Orbital interference, planetary close-approaches detection and memory handling on GPUs

Tesi di Laurea Magistrale in

Space Engineering - Ingegneria Spaziale

Author: **Adriano Filippo Inno**

Student ID: 921033

Advisor: Prof. Camilla Colombo

Co-advisor: Alessandro Masat

Technical-advisor: Lorenzo Bucci

Academic Year: 2020-21

Abstract

The growth of space mission design complexity entails the need of faster computational tools, especially aimed to accelerate the preliminary assessments. Graphical Processing Unit (GPU) applications can grant a huge computational time speed-up with respect to classical Central Processing Unit (CPU) programs, up to two order of magnitude.

The interest in graphic card-based software for space applications is recently growing, especially regarding orbits propagation. GPU software are particularly powerful in large-scale analysis, such as Monte-Carlo statistical methods or grid search optimisations. However, modern propagation software may lack at detecting events during the integration, due to the complexity of efficiently design branch-dependent algorithms on GPUs. Moreover, handling the consequent huge sets of unknown-sized output arrays complicates the development of the software itself.

The aim of the work is to tackle the described problems suggesting algorithms to detect planetary close-approaches and orbital interference with respect to protected regions and space massless points, that exploit logical operations to efficiently run in a GPU based application.

The work presents the improvements on the detection capabilities made on CUDAjectory, an open source orbit propagation software available under ESA Community License. Additionally, the thesis focuses on the optimisation process performed to efficiently handle the output in CUDAjectory.

Keywords: GPU; parallel computing; CUDA; interference detection; orbit propagation; code optimization; mapped pinned memory;

Abstract in lingua italiana

La crescita della complessità progettuale delle missioni spaziali comporta la necessità di strumenti computazionali più veloci, volti soprattutto ad accelerarne le valutazioni preliminari. Le applicazioni basate su GPU possono garantire un'enorme velocità di calcolo rispetto ai classici programmi per CPU, fino a due ordini di grandezza superiore.

Recentemente l'interesse per i software basati su schede grafiche per applicazioni spaziali sta crescendo, soprattutto per quanto riguarda la propagazione orbitale. I software GPU sono particolarmente potenti nelle analisi su larga scala, come i metodi statistici Monte-Carlo o le ottimizzazioni di tipo grid-search. Tuttavia, i software di propagazione potrebbero non riuscire a rilevare eventi durante l'integrazione, a causa della complessità della progettazione efficiente di algoritmi sulle GPU. Inoltre, la gestione dei conseguenti enormi array di output, tuttalpiù di dimensioni non note a priori, complica lo sviluppo del software stesso.

Lo scopo del lavoro è di affrontare i problemi descritti suggerendo alcuni algoritmi per rilevare close-approaches planetari e interferenze orbitali rispetto a regioni protette e punti nello spazio modellati senza massa, sfruttando operazioni logiche per aumentarne l'efficienza in un'applicazione basata su GPU.

Il lavoro presenta i miglioramenti alle capacità di rilevamento apportati su CUDAjectory, un software di propagazione orbitale open source disponibile con licenza comunitaria di ESA. Inoltre, la tesi si concentra sul processo di ottimizzazione volto alla gestione efficiente degli output in CUDAjectory.

Parole chiave: GPU; parallel computing; CUDA; interference detection; propagazione orbitale; ottimizzazione software; mapped pinned memory.

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

Introduction **1**

1 Parallel computing **3**

1.1 The growth of parallel computing 3

1.2 Levels of parallelism 4

1.3 Computer memory types 4

1.3.1 Static random access memory 4

1.3.2 Dynamic random access memory 5

1.4 Brief historical background of GPU computing 5

1.5 Fundamentals of GPU architecture 6

1.6 Basics of CUDA programming 9

1.6.1 Memory hierarchy 11

1.6.2 Heterogeneous programming 12

1.6.3 Performance losses 13

2 CUDAjectory overview **15**

2.1 High level software structure 15

2.1.1 Main library 15

2.1.2 Simulation Python interface 15

2.1.3 Ephemerides retrieval and CUDAjectory management 16

2.2 Reference frames 17

2.3 Dynamics model and orbital perturbations 17

2.3.1 Gravity models 18

2.3.2 Planet oblatness perturbation 20
2.3.3 Solar radiation pressure 20
2.3.4 Third body perturbation 20
2.4 Events detection 21
2.4.1 Collision check 21
2.4.2 Close-approach check 22
2.4.3 SOI and SOI-crossing checks 22
2.4.4 Altitude check 22
2.5 Integration scheme 23
2.5.1 Step size control 23
2.6 Simulation management 24
2.6.1 Data preparation and allocations 24
2.6.2 Kernel launch and execution 25
2.6.3 Samples clustering and terminal conditions 25
2.6.4 Data handling 26

3 Close-approach event **27**
3.1 Previous implementation 27
3.1.1 Closest celestial body retrieval 28
3.1.2 Close-approach event condition 29
3.1.3 Sample status 30
3.1.4 Bisection scheme 30
3.1.5 Optimisation margin 33
3.2 New implementation 33
3.2.1 Sample status 33
3.2.2 Bisection scheme 33
3.2.3 Infinite looping 34
3.3 Validation 35
3.3.1 Setup 35
3.3.2 Results 36
3.4 Computational time comparison and conclusions 37

