During my time here, I implemented a simulation of a robot arm which pushes a box to a target position, in *MuJoco* and *PhysX*. The robot arm used a simplified version of the Model Predictive Path Integral Control algorithm.

My supervisor, Michael Mistry, asked me to draft this paper so the next person who works on this topic can easily pick up on what I've already built and learnt.

All of the programs and code I developed while being here can be checked on this repo: https://github.com/TomasRibeiro96/tomas_work-UoE. Note that some of the code might not be in the cleanest form, since it was a work in progress.

MuJoco

Getting Started

The MuJoco physics simulator can be downloaded from this webpage: https://www.roboti.us/index.html. I downloaded the latest version of *Mujoco* (formerly *MuJoco Pro*), more specifically, the "*mujoco200 linux*". If you need help installing *MuJoco* on *Linux,* I strongly suggest watching this YouTube video: https://www.youtube.com/watch?v=xG8oujhD9lA. This video displays a step-by-step tutorial of how to install *MuJoco* as well as how to run programs. You can also find a simplified version of this in the *README.txt* file which can be found on the "*doc"* directory.

In order to understand how *MuJoco* works, how is its API defined and how to define scenes, I would refer the reader to the *MuJoco* documentation (http://www.mujoco.org/book/computation.html) as it is very extensive and helpful. In addition to this, *MuJoco* also enjoys its own Forum (http://www.mujoco.org/forum/index.php), where users can ask questions or report problems and *MuJoco*'s creator (Emo Todorov) will promptly respond. I found these two resources extremely useful and almost all my problems were solved by consulting one of these. Plus, a *Google* search of "mujoco" followed by the problem you're experiencing will many times refer you to a previous *MuJoco Forum* post where someone has experienced a similar problem.

MuJoco has its own format of *xml* files, from which it loads *links, joints* and other *actors*. Although the *MuJoco* format is similar to *URDF*, it has its differences. Knowledge of the *URDF* format might ease the learning of the *MuJoco* format, but it does not dispense the need to learn it by itself. The best way to learn the *MuJoco* format is to read the examples given throughout the *Documentation* (e.g. http://www.mujoco.org/book/index.html#BodyGeomSite, http://www.mujoco.org/book/modeling.html#CDefault) and to compare the models given on the directory "*model"* to their simulation, which can be obtained by running the command "*./simulate"* in the "*bin"* directory*.* Additionally, you can also try to search online for other examples (e.g. https://medium.com/coinmonks/mujoco-tutorial-on-mits-underactuated-robotics-in-c-part-0-2cbd259f6adc) although I did not find anything that was as helpful as the *Documentation* itself and the examples provided in the *model* directory. The "*MJCF Reference"* section (http://www.mujoco.org/book/XMLreference.html#Reference), inside the "*XML*

Reference" section, is also very useful because it does not limit itself to define the hierarchies and commands. Each command is accompanied by a piece of text which explains its function and which values can be attributed to it. This piece of text can sometimes also refer the reader to other sections of the documentation (such as *Computation, Modeling* or *Programming*) when more details are sought about something. After surpassing the initial learning curve and understanding how the *MuJoco* format works, I found that the "*MJCF Reference"* was the only resource I needed. Since the "*MJCF Reference"* is written in such a clear way, many times just going through it might be enough to understand how to do something.

Note 1: Although *MuJoco is capable of loading URDF* files, it does not do so ideally and problems can arise. It would be preferable to write your model directly with the *MuJoco* format. As an example, here http://www.mujoco.org/forum/index.php?threads/meshes-ignored-when-converting-urdf-to-mjcf.3433/ you can find a *MuJoco Forum* post where the problem arises from using a *URDF* file (see especially from reply #13 onwards).

Compile and Render MuJoco Simulations

To render *MuJoco* simulations, make sure you have *GLFW* installed and updated on your computer.

To understand how to create a *C* file which loads the *xml* files and renders the simulation, you may check the file "*basic.cpp" in* the "*sample"* directory.

