(self.env_randomizer.get_rand_floor_friction(),
"floor")
        # Also possibly thruster scaling:
        self.thruster_force_scale =
self.env_randomizer.get_rand_thruster_scale()
    # ...
```

(Where `get_rand_thruster_scale` might return a value like 0.8–1.2 that you multiply your thruster actions by in this episode.)

From your code, `__set_dynamics_properties` is doing a lot of this on reset [31] [32]. Ensure `minimal_rand` vs full randomization flags are set appropriately for Stage 3 (perhaps `minimal_rand=False` to randomize all parameters).

Policy Robustness Techniques: Besides domain randomization, there are other techniques: - *Trajectory Noise:* Add minor noise to reference trajectories (if you use them) or to the target commands, to avoid the policy relying on perfectly consistent commands. - *Curriculum in Randomization:* Start with narrow randomization and gradually widen the range as training progresses, so the policy isn't overwhelmed early. You might, for example, linearly increase the randomization range over the first few thousand iterations of Stage 3. - *RMA (Rapid Motor Adaptation):* The Cassie paper mentions an adaptive method (A-RMA) in

comparisons [33], but their solution didn't require an explicit adaptation module; they relied on the policy's history of experience to adapt. You likely don't need to implement RMA if domain randomization suffices. - *Real-World Considerations:* If transferring to hardware, remember to enforce safety. For example, limit thruster commands to not damage hardware, and consider that real thrusters might have ignition delay or minimum thrust before they turn on. Simulating those (like a deadzone below 10% thrust) in training could be wise.

By training in a variety of conditions, the policy should learn strategies that generalize. The original jumping policy was indeed transferred to the real Cassie with no further tuning, thanks to heavy randomization [29] [30]. Aim for the same: randomize enough that the real robot's characteristics are just another variation within the training distribution.

Step 6: Integrate Thrusters and Curriculum into the PPO Training Loop

Finally, you need to integrate all these changes into your PPO pipeline (`ppo_sgd_mlp.py` and `train.py`) so that training runs smoothly through the stages and the thruster controls are optimized.

Policy Network Adaptation: Since the action space is larger (12 outputs instead of 10), adjust your policy network's output layer size. If you're using an MLP policy, simply ensure the final linear layer outputs 12 means (for the Gaussian policy in PPO) instead of 10. Likewise, the logstd or covariance structure should accommodate 12 dimensions. If you built the action space using Gym, libraries like Stable Baselines will handle matching output dimensions automatically when given the env. If writing your own, just be sure to change any hard-coded action dimension.

Observation Space Changes: The network input dimension might also increase (if you added thruster states or goal parameters). Adjust the input layer size or observation normalization accordingly. For example, if you added 2 thruster values and (say) a jump height command to observation, that's 3 extra inputs. Confirm that the data flows through (your `train.py` might be using environment observation directly, so if you updated the env's `observation_space`, the PPO code should adapt, but double-check any preprocessing).

Multiple Stages Training Script: There are a couple ways to implement the multi-stage training: - **Sequential Training Runs:** The simplest is to treat each stage as a separate training run that initializes from the previous one. For instance, you run PPO for N iterations in Stage 1 (balance) until convergence, save the model. Then modify the environment config for Stage 2 (enable jump rewards, etc.), load the saved model weights into the policy, and continue training for M iterations for Stage 2. Then do the same for Stage 3. This approach gives you control to tweak hyperparameters or reward weights between stages manually. It sounds like you have a configuration system (perhaps in `config` dict passed to CassieEnv) – you could have something like `config["stage"]=1` that the env uses to set up the appropriate rewards and dynamics. Indeed, your code references that Stage 1 specifics are in an appendix in the paper [34], which suggests you might have separate config/hyperparams for it. So, you might do:

```
```python
# Pseudocode for training sequence
```
```

```
env = CassieEnv(config_stage1)
policy = PolicyNetwork(obs_dim=env.obs_dim, act_dim=env.act_dim)
ppo = PPO(policy, ...)
ppo.train(env, timesteps=... )  # Stage 1 training
ppo.save("model_stage1.pth")

env = CassieEnv(config_stage2)   # new env with multi-goal, no rand
policy.load_state_dict("model_stage1.pth")  # initialize with stage1
ppo = PPO(policy, ...)
ppo.train(env, timesteps=... )  # Stage 2 training
ppo.save("model_stage2.pth")

env = CassieEnv(config_stage3)   # env with randomization on
policy.load_state_dict("model_stage2.pth")
ppo = PPO(policy, ...)
ppo.train(env, timesteps=... )  # Stage 3 training
ppo.save("model_stage3.pth")
```
Each stage might use the same hyperparameters and network structure (the Cassie paper used same network and reward scaling for Stage 2 & 3, with some differences only in Stage 1【3†L49-L57】【3†L115-L123】). You can also adjust learning rate per stage if needed (e.g., maybe a lower LR in Stage 3 as things get fine-tuned).

- **Single Run Curriculum:** Alternatively, you can program the training loop to automatically progress through stages. For example, after X million steps, switch the environment's mode. This could be done by checking the timestep and calling something like `env.set_stage(2)` to change the internal parameters (enabling new reward terms, etc.). Since your environment likely doesn't seamlessly switch modes on the fly (and PPO's stability might be affected by a sudden reward change), the sequential approach is safer. But a well-planned curriculum scheduler can work too. It's more complex to implement, so given your moderate familiarity, we suggest doing separate runs or manual stage transitions.

PPO Algorithm Adjustments: Integrating thrusters doesn't fundamentally change PPO, but here are some considerations: - *Reward Scaling:* With new reward terms (especially penalties and bonuses), ensure the reward scale is reasonable. Extremely large positive or negative rewards can destabilize training (if you give a +50 success bonus, you might want to clip rewards or normalize advantages accordingly). PPO typically can handle it, but monitor the returns. - *Normalization:* If you use observation normalization or reward normalization in PPO, include the new obs dimensions and the sometimes sparse rewards (like height success). It might help to normalize observations (e.g., Z-scoring sensor inputs) especially since including thruster commands (0 when off, maybe occasionally large when on) could have different scale than other observations. Libraries often do this automatically, but if not, you can track running mean and var of obs. - *Clipping:* If you imposed any action clipping in PPO (beyond the env bounds), ensure it accounts for thruster dimension too. For instance, if clipping gradient or action magnitudes, 12-dim vs 10-dim might slightly change norms. - *Entropy Bonus:* With a larger action space, the default entropy regularization might encourage more exploration in thrusters. That's fine, but you might need to tune the entropy coefficient. If

you see the policy dithering thrusters too much, you could lower entropy bonus to encourage exploitation once it finds a good strategy.

Logging and Debugging: Extend any logging to include thruster info. For example, log the average thruster force per episode, the max height achieved, etc. This will help you know if training is progressing: - In Stage 1, thruster usage should ideally be near zero (if penalized) and no falls. - In Stage 2, you'd expect to see thruster usage spike around takeoff, and some height achieved. Initially it might not jump at all – if so, adjust the reward to push it more. - In Stage 3, watch for oscillations or instability when randomization is introduced. If performance drops too much with random dynamics, you might need to ease into it (e.g., start Stage 3 with mild randomization, then ramp up).

Policy Architecture (optional improvement): The Cassie jumping policy used a special network with an LSTM/CNN-based history encoder to handle the timing of jumps [36] [37]. If your PPO pipeline supports recurrent or history-based policies, consider using them. For example, a **recurrent PPO (LSTM)** could inherently handle the phase of the jump by remembering past states. If not, your current approach of feeding previous actions as part of state is a simpler substitute. Given you have `previous_obs` and `previous_acs` in a deque [38], it looks like you might already be constructing some history input. Ensure your policy network is set up to receive that if intended. A recurrent policy or a stacked observation can improve performance on this highly dynamic task. However, if that's too much engineering, a well-shaped state (including velocities, phase indicators like "time since jump command", etc.) can suffice with an MLP. Many successful PPO locomotion policies are MLPs with carefully constructed state vectors.

Training Duration: Jumping is a complex skill – be prepared to train for a lot of timesteps. The paper collected very large batches (65k samples per iteration, thousands of iterations) [39] [35]. In your case, you might not need that extreme, but don't be surprised if it takes tens of millions of environment steps to converge through all stages. Using multiple parallel environments (if supported) can speed up data collection.

Stage Transitions: Before moving from one stage to the next, check if the policy has sufficiently met the criteria: - Stage 1: Hardly ever falls over in the allowed time. - Stage 2: Successfully gets airborne and lands on feet more often than not (perhaps with some instability). - Stage 3: Handles random variations without failing most of the time and hits various targets. If one stage hasn't learned the basics, it will struggle in the next. For example, if Stage 2 still often falls, adding randomness in Stage 3 will be too hard. Spend extra training time or tweak rewards in earlier stages as needed.

Integrating with `train.py`: Your `train.py` likely orchestrates creating the env and running PPO. To integrate: - Add command-line or config options for stage selection (so you can run `train.py --stage 1`, etc.). - When stage 2 starts, load the checkpoint from stage 1 (you might use torch load for model weights or if using StableBaselines, use their `load`). - Make sure the environment is constructed with the correct parameters for each stage (you might have a config file or just programmatically set flags like `env.randomization_enabled = False` in stage2, `True` in stage3, etc., and possibly `env.multi_goal = True` in stage3). - Incorporate the reward shaping differences: e.g., you might have a function to compute reward that references some stage-dependent weights. You can pass the stage as part of the env or policy config.

Example (Pseudo-code for PPO loop with thrusters):

```
# Stage 2 example snippet: training loop iteration
obs = env.reset()
while not done:
    action = policy(obs)             # policy outputs 12-dim action
    obs_next, reward, done, info = env.step(action)
    # PPO storage & update...
    obs = obs_next
    # (The env internally computes reward with thruster penalties, etc.)
```

The main difference from before is just that `action` includes thruster commands and the env's step function handles them and computes the reward accordingly. PPO's update step will use the reward, which now includes thruster and jump-specific terms, to compute advantage and policy gradients as usual.

Test in Simulation: After training, always test the learned policy in the simulator (visualize it if possible) for a variety of scenarios: no perturbation vs randomized, different targets, etc., to verify it meets expectations. Tweak and retrain if needed.

By following these steps, you will systematically implement the thruster-assisted jumping as described in the 2023 Cassie paper's method, tailored to your setup. We started by expanding the state and action definitions to include thrusters, then gradually taught the policy fundamental stability, explosive jumping, and resilient landings through a staged curriculum [10] [20], shaping the reward function to encourage effective thruster use, sufficient jump height, and safe landings (using terms analogous to those in the paper [25] [15]). We modified the simulation to apply thruster forces each step [4] and included domain randomization to bridge to reality [19]. Finally, we integrated these changes into the PPO training loop, ensuring the neural network and training process accommodate the new thruster controls.

With careful tuning and sufficient training, your Cassie should learn to balance with thrusters, execute powered jumps, and land reliably on its feet. Good luck, and happy jumping!

Sources:

- Zhongyu Li *et al.*, "Robust and Versatile Bipedal Jumping Control through Reinforcement Learning," 2023 – for multi-stage RL approach and reward design [10] [25].
- Your provided Cassie environment code – for implementation details of thruster integration [1] [4].

---

[1] [2] [3] [4] [5] [6] [7] [28] [31] [32] [38] cassie_env.py
file://file-BHoLWp4RVMboEAnZWSRmKa

[8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [29] [30] [33] [34] [35] [36] [37] [39]
Cassie_Jump_2023.pdf
file://file-PmsqzSDf2QF94SLHSXiWpQ