4 Earth protected regions events **39**
4.1 Protected region convention 39
4.2 Requirements 40
4.3 Bisection scheme limitations 40
4.4 LEO protected region 41
4.4.1 Adaptive step refinement 41

4.4.2 Check implementation . . . 42
4.5 GEO protected region . . . 43
4.5.1 Check implementation . . . 44
4.6 Validation . . . 44
4.6.1 Setup . . . 45
4.6.2 Results . . . 45

5 Massless bodies collision event 49
5.1 Current limitations . . . 49
5.2 Libration points . . . 50
5.2.1 Collinear points . . . 51
5.2.2 Triangular points . . . 51
5.3 Check implementation . . . 52
5.3.1 LEO protected region similarities . . . 53
5.3.2 Adaptive step refinement . . . 53
5.4 Validation . . . 54

6 Output handling optimisation 57
6.1 Output handling . . . 57
6.2 Pinned memory . . . 58
6.3 Improved output handler . . . 59
6.3.1 New output array structure . . . 59
6.3.2 Data structure choice . . . 60
6.4 Graphic card compatibility . . . 61
6.5 RAM allocation compatibility . . . 61
6.6 Checks support . . . 61
6.7 SOI output-handler decoupling . . . 62
6.8 Performance comparison . . . 62

7 Study cases 65
7.1 Lunar mission: collision assessment . . . 65
7.2 SEL5 dispersion analysis after a collision . . . 70

8 Conclusions and future developments 73

Bibliography 75

A Spice kernel types **79**

List of Figures **83**

Acknowledgements **85**

Acronyms and Initialisms

AI	Artificial Intelligence
AoP	Argument of Pericentre
AU	Astronomical Unit
CPU	Central Processing Unit
CUBE	CUdajectory Binary Ephemerides
CR3BP	Circular Restricted three Body Problem
DAF	Double precision Array Files
DRAM	Dynamic Random Access Memory
ESOC	European Space Operation Center
GB	GigaBytes
GEO	Geostationary Earth Orbit
GPU	Graphical Processing Unit
GTS	Giga Thread Scheduler
IADC	Inter-Agency space Debris coordination Committee
ICRS	International Celestial Reference System
ICRF	International Celestial Reference Frame
LEO	Low Earth Orbit
LLO	Low Lunar Orbit
LTO	Lunar Transfer Orbit
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MJD2000	Modified Julian Date 2000
ODE	Ordinary Differential Equation
PCB	Printed Circuit Board
RAM	Random Access Memory

RAAN	Right Ascension of the Ascending Node
SRAM	Static Random Access Memory
VRM	Voltage Regulation Module
VRAM	Video RAM
SPK	Satellite and Planet Kernel
SEL	Sun-Earth Libration
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SISD	Single Instruction Single Data
SM	Streaming Multiprocessor
SNAPPshot	Suite for the Numerical Analysis of Planetary Protection
SRP	Solar Radiation Pressure
SOI	Sphere Of Influence
TA	True Anomaly
TOF	Time Of Flight

Introduction

The interest in GPU programming is continuously growing, especially thanks to deep learning and Artificial Intelligence (AI) training which are nowadays popular scientific fields. However, even if the advantages of using graphic cards for scientific purposes can be significant, the employment of GPUs for space applications is not yet widespread. The reason behind that is partially related to the preference in exploiting deep-rooted tools in the space field, rather than utilising the latest computer technologies. Indeed, for obvious reasons, reliability is always put in front in the space industry. Moreover, GPU programming can be hard, and it is not the focus of space engineers. This work side-aims to thrill the readers in using or developing GPU based software.

Chapter 1 introduces the reader to the work topics illustrating the growth of parallel computing for scientific purposes, the basics of GPU architectures, and the fundamentals of graphics card based programming. This work focuses on the improvements made in CUDAjectory, an orbit propagation software written with the Nvidia [32] CUDA libraries. In order to help the reader in approaching the development of new programming features, Chapter 2 gives an overview of CUDAjectory, showing its capabilities and discussing the main limiting aspects. CUDAjectory is a powerful orbit propagator, especially viable for large-scale analyses. It allows to propagate millions of orbits in a much faster way with respect to other CPU applications [17]. This work aims to increase the number of analyses that the software grants, in order to reach a bigger user pool.

The current space debris problem is getting critical: the number of objects, their combined mass and volume is steadily increasing since the beginning of the space exploration, leading to involuntary collisions between operational payloads and space debris [35]. The stricter debris mitigation guidelines lead to the need of new mission-design assessments such as interference analyses with respect to the Earth protected regions. With the aim to perform this type of analyses, Chapter 4 presents the development of an event algorithm to detect the orbital interference with respect to the crowded near-Earth regions. Another developed event algorithm comes from the increasing interest to exploit the Libration points for scientific missions, mainly the Sun-Earth ones. Interception and interference analyses with respect to the Libration points are also extremely valuable applications that can be utilised in many mission designs. Chapter 5 shows the development of the algorithm devoted to detect the collisions with respect to massless bodies, such as the Libration points. Together with the development of the new algorithms, this work explains the optimisation process performed on the output handling capability of CUDAjectory. Chapter 6 presents the implementation of an efficient type of data handling on GPUs using the pinned mapped memory.

