Simulation Tuning

There are many factors that affect simulation stability and performance.

Simulation Parameters

The `SimParams` allow you to specify parameters for the physics solver.

Simulation Parameters Tuning

Either PhysX CPU or Flex could be used as a simulation backend. Support of PhysX GPU Articulation Solver will be added soon. There are 2 common parameters that could be tuned to improved performance and/or simulation stability - timestep and number of substeps and a number of engine specific parameters. Deformable object simulation is supported only be Flex at the moment.

Common Parameters

- **dt** - Simulation timestep, the default is **1/60 s**
- **substeps** - Number of simulation substeps, the default is **2**. The effective simulation timestep is **dt/substeps**.

The **dt** parameter is the time step used to advance the simulation, in seconds. Accurate and stable physics simulation requires fairly short timesteps, generally under 1/50th of a second. Time steps that are too long can lead to instabilities, especially with fast-moving objects, strong forces, or complex articulated assemblies. In addition to advancing the simulation, you will typically have some code for querying the state of the world and applying controls during every iteration of the loop. The **dt** parameter, then, determines the frequency at which you interact with the simulation. Quite often, the frequency of that interaction is lower than the frequency required for stable physics simulation. The **num_substeps** parameter can be used to subdivide each timestep into equal sub-intervals to achieve a stable physics simulation. In the snippet above, **dt** is 1/60, so you can interact with the simulation 60 times during every simulated second. But **num_substeps** is 2, which means that the physics simulation advances in increments of 1/120 seconds. These are the default simulation parameters.

Flex

- **solver_type** - there are 6 constraint solvers available for Flex, numbered as follows:
- **0 - XPBD (GPU)** - a position-based solver that uses an iterative descent method, this method is less accurate than the Newton solvers (below), but generally fast and robust for moderately stiff systems.

Flex also supports a nonlinear Newton solver with a number of different linear solvers as a backend, these are listed below:

- **1 - Newton Jacobi (GPU)** - Jacobi solver on GPU (CUDA)
- **2 - Newton LDLT (CPU)** - Cholesky backend on CPU (Eigen-based)
- **3 - Newton PCG1 (CPU)** - Preconditioned conjugate gradient on CPU (Eigen-based)
- **4 - Newton PCG2 (GPU)** - Preconditioned conjugate gradient on GPU (CUDA)
- **5 - Newton PCR (GPU)** - Preconditioned conjugate residual method on GPU (CUDA)

The default and recommended one is **5 - Newton PCR** solver, the CPU backends may be very slow for large systems and are mostly present for verification purposes.

- **num_outer_iterations** - number of iterations taken by the solver per simulation substep.
- **num_inner_iterations** - number of linear solver iterations taken for each outer iteration, is used only by Newton solvers
- **relaxation** - control the convergence rate of the solver. The default values is 0.75 Values greater than 1 may lead to instability. Newton solvers currently use a zero velocity starting iterate, so if not converged sufficiently too high relaxation can introduce a kind of numerical equivalent to damping = 1 - (1 - relaxation)^numOuterIterations.
- **warm_start** - Fraction of the cached Lagrange multipliers to be used on the next simulation step. Accelerates convergence, the default, conservative values is 0.4. Larger values could lead to a more bouncy behaviour and some times instabilities in case of presence of fast movement and/or large, rapidly changing forces in a system. In a situation where you are trying to simulate a quite slow-moving system with a lot of contacts - say robotics manipulation tasks with grasping complex shapes you can benefit from trying higher warm start values, up to 1.0.
- **shape_collision_distance** - Distance in meters for particles to maintain against rigid shapes (separate from radius parameter).
- **shape_collision_margin** - Distance in meters at which contacts are generated. Flex uses a speculative contact model, when feature pairs (e.g.: vertices/edges) come within this distance of each other at the start of a time-step a contact constraint will be generated. If bodies are moving quickly then the margin should be large enough to ensure that contacts are not missed during a time-step. One way to formalize this is to say that if the maximum velocity a feature travels with is v and the time-step is dt, then the margin should be margin =

v*dt/substeps. If you are dropping objects from a large height and seeing them ‘pop’ back up, then it is likely to having to low a collision margin (objects interpenetrate and are then ejected).

The most universal way of improving simulation stability (convergence) and decreasing penetrations is increasing number of substeps. But it could be simulationally quite costly. Other methods to try are:

In case lack of convergence increasing number of outer and inner iteration could help. Another source of instabilities could be dues to the bad initial configuration and presence of self-collisions. It can be diagnosed by visualizing contact forces in the system and disabling self-collisions for a robot whenever it is possible or fixing the initial configuration. In addition to increasing number of substeps making shapeCollisionDistance larger when it is possible could help to prevent penetrations for fast moving objects. In case penetrations are present in a slow moving system, like robotic arms, first of all contact forces generated in a simulation should checked and reduced if needed, for example by making robot motors weaker.

Shape representation in Flex is through triangle meshes - all primitive shapes including spheres, capsules, and boxes are represented internally by triangle meshes stored in a GPU BVH structure (spheres and capsules can be thought of as a single degenerate triangle + thickness). This unified representation means Flex can handle non-convex and arbitrary triangle meshes for dynamic physical shapes, however unlike e.g.: convex polyhedra, triangle soups do not generally define “inside/outside” regions. This means once interpenetration occurs it cannot be resolved in a well defined way. To avoid interpenetrations it is recommended to use a sufficient thickness layer on the collision shapes.

PhysX

- **num_threads** - number of CPU cores used by PhysX. Default is 4. Setting to 0 will run the simulation on the thread that calls PxScene::simulate(). A value greater than 0 will spawn numCores-1 worker
- **solver_type** -type of solver used. The default and recommended to use is 1 - **TGS**: temporal gauss seidel solver. It is a non-linear iterative solver.
- **contact_offset** - shapes whose distance is less than the sum of their contactOffset values will generate contacts. Default 0.02 m
- **rest_offset** - two shapes will come to rest at a distance equal to the sum of their restOffset values. Default is 0.01 m
- **num_position_iterations** - PhysX solver position iterations count. Default 4
- **num_velocity_iterations** - PhysX solver velocity iterations count. Default 1
- **bounce_threshold_velocity** - A contact with a relative velocity below this will not bounce. Default is 0.2 m/s
- **max_depenetration_velocity** - The maximum velocity permitted to be introduced by the solver to correct for penetrations in contacts. Default is 100.0 m/s

To improve solver convergence, usually only position iterations number should be increased. Velocity iterations can negatively impact solver convergence. They are there to reduce some energy gain from penetrations but make the overall simulation less stiff. Good the default choice of velocity iterations for the TGS solver is 0. Instability has to be looked at at a case-by-case basis. However, general rules are:

If it is caused by deep penetration, either more sub-steps should be used or limiting the energy the solver can inject to correct errors will help. See `PxRigidBody::setMaxDepenetrationVelocity`. The default value is 5 m/s but a larger value, e.g. 100m/s, can help to remove penetrations in the case of big forces present.

Increasing joint armature values, if possible, can also increase simulation stability.

If instability is basically the system failing to converge, increasing the number of position iterations or sub-stepping should help. If PGS is used, substepping has a much bigger influence on sim quality than increasing iterations, but substepping is way more expensive (iterations are relatively cheap). If TGS is used, substepping and iterations are more similar in terms of how they impact convergence. Iterations are much cheaper than a substep, but not as cheap as the PGS solver. Whenever possible, using TGS instead of PGS is highly recommended. Other parameters that may influence stability are: shape contact offset (margin), rest offset (inflation)

Scene parameters

- **bounce_threshold_velocity** - relative velocity required to trigger a bounce. The default is 0.2 m/s, which is some cases could be pretty high. Reducing could give a more natural bouncing behaviour, for example in case of bouncing balls.
- **friction_offset_threshold** - the contact distance at which friction starts being computed. The default is 0.04 m. If a very small objects are simulated, it should be decreased.
- **friction_correlation_distance** - contact points can be merged into a single friction anchor if the distance between the contacts is smaller than correlation distance. The default is 0.025 m.

PhysX Visual Debugger (PVD)

The PhysX Visual Debugger (PVD) allows you to visualize, debug, and interact with physical scene representation when PhysX simulation is used: https://developer.nvidia.com/pvd-tutorials

PVD will only work if you are using the PhysX backend.

If you are building from source, search premake5.lua for a line like this:

```
local physxLibs = "profile"
```

The physxLibs variable should be set to either “profile” or “checked” for PVD to work. If you change this variable, make sure to rebuild.

By default, Gym will try to connect to PVD running on localhost. If you wish to connect to PVD on a different machine, set the environment variable GYM_PVD_HOST to the IP or hostname.

You can set the environment variable in the terminal or you can do it in your Python script like this:

```
import os
os.environ["GYM_PVD_HOST"] = "xxx.yyy.zzz.www"
```

If you want to save the PVD capture to a file instead of connecting to PVD live, set the environment variable GYM_PVD_FILE to the file name. You may omit the extension.

For example:

```
os.environ["GYM_PVD_FILE"] = "foo"
```

will create a file named foo.pxd2.