To compile your C file, you can use the following line of code on your terminal:

gcc -I/path/to/mujoco200_linux/include -L/path/to/mujoco200_linux/bin -fopenmp -mavx -pthread -Wl,-rpath,'$$ORIGIN' FILENAME.c -lmujoco200 -o EXECUTABLE_NAME -lpthread -lglfw -lm -lGLEW -lGLU -lGL.

Notice that although this line can be used to compile your code, not all flags used in it are strictly necessary. For example, "*-fopenmp"* is only necessary if you're using *OpenMP* for parallelisation and "*-lglfw"* is only necessary if you're rendering the simulation.

While trying to run a simulation, if you ever get an error such as *"error while loading shared libraries: libmujoco200.so: cannot open shared object file: No such file or directory"* then it can be simply solved by typing *"LD_LIBRARY_PATH=/path/to/mujoco200_linux/bin"* on the terminal.

Parallelisation

To achieve parallelisation I used *OpenMP*. *OpenMP* does not need to be installed, it is already included but disabled in most compilers by default. To enable it on *gcc,* you need only to use the flag "*-fopenmp"* when you compile your code (just as it is used on the compilation line provided above). To learn how to use *OpenMP*, I strongly suggest that you watch the video tutorials provided by *OpenMP* itself on YouTube (https://www.youtube.com/watch?v=nE-xN4Bf8XI&list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG). Although watching the videos may be more time-consuming, I found them to be extremely helpful. The videos teach not only how to use OpenMP, but also how to do parallel computation, even addressing several mistakes you might make and explaining how to optimse your code.

Note 2: Given that, in the file *“robot-arm-paral-v2.c”* , there are arrays with the number of threads and timesteps as their size, the code will create very large arrays. Creating arrays this big would result in an error or weird results, due to the memory occupied by it being too much for the stack size. As a way to get around this problem, you may declare the arrays as “*static*” or “*global*”. A more detailed and technical answer can be found here https://www.researchgate.net/post/What_is_the_maximum_size_of_an_array_in_C.

Note 3: I tried to automate the process of detecting how much time the simulation took with different numbers of threads. However, since all the arrays were declared as *“static”* or *“global”,* it was not possible for the code to change their sizes. Given I had a limited amount of time to work on this, I ended up changing the number of threads and timesteps manually each time I ran the code. Afterwards, I would simply check how long the program took to run. I recorded the run time values on an Excel spreadsheet and plotted the data myself.

NVIDIA PhysX SDK

I would say that, overall, *PhysX SDK* was more difficult to work with than *MuJoco*. Some of the reasons why it was more difficult to use are:

- It was very hard to find answers online for the problems I was having. The *PhysX Forum* didn’t prove as helpful as the *MuJoco’s* and, besides this, PhysX SDK is a physics simulator whereas *PhysX* is a program to enhance gaming graphics and experience. This results in a lot of confusion while searching online for answers to problems since there more questions being asked about *PhysX* (because it’s used by the general public) than about the *PhysX SDK*. In addition to this, there are also users who use the *PhysX SDK* with *Unity* or *Unreal Engine* directly, so even some of those questions might not be relevant for robotics researchers;
- The *PhysX SDK* possesses a debugging tool called *PhysX Visual Debugger*, but it is only available for *Microsoft Windows* platforms. Hence, I highly recommend using *Microsoft Windows* when working with the *PhysX SDK*. A dual boot solution may be preferable, using *Linux* for *MuJoco* and other physical simulators, but using *Microsoft Windows* for *PhysX SDK*;
- In December 2018, *PhysX 4.0 SDK* was released, with features useful for robotics and machine learning simulations. However, this means that, for the time being, this update has been out for less than one

year and there aren’t many online resources about the new features it added;

- The *PhysX SDK* documentation can be quite vague and sometimes incomplete. If you try to do something just as it is explained in the documentation, there’s a chance it won’t work because, without it being pointed out to you, you needed to edit other parts of your code;
- By some reason, all the online tutorials I found were from 2010 up to 2012, which roughly corresponds to the time period that PhysX 3.0 SDK was released. This means it initially got a lot of attention but it was soon followed by disinterest, since there were no online resources emerging. Only recently has PhysX SDK got more attention due to it becoming open-source. Unfortunately, this means that almost all of the online resources are outdated;
- In PhysX, unlike in the URDF format (Note 6), the rotation of a body does not affect the orientation of its origin. For this to happen, the origin itself would need to be purposely rotated. However, to rotate the bodies one must resort to quaternions. Quaternions can be quite a complicated concept to grasp, especially if one is not familiar with them. There is an excellent series of interactive videos (https://eater.net/quaternions) which explain quaternions and their rotations. These videos can be paused at any time for the user to interact with the simulation. For a more practical knowledge of quaternions, this website https://quaternions.online/ can also be useful to obtain quaternions from Euler angle’s rotations. I found that, using both these websites, allowed me to rotate all the bodies to the orientations intended.

Getting Started

PhysX SDK can be downloaded here: https://developer.nvidia.com/physx-sdk. The same link can also take you to the “*Developer Guide*”, in which you can find the:

- “*User Guide”*, which corresponds to the *PhysX SDK* documentation (https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Index.html);
- “*API Reference*” (https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxapi/files/index.html);
- “*GitHub Forums*” which corresponds the *PhysX Forum*. Be aware that the GitHub Forum is fairly recent. PhysX used to have its own Forum

(https://devtalk.nvidia.com/default/board/66/physx-and-physics-modeling/) which they intend to stop using. They want users to post their questions to the *GitHub Forums*. However, *NVIDIA* did not port the questions from their previous Forum to the *GitHub Forum*, so it might be wise to check both.

This might sound a bit obvious but the webpage design can be misleading. In the old *PhysX Forum*, the magnifying glass found in the top right corner of the screen will only search on the Forum and not the entire NVIDIA website. You can use this magnifying glass to search for a problem inside the Forum.

For the reasons aforementioned, it is not easy to learn how to work with *PhysX SDK* and I cannot refer the reader to many online resources. I would advise the reader to start by watching this YouTube video: https://www.youtube.com/watch?v=VfrFpcPvaos&t=236s. This video explains how to install *PhysX SDK*, as well as how to compile and run its snippets. After watching the video, I believe that the best way to learn how to work with *PhysX SDK* is a mix between consulting the *PhysX SDK Documentation* and comparing the codes found on the "*snippets*" directory with their resulting simulation.

Unfortunately, I was not able to compile my own code on the PhysX SDK. Because of this, and for time-management reasons, I ended up substituting the snippets' code with my own code and compiling them with the "sudo *make"* command on the "*compiler/linux-checked"* directory. This might prove to be an easy and quick solution, although not ideal.

Note 4: *PhysX SDK* does not support *URDF*. Despite this, there is a *Forum* post where a *PhysX SDK* employee hints that there might be an update in the near future which solves this issue (https://github.com/NVIDIAGameWorks/PhysX/issues/23). PhysX does not use any kind of *xml* files, instead the scene and its actors are entirely defined in C++ code.

Note 5: *PhysX SDK* may be quite confusing, especially with complicated robots possessing several links. Each link (or body, as it is called in *PhysX SDK*) has an origin located at the centre of its body. To create a joint between two links, you must move the origins of both links to the position where the joint will be. This means that it will be necessary to use geometry and make sure that everything lines up perfectly, otherwise it will result in unwanted behaviour during the simulation. From the information I gathered, I believe that with the *PhysX Visual Debugger* it is possible to view the origins of each link, making this process a lot easier. Supposedly, it is also possible to do this with the *Debug Visualization* (https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/DebugVisualization.html?highlight=visualization) but I was not able to get this feature to work, hence I believe that the *PhysX Visual Debugger* may be the easiest solution, as well as the more powerful one.

Parallelisation

I will not get into too much detailed about how I achieved parallelisation in *PhysX SDK* because it was similar to *MuJoco*'s method. I used *OpenMP* and the code can be checked on my repository. However, it is important to note that *OpenMP* is only for *CPU* parallelisation, not *GPU*. For *GPU* parallelisation you would need to use other tools, such as *CUDA*. *PhysX SDK* runs on the *CPU* by default, but by defining *GPU Rigid Bodies* (https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/GPURigidBodies.html) it is possible to run the simulation on the *GPU*. This means that, to run *PhysX SDK* on the *GPU*, the code will need to be constructed from the ground up to do so. It is possible to reuse parts of the code utilised in the *CPU* parallelisation, but not entirely.