1 | Parallel computing

This chapter presents the basics of parallel computing, focusing on GPU architectures and the fundamentals of GPU programming.

1.1. The growth of parallel computing

Since the distribution of the first CPUs in the market in the 70s, the most adopted strategy to increment its computational power was to speed up the operating clocks. This approach led in the 2000s to physical limitations to the CPU transistors manufacturing process, due to the continuous transistor miniaturization process [21]. The companies, trying to sustain the computational power progression, started to produce multi-core CPUs; thanks to this choice, despite a penalty in terms of frequencies, the overall computational power kept growing. Multi-cores CPUs indeed allow to do more computations in parallel.

Traditionally, computer software has been developed for serial computation. Algorithms are constructed and implemented as serial streams of instructions, where only one instruction may execute at a time. In parallel computing, a computational task is typically broken down into many sub-tasks, often very similar or identical to each other, that can be processed independently.

The so called *core* is the computing unit of a processor and in multi-core CPUs each core is independent and can access the same memory concurrently. Multi-core processors have brought parallel computing to desktop computers. Thus parallelisation of serial programs has become a common programming task. Consequently, new features have become available for the developers, allowing them to use the parallel computing tools for building faster applications. In the recent years, a technology called hyper-threading has been established, granting the capability to have multiple (usually two in CPUs) computational unit in each core. The computational units are then called *logical cores* or *threads*.

1.2. Levels of parallelism

Different types of parallelism exist. Michael Flynn defined one of the most common classification for parallel programs, known as Flynn's taxonomy [16]. Flynn classified programs by the capability of using a single or multiple sets of instructions, and whether those instructions were applied on a single block or multiple blocks of data.

The Single Instruction Single Data (SISD) classification is referred to serial programs. The Single Instruction Multiple Data (SIMD) classification identify programs where the same instructions are repeatedly executed over a data set. This type of parallelism is discussed in details in this paper, being one of the most used way to work on GPUs. Multiple Instruction Single Data (MISD) is not a common practice and barely adopted. On the other hand, Multiple Instruction Multiple Data (MIMD) programs are by far the most common type of parallel programs.

1.3. Computer memory types

In SIMD programs, the same set of instructions is executed in parallel by the CPU or the GPU threads. The data set is the input of the program and for each data the corresponding output is computed in parallel. The developer has to manage multiple sets of data and consequently, handling the memory transfers is one of the most important task in parallel programming.

Sometimes memory accesses have to be synchronised, in order to avoid a phenomenon called *race condition*. Race condition refers to data that can possibly be read before written or vice-versa. This type of issue is one of the most common class of problems in GPU programming. For this reason, memory layout, hierarchies, and accessibility are key aspects of parallel programs [33].

Data and machine code are stored in the Random Access Memory (RAM). The two widely used forms of RAM are the Static Random Access Memory (SRAM) and the Dynamic Random Access Memory (DRAM), which can be found in almost every CPU and GPU [37].

1.3.1. Static random access memory

SRAM is typically used as *cache memory*, a low latency access memory used to store instructions and frequently used data. The cells of data are built thanks to transistors, making this type of memory simple, reliable, and the fastest in the market. However, the storage capacity of SRAM is typycally low (8-50 MB) and it is much more expensive to be manufactured than DRAM [26].

1.3.2. Dynamic random access memory

Usually DRAM is the main memory of a computer. Instead of transistors, capacitors are utilised to store data and are coordinated by integrated circuits [26]. There are many types of latency involved in memory transactions. By definition, latency is expressed in clock cycles needed to perform an operation, and so it represents a delay between a command issue and its execution.

DRAM is composed by elements, called *cells* that store the data. A DRAM module can be imagined as few 2D matrices, each one accessible by the CPU; the different matrices are called *banks*, which lie in physically different parts of the memory chips composing a DRAM module [26]. Multiple latencies are associated to the read or write operations to a specific element of these matrices. These latencies are also called *timings*. Each latency is associated to a specific electrical process involved in memory transactions. The latencies associated to the most frequent command issues are called *primary* timings, and are: *column access strobe* delay, *row-to-row* delay, and *pre-charge* delay [26]. Timings are specific to the memory chips used by the manufacturers to build the DRAM modules.

Using again the matrix analogy, an element in DRAM can be accessed just if the row is activated via a pre-charge activation, ruled by the pre-charge delay. Rows are generally of 2-32 kB each, and once a row is active, the access to each element is faster and governed by the column access strobe delay. Therefore, the latency to access consequent cells is much lower compared to retrieving memory-scattered elements and should possibly be avoided.

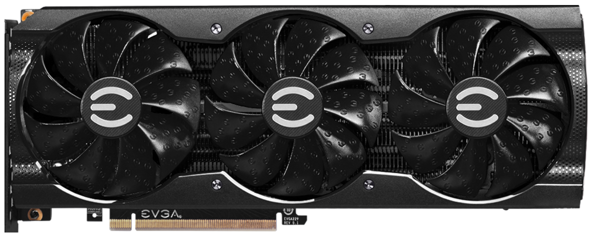
1.4. Brief historical background of GPU computing

In the 1990s, the growth in popularity of graphically driven operating systems such as Microsoft [24] Windows 95 led to the creation of a market for a new type of processor [37]. The first 2D display accelerators for consumers were released, offering to assist graphical operating systems and applications. At the same time Silicon Graphics was commercializing some three-dimensional graphics tools for scientific and cinematic purposes. Silicon Graphics released OpenGL [36] in 1992, a library of APIs intended for 3D graphics.

Approaching the 2000s, the 3D graphics power demand increased drastically, mostly due to the growth of PC gaming. Nvidia and ATI Technologies started producing graphics cards that were affordable for the consumers. From a parallel computing point of view, the release by Nvidia of the GeForce 3 series, in 2001, is probably the most important milestone [37]. That generation had the first chip to implement the DirectX 8.0 standard, which gave for the first time a lot of control to the developers thanks to a programmable pipeline. Unfortunately OpenGL and DirectX at the time were still the only way to build application for GPUs, no specific application was available for scientific programming. The developers found a way to adapt their problems by making them appear as a 3D rendering to the GPU. However, this trick was time demanding. Furthermore, no debugging tools were available. In 2006 Nvidia released its first GPU with CUDA architecture, the GTX 8800. This chip included lots of components aiming also to general-purpose computing, paving the way for a long term GPU scientific programming. Nowadays several general-purpose GPUs are available in the market. Nvidia is the market leader, followed by AMD [2] (previously ATI Technologies). The most adopted GPU computing languages are OpenCL and CUDA. The main difference between them is that OpenCL libraries are compatible with all the genaeral-purpose GPUs in the market, while CUDA is the proprietary framework of Nvidia and its compatible with Nvidia genaeral-purpose GPUs only.

1.5. Fundamentals of GPU architecture

This section aims to let the reader understand the basics of GPU architectures. Since the framework of this work totally relies on CUDA, this section presents just the fundamentals of Nvidia graphic-card architectures. The terms graphic chip and graphic card are often mistaken: in Fig. 1.1b and Fig. 1.1a an example of a graphic card and its Printed Circuit Board (PCB) layout (the actual board where the hardware and the circuits are located) is shown. The GPU is the chip that is located in the centre of the PCB. In the example, the GPU is designed by Nvidia, while the graphic card is assembled by EVGA [13], another computer hardware company. Graphic cards nowadays are complicate systems composed by lots of hardware inside: besides the GPU, a Voltage Regulation Module (VRM) is always present as well as multiple DRAM memory chips and a thermal dissipation subsystem.


(a) Graphic card.

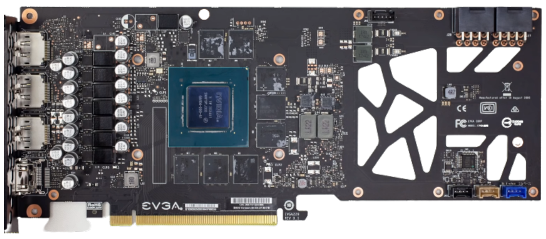

(b) PCB layout.

Figure 1.1: EVGA RTX 3060ti ftw3 graphic card and its PCB layout [14].

The DRAM of video cards is usually called Video RAM (VRAM). VRAM can be classified for generations, which basically imply a range of operating frequency and timings, or for capacity. The latest version of VRAM chips is called GDDR6X, where GDDR means Graphical Double Data Rate, while 6X represents the actual generation number. GDDR6X chips are extremely fast but also require more heat dissipation with respect to the previous generations; for this reason, these chips are currently used in consumer graphics cards only. Nvidia CUDA GPU micro-architectures together with the release date are listed here in chronological order of release:

- Tesla, September 2006;
- Fermi, April 2010;
- Kepler, April 2012;
- Maxwell, February 2014;
- Pascal, April 2016;
- Volta, May 2017;
- Turing, August 2018;
- Ampere, September 2020;
- Hopper, March 2022.

Although a detailed dissertation about these architectures is behind the scopes of this work, GPU programming is hardware-dependant. A Fermi GPU is presented, to have some general insight on the programming paradigms. In Fig. 1.2 an example of the internal architecture of a Fermi GPU is shown; it is possible to count 16 Streaming Multiprocessor (SM)s, which are clusters of cores. Each SM is identical to the others and can read/write to a shared L2 cache memory of 768KB. A memory controller regulates the data transfers between the L2 cache (which is phisically located inside the GPU chip) and the VRAM, which is instead situated outside the GPU, as it is shown in Fig. 1.1b. For this reason, VRAM it is sometimes called off-chip memory. The GPU can communicate with the CPU through the PCIe interface, a set of links that is part of the circuitry of the PCB of the graphic card and the motherboard of the computer. Finally a Giga Thread Scheduler it's present and its job is to schedule the computational tasks, dividing the load among the SMs.

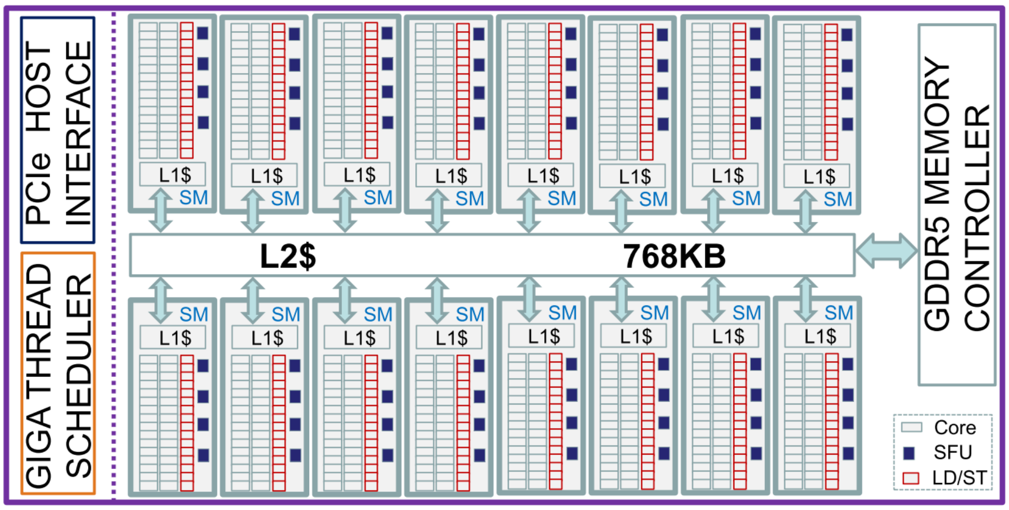


Figure 1.2: Example of a Nvidia Fermi GPU architecture [39].

The previous GPU example uses the Fermi GF110 SM micro-architecture, shown in Fig. 1.3. The Instruction Cache holds the instructions issued by the Giga Thread Scheduler (GTS). The two warp schedulers are responsible to divide the instructions into a set of schedules for each execution unit. The execution units are the cores, Load/Store(LD/ST) units and Special Function Units (SFUs). LD/ST are used to queue memory transfers; instead, a SFU is responsible for computing special functions, for example trigonometric function. Each core has a Floating Point (FP) and an Integer (INT) execution unit, differently issued by the warp scheduler depending on the requested instruction or data. Each SM has a 128KB register block that contains 32,768 registers, each one 32-bits sized. Rather than having a dedicated register to each core, like in a CPU, the registers can be accessed by each computational unit. Moreover, a 64KB cache memory is used as L1$ and shared memory between the cores.

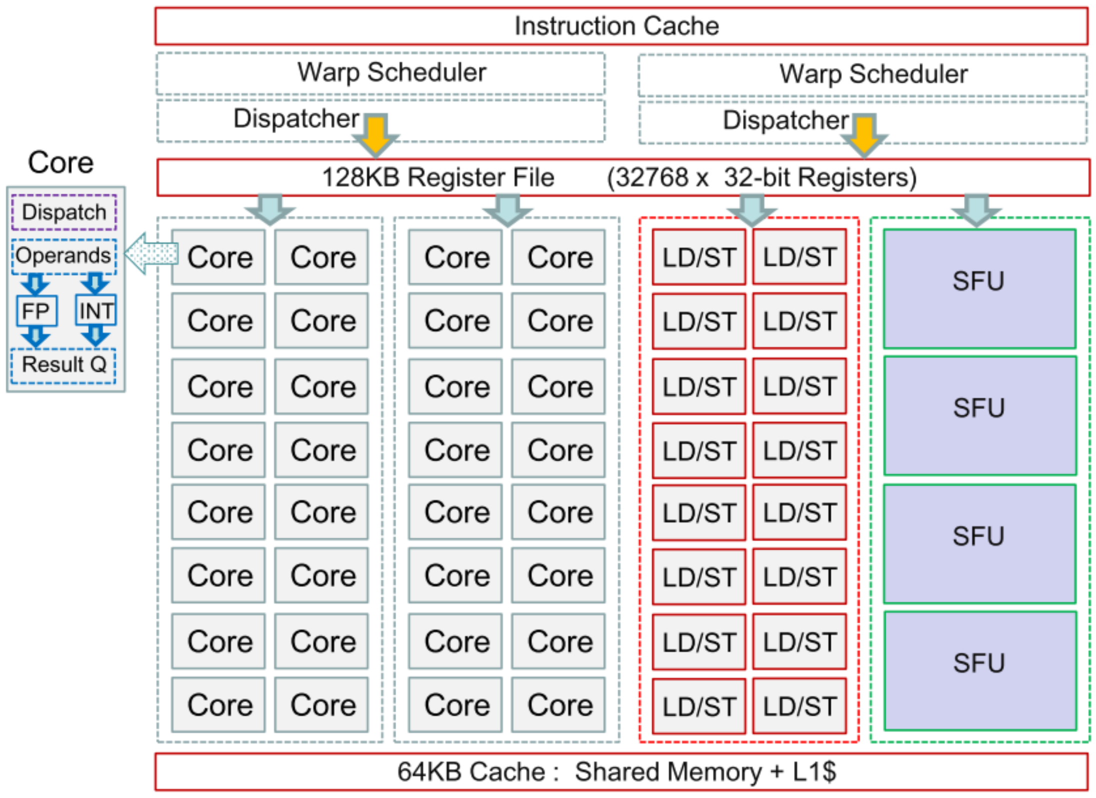


Figure 1.3: Fermi GF110 microarchitecture [39].

As Fig. 1.2 shows, the amount of cores available inside a GPU can be much higher with respect to a CPU. This implies an high instruction throughput, much higher with respect to a CPUs within the same range of price and power consumption [33]. The Chips are usually comparable in terms of dimension but GPUs are designed to have more transistors devoted to data processing whereas more are used for cache and flow control in CPUs.

1.6. Basics of CUDA programming

GPUs are co-processors cooperating with a CPU and never working as standalone devices. This leads to the programming nomenclature in which, since the CPU is hosting the program, it is called *host*. The one ore more GPUs are called *devices*. The host is responsible of data pre-processing, memory allocation (also on the device), and issuing device instructions. The set of commands ran on the host are called host code. On the other hand, the part of the code that is written to run on the device, is called device code.

The main functions that run on the devices are called *kernels*. Kernels can only call other device code functions, while is not possible to command any instruction on the host from the device. A typical CUDA code structure is:

- host data pre-process;
- host and device memory allocation;
- kernel input data transfer from host to device;
- kernel launch and execution;
- kernel outputs data transfer from device to host;
- host data post-process;
- free the allocated memory, on both host and device.

The parallelisation given by CUDA is exploited with the kernel execution. The device code written inside a kernel is the instructions set which is executed by the threads over a data set. The type of parallelism is of SIMD type, which has been actually renamed to Single Instruction Multiple Threads (SIMT) by Nvidia, since the same instructions are executed on many threads simultaneously. The developer can also manage how many threads are launched for a given kernel. In CUDA programming, kernel launches are automatically scaled, which consist in a peculiar feature: if the number of threads requested by the developer is higher than what the device can actually run, the whole set of threads to be ran is automatically divided in groups of threads, issued one by one.

Small chunks of instructions are executed by *warps*. Warps are group of 32 threads that are forced to execute the same exact instructions set; the total number of threads to be launched have preferably to be a multiple of 32, otherwise at least one warp cannot be fully populated.

Blocks are groups of threads. Threads of blocks can be collected in one-dimensional, bi-dimensional or tri-dimensional groups. A block may also contain up to 32 warps, leading to maximum 1024 threads per block. The maximum number of thread per block is hardware peculiar, but can be retrieved using `cudaGetDeviceProperties()` to get the whole set of device properties, and then checking the field `maxThreadsPerBlock` of the returned structure. Finally, groups of blocks are called *grids*. Grids can be organised as 1D, 2D, and 3D groups as for the blocks.

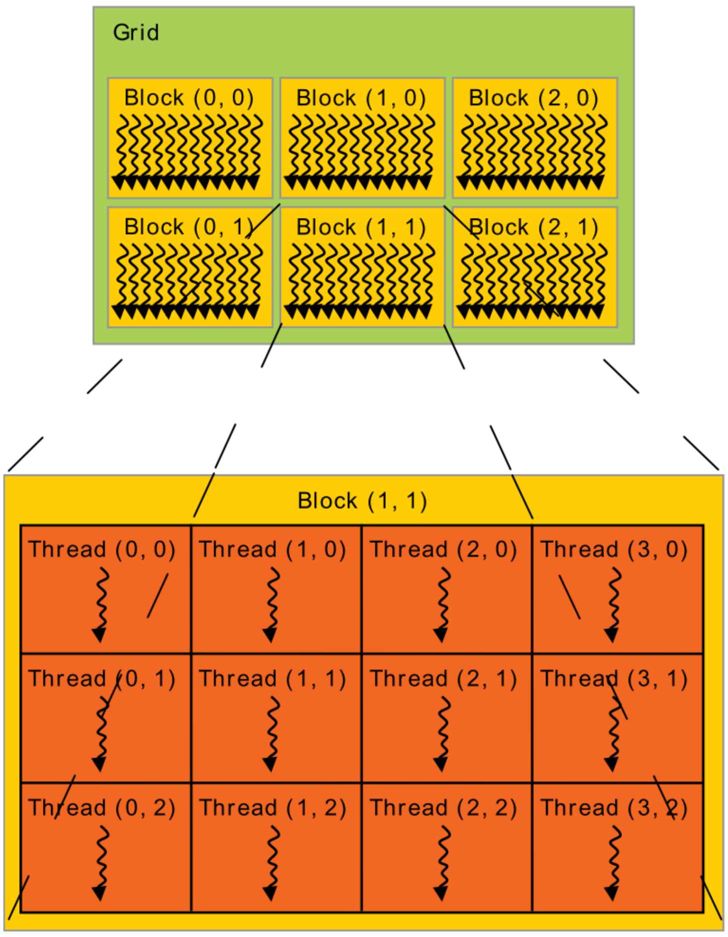


Figure 1.4: Graphical view of a bi-dimensional block and grid arrangement [33].

In Fig. 1.4 an example of bi-dimensional blocks and grids layout is shown. Each block executes 12 threads, and each grid is composed by 6 blocks. The dimensions of grids and blocks can also be different (e.g. 1D block and 2D grids).

The threads organisation of a kernel launch is specified with two variables: the number of blocks per grid and the number of threads per block. The number of warps cannot be specified: warps are automatically issued by the warp schedulers inside each SM. Kernels are often started using the `maxThreadsPerBlock` property to define the number of threads per block.

1.6.1. Memory hierarchy

Threads have some private on-chip memory allocated per-self: the **registries**; registries are the fastest memory type of a graphic card [33]. However, the developers cannot directly decide to store data inside registries. Registry storing operations are directly managed by CUDA libraries. The lifetime of registries is shared with the kernel: after the kernel execution the registries are automatically restored. The output of the kernel have to be manually allocated outside the registries, and stored during the computations.

Global memory is much larger than any other type of memories; usually, modern workstation devices have 16 GigaBytes (GB) or more available. This memory can be accessed by each block of threads and the read/write operations to it have the highest latencies [33]. Consequently, the goal is to utilise it just for loading inputs and off-loading outputs from the graphic card.

Shared memory is shared between the threads of a block. It lies on-chip and so has much higher bandwidth and lower latency than global memory. The location of the shared memory has been shown in Fig. 1.3 and it is shared between the cores inside an SM. Shared memory is divided between one or more blocks running inside an SM, it can be read or written by each thread of a block, and has the lifetime of the block itself [33].

Constant memory resides in the constant cache device memory. For example, in Fig. 1.2 constant memory is cached inside the so-called L2$. In general, it lies on-chip and it is shared among all the blocks. It must be allocated prior to any kernel execution and the stored variables are globally available. Constant memory is read only, its purpose is keeping constant variables available for the whole application lifetime. The size of the constant memory is 64KB for all the graphic chips [33]. Functions or methods called inside the kernel might occupy some constant memory; for this reason, the developer should carefully manage the constant memory allocations. All the constant memory declarations associated to the kernel can be checked at compilation time using the `-ptxas-options=“v”` flag [34].

Texture memory and surface memory are read-only, both with the same location and lifetime of the constant memory. Texture memory is optimised for bi-dimensional data, and allows better performance than constant memory if threads within a warp access addresses that are close together in 2D [33].

1.6.2. Heterogeneous programming

The CPU might also have a major computational role throughout the program: while the kernel is executing, the developer can use the CPU to perform additional computations. This strategy is typically exploited when the application requires some sort of kernel output post-processing. Considering the classic code layout presented in Section 1.6, the total number of threads to be launched is typically higher than the threads that can physically run in parallel. The developer can therefore split the computation in multiple sequential kernels, using the CPU to post-process the kernel output, while the GPU is still executing some computations.

Nowadays, most of the GPUs can also perform memory copies from and to the global memory during the kernel execution and while the host is executing other calculations. These asynchronous behaviours are regulated by *streams*. A stream is a set of device instructions that can be synchronised with many intrinsic functions and barriers. The developer must pay attention on avoiding race conditions. If fully utilised, asynchronous streams offer huge performance optimisations.

1.6.3. Performance losses

The five most common performance losses in CUDA programming are called *registry spilling*, *partial warp occupancy*, *warp divergence*, *memory coalescing*, and *bank conflict*. The behaviour of a GPU is deterministic and so the developer must try to lower the performance drop associated with this phenomena.

Registry spilling

In Section 1.6.1 the role of the registries has been presented. Being the fastest memory available to the threads, registries have to be utilised as best as possible. However it is a sparse resource and if the size is exceeded, a phenomenon called registry spilling happens: the extra data are sent to off-chip memory (VRAM), with much higher latency. This part of the VRAM is called *local memory*, because its cell locations are still thread-private. Registry spilling might be reduced by increasing the number of registries allocated per thread, which the developer is allowed to do.

Partial warp occupancy

The occupancy is the percentage of working warps upon the total number of warps available. Warps may not being used for multiple reasons. Warps are scheduled by the warp schedulers of an SM and during the latency of these operations the warps may not be used to perform computations. This loss of performance is in general hard to address, since it involves both the computational time of scheduled instruction set and the scheduler latency.

Warp occupancy may also be affected by registry and shared memory management: if the number of registry per thread is modified by the developer, the device might not be able to starts all the warps on some SM. On the other hand, if the shared memory allocated per SM is too much, the device is not able to utilise each SM. Registry spilling and partial occupancy are mutually influenced. In many applications it can be crucial to trade off the number of registry per thread.

Warp divergence

As discussed in Section 1.5, CPUs have a lot of transistors devoted to flow control and branch prediction. For example when dealing with if-else statements in a SIMD/SIMT logic, it is clear that the set of instructions is path-dependent. Data may not be applied to the same instructions blindly. Threads inside a warp are forced to execute the same instruction. As a consequence, path-dependent instructions lead to stalled threads that have to wait before a certain path is concluded before proceeding to the next one. A CPU can be thought as a smarter piece of hardware in this context, since it is able to counteract this behaviour.

Memory coalescing

Global memory accesses by the threads of a warp are coalesced. Depending on the size of the data to be accessed by the threads and distribution inside the global memory, the accesses can be coalesced in one or more memory transactions [33]. Usually the higher the number of transactions, the more the number of the unused words are transferred, leading to a penalty in the instruction throughput [33].

Bank conflict

To achieve a bandwidth peak, shared memory is split into equally-sized DRAM on-chip banks, which can be accessed simultaneously. Memory read/write commands issued to N shared memory addresses that lie in N distinct banks can therefore be operated simultaneously, granting a bandwidth peak that is N times higher with respect to the bandwidth of a single bank [33]. Instead, whenever the addresses of a memory request fall in the same memory bank the full bandwidth is not exploited. This phenomenon is called bank conflict and whenever it happens the hardware has to split the memory request into multiple serial requests.

2 | CUDAjectory overview

CUDAjectory is a parallelised software for trajectory propagation, based on CUDA. The name CUDAjectory comes from the key words CUDA and trajectory. The software has been developed in the last years to propagate millions of user-defined bodies in parallel, called *samples*, exploiting the ephemerides of the celestial bodies. The samples are supposed to not influence the gravity field. The user must define the set of samples providing a file that contains the initial states and the epoch of each body. CUDAjectory is freely available under ESA Community License [11]. The goal of this work is to extend the event detection capabilities and to optimise the computational time of the software.

This chapter gives an overview of CUDAjectory before the changes made by the author. The objective is to help the reader in understanding the organisation of the software prior to approach the development of the new features.

2.1. High level software structure

CUDAjectory is organised in three main parts: a main source code library, a Python simulator interface, and an ephemeris retrieval suite.

2.1.1. Main library

The main library of the software is called TRAMP, which stands for TRAjectory Massively Parallelised. This library contains all the functions needed to perform the orbit propagation, the event detections and the output handling. The contained code is written in C/C++/CUDA. This source code is not a stand-alone piece of software, and it is integrated with the Python interface.

2.1.2. Simulation Python interface

The Python interface is the interface that the user utilises to run the simulations. It has been developed to let the user interact with the application through simple Python scripts, and to reach a bigger user pool; indeed, Python is nowadays more utilised in scientific programs. The aim of the software is to provide a set of tools allowing the user to perform different types of analyses. For this reason, the Python interface does not provide any post-processed results, which are leaved to the users themselves. The user is able to define the simulation settings thanks to a YAML configuration file, read by the Python interface.

2.1.3. Ephemerides retrieval and CUDAjectory management

The user has also to provide the ephemerides of the celestial bodies involved in the simulation. The ephemerides can be downloaded from the NAIF-JPL website [30]. The data are stored in binary files, named as *deXXX*, where *de* stands for *development ephemerides* and it is followed by a three-digits number, denoting the unique file.

Data are listed in a format called Double precision Array Files (DAF), common to every NAIF-Spice file. Spice is a toolbox available under JPL license that can be used to create, manage and utilise ephemerides. Unfortunately, it is only optimised to run in a CPU environment, and cannot be directly integrated inside CUDA kernels. Consequently, an ephemerides toolkit of CUDAjectory has been specifically developed to exploit the data stored inside deXXX files. This files are also called *kernels*, and in this work are refereed as Spice-kernels to avoid confusion with respect to CUDA-kernels.

Despite Spice-kernels could be directly read in a GPU program, a custom file format has been studied in order to maximise the ephemeris evaluation performance [17]; this file format is called CUdajectory Binary Ephemerides (CUBE). The ephemerides toolkit is composed by two main parts: one is devoted to read and convert the Spice-kernels into CUBE files, written in Python, and the other one is contained inside the main CUDAjectory library and it provides the set of functions to retrieve and exploit the ephemerides data at running time.

Many Spice-kernels types exist: data are written in different layouts accordingly to the type. Each type corresponds to a different interpolation technique and is more suitable for storing data of specific bodies, depending on the motion. Currently, CUDAjectory is compatible with type 2 and 3 only. This compatibility represents a limiting factor in the developments, as explained in Chapter 5. Appendix A gives a brief explanation of the format and the usage of these files.

2.2. Reference frames

A *reference system* means a set of conventions and models required to define at any time a generic triad of axes. Conventionally, it is defined by specifying its origin and direction of the axes as well as the scale of the system. *Reference frames* are required to describe the positions and velocity of a body in time; it specify a set of three ordered directions, possibly time dependent, in which position and velocities are projected. Reference frames are also referred as practical *realizations* of reference systems, achieved through a set of bodies or points utilised to define the origin of the axes. A reