Frequently Asked Questions

The Isaac Gym has an extremely large scope. It deals with physics simulation, reinforcement learning, GPU parallelization, etc... There's a great deal going on "under the hood" and so it's only reasonable that a user might have questions about what exactly is going on or how exactly to do certain common things. In this section we address those questions that seem to occur the most often.

What is the difference between dt and substep?

dt is the time increment of the simulation. Every time you step the simulation via `gym.simulate(...)` you more the sim forward in time by dt. The substep is how many slices this dt is split into in order to do the simulation. The more substeps, the finer the slicing of time and the greater the accuracy of the simulation, but the more computationally intensive it becomes. For example: say your dt is 1/60 and your substep is 2. When you call `gym.simulate(...)` the physics is propagated forward 1/60th of a second in two time steps of 1/120th of a second each. You do not have access to the results of these 1/120th of a second time steps, only the total result at 1/60th of a second can be retrieved by calling `gym.fetch_results(...)`. This is probably a consequence of certain numerical integration methods.

As a zeroth order example, lets say the integration method was Euler's method. If we have some function, $X(t)$, and we know $X(t = t_0)$, and we know the first derivative, X', then we can estimate $X(t)$ at some future time $t + \mathrm{d}t$ naively

$$X(t + \mathrm{d}t) \approx X(t_0) + \mathrm{d}t X'(t_0)$$

In the limit as $\mathrm{d}t \to 0$ this relationship becomes exact. However, in real life, $\mathrm{d}t$ is *never* zero and we will always have some form of integration error. Suppose $\mathrm{d}t = 1, X(t_0) = 1, \mathrm{and} X' = X$. This means that $X(t)$ is exactly e^t, and so at t = 1 our exact result should be e, or about 2.71828. If we approximate $X(t = 1)$ in a single time step we get 1 + 1*1 = 2, more than 25% less than what it should be. If we divide that single time step into two substeps we get

$$X(t + 0.5) \approx 1 + 0.5 * 1 = 1.5$$

$$X(t + 0.5 + 0.5) \approx 1.5 + 0.5 * 1.5 = 2.25$$

which is off by about 17%. Much improved! Euler's method is sensitive to the size of dt and the number of substeps chosen and while Gym doesn't use Euler's method, even advanced numerical integration algorithms are sensitive to step size and the number of substeps taken. You cannot escape integration error.

In general, dt should be much less than 1 and the number of substeps should be a small integer (2 or 4 is common). If dt is too large you will see weird behavior in the simulation (things will explode). If substeps is too large, the simulation will take an undesireably long time to run.

What happens when you call gym.simulate()?

When you call `gym.simulate(...)` you are stepping the simulation through time by an amount equal to dt. As described above this simulation is usually divided into substeps for accuracy reasons. To save memory, the simulation results for each substep are only temporarily saved, while the results at the end of dt are available to be fetched from the device.

It is entirely possible that simulating 1 second of the virtual world could take longer than 1 second of real time, but if you are using modern hardware this should only be a concern for the most complex of physics scenes (if you see this in practice and you aren't trying to simulate an entire ocean of particles, then chances are high that there's some other issue elsewhere in your code). More likely than not, it will take significantly less than 1 second of real time to simulate 1 second of virtual time. If the frame rate renders as fast as possible this would make things in the world appear to move much faster than they would in real life. You can mitigate this by using the `gym.sync_frame_time(...)` python binding, which synchronizes the simulation to the rendered frame rate, and forces the program to wait until dt real time seconds have elapsed before continuing.

How do you handle multiple actors in one environment?

Isaac Gym is data oriented, and so all actors and bodies are accessed via their handles. Currently these handles are unique on a per environment basis, but this could change in the future. Presently, actors are only 'aware' of other actors and objects in their environment. This is true for sensors as well. If you render a camera sensor in env 1, all objects from env 2 will not appear in the render. It is left to the user to determine how best to manage these handles and there are no real consequences to having multiple actors in a single environment.

API function calls for accessing actors, rigid bodies, and DOFs all require the corresponding actor handle, body handle, and environment pointer. For example, suppose you have an environment with a Franka Panda robot arm and a cabinet. Say you spawn a few hundred of these on your device. If you want to access the end effector on a specific panda in a specific environment, you would need the handle for the end effector, the handle for the actor, and the env pointer to that environment. Getting the actor handle and the env pointer are easy, you just save those at

construction. Getting the end effector is a bit trickier. We know that the end effector is called "panda_hand" in the asset file, so we can use the `gym.find_actor_rigid_body_handle(...)` function to get the handle of that end effector.

```
effector_handle = gym.find_actor_rigid_body_handle(env_ptr, franka_actor_handle,
"panda_hand")
```

In short, multiple actors in a single environment is handled by handles.

How do I move the pose, joints, velocity, etc. of an actor?

There are a number of ways to do this.

First, we can simulate an applied torque on the joints. This requires a few things; we need the joint "gear", the power scale, a vector of the actions to take, the environment pointer, and the actor handle. Applying torques to an actor is done by calling `gym.apply_actor_dof_efforts(...)`. This function accepts the environment pointer, the actor handle, and an array of the torques to be applied (the efforts). The length of that array must match the number of articulatable joints in the actor. This effort array is in units of Nm, and it is usually calculated by the product of the joint gear, the power scale, and a vector of actions (floats on -1,1 or 0,1 or whatever, depending on the actor). The gear of the joint is the maximum torque that can be applied to the joint, while the power scale is just some factor to further scale that value. When the `gym.simulate(...)` function is called, these torques will be applied to the joints, resulting in angular acceleration etc...

Second, we can specify inverse kinematic targets for the joints. You can set position targets using `gym.set_actor_dof_position_targets(...)`. When simulate is called, the actor joints will move based on their joint constraints and the effort DOF parameter to the target positions.

Third, you can just manually set the state of the bodies in the actor. **This can be dangerous!**. In order to do this you need to "respect" the joint constraints in the target state or you risk making your actor explode. Because of this, you should only ever use this when you have a "cache" of target states you want to reset to (like resetting a training environment or whatever). This is done by calling `gym.set_actor_rigid_body_states(...)`. This function uses a numpy strucured array to define the target rigid body states. The bindings define a set of dtypes that can be used to create numpy structured arrays for specific types of data: GymRigidBodyState is one of those types. It has two fields, 'pose' and 'vel' for pose and velocity respectively. Pose and velocity are actually GymTransform and GymVelocity respectively, which are also structured arrays (again, you don't ever want to set this manually if you can avoid it). This call also requires a gymapi state enum, which defines exactly what state data you want to set with this call, with the most common being `gymapi.STATE_ALL` meaning you want set both the body transforms and velocities. When `gym.simulate(...)` is called, the actor bodies will teleport to the specified state

instantaneously. The best way to use this function is to call `numpy.copy(gym.get_actor_rigid_body_states(...))` to save the state, and then use the “set” call to reset to that state some time later.

Finally, you can call `gym.set_actor_dof_states(...)` to manually set the states of the DOFs (position and velocity). This is done just as above, and just like before **it can also be very dangerous** . Use this to reset joints to a cached state that you copied and saved from before.