I was not able to get the *CPU* parallelisation to work on *PhysX SDK* in the time I had. By some reason that I was not able to figure out, when I increased the number of threads or timesteps I got the error *"invalid parameter : PxRevoluteJoint::setDriveVelocity: invalid parameter".* In addition to this, for smaller number of threads, the code worked fine sometimes, whereas other times it gave me the same error or it registered values that did not make sense (values too big immediately followed by others too low). I believe this was due to some memory problem that I was not able to solve in the time I had left.

PyBullet with PhysX backend

Besides these two physics engines, I also tried to use *PyBullet*. *PyBullet* consists of the *Bullet* engine but with a *Python* interface. However, PyBullet also supports the use of other physics engines, such as *MuJoco, PhysX* and *DART*. In January 2019, *PyBullet*'s founder (Erwin Coumans) announced that he was able to develop a first working version of a *URDF* loader for *PhysX 4.0 backend for PyBullet* (https://twitter.com/erwincoumans/status/1086745188742004736). This means that it would be possible to build a simulation and simply change the physical simulator being used to make the calculations, in one line of code. However, this is still a work in progress and there are still many features which are being developed. Erwin Coumans himself admits that the *PhysX* backend is still very preliminary and there is no support provided (https://github.com/bulletphysics/bullet3/issues/2090). Hence, although using *PyBullet* might reveal to be very interesting, it may also prove to be very challenging since it is still an unfinished product with several bugs. From my experience, I gave up on *PyBullet* because it was not recognising the joints from the *URDF* file and the objects seemed to fly away for no reason. In addition to this, I also had the help from another student which set up parts of *PyBullet* for me. From what I understood, he needed to use *Conda* (https://docs.conda.io/en/latest/), a package management system and environment management, for the *Python* interpreter. I admit that I do not fully understand these concepts and cannot explain them better.

If you wish to use *PyBullet*, then you can start by installing the latest version from this website: https://github.com/bulletphysics/bullet3/releases. In the *“README.md”* you can find instructions on how to install *PyBullet*. Plus, in the “*docs*” directory, you can find the “*pybullet_quickstartguide.pdf*” which explains how to use *PyBullet*.

Note 6: If the reader intends to use *URDF*, then it is important to understand that, when changing the pose of a link, the rotation is first applied to the link and only afterwards is it moved. When the link is rotated, its origin is also rotated. This means that, for example, if you try to move a link to position (3, 0, 0) and rotate it 90° around the z-axis, the link will actualy move to (0, -3, 0). This happens because it will start by rotating the link 90° around the z-axis, aligning the x-axis of the link’s origin with the negative part of the world’s y-axis. Hence, the link will move along the world’s y-axis.

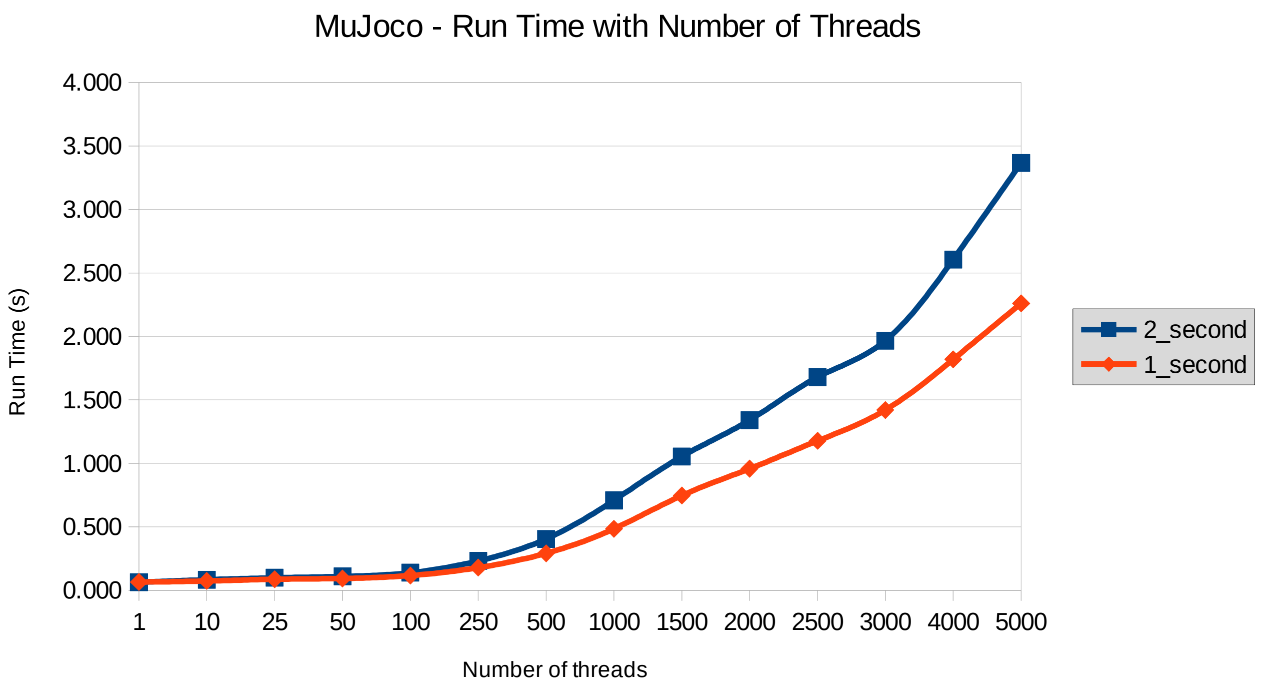
Note 7: Since *PyBullet* uses a *Python* interface and *Python* is slower that other programming languages such as C and C++, *PyBullet* is expected to be slower than using directly *MuJoco* or *PhysX SDK*. However, since the *Python* language is only used as an interface and all the heavy calculations are performed by the physics engines (written in C and C++), the differences should not be that significant. Nonetheless, it would be necessary to run several tests to be sure of this.

Comparison between different physical simulators

I intended to compare *PhysX SDK* and *MuJoco* in terms of performance and simulation speed. However, since I wasn't able to solve the parallelisation problem in *PhysX SDK*, I was not able to do so. Despite this, there have been some articles which compare different physical simulators. All the articles I've gathered can be found in my repo.

One simulator that is fairly recent but might prove to be interesting is *RaiSim* (https://github.com/leggedrobotics/raisimLib). *RaiSim* is a physics simulator developed by the *Robotic Systems Lab* at *ETH Zurich* and it is especially designed for robotics and reinforcement learning. The creators of *RaiSim* have also developed a series of benchmark tests to compare their simulator to others (https://leggedrobotics.github.io/SimBenchmark/). They compared *RaiSim* to *Bullet*, *ODE*, *MuJoco* and *DART*. In their tests, perhaps unsurprisingly, *RaiSim* behaved extremely well. However, the most interesting aspect seems to be the performance of their simulator, which is capable of running the simulations much faster than the others. Despite this, it is still a very early product and it might be challenging to use, since it hasn't had a lot of "market time" and several bugs may persist. In addition to this, since it is so recent, there isn't a great deal of online resources about it yet. However, they do possess a short manual (https://slides.com/jeminhwangbo/raisim-manual#/).

Lastly, with the data I gathered, I was able to make a plot of the program's run time in *MuJoco* with the number of threads used, for a 1 second and 2 second simulation separately. Be aware that "number of threads" is the same as "number of parallel simulations", since each thread is calculating its own slightly different simulation.



Model Predictive Path Integral Control (*MPPI)*

As I previously mentioned, a very simplified version of the Model Predictive Path Integral Control algorithm was used in the simulations. Inside the "*Model Predictive Path Integral Control – Articles*" directory, found in my repo, you will find several scientific articles and one PhD thesis about this topic. In general, *MPPI* can possess some heavy mathematics and be hard to decode. From what I've understood, I do believe they use a very simplified model of the system, hence how they are able to run as many simulations per second.

Tomás Rei dos Santos Ribeiro

tomasribeiro.pt@gmail.com

30/08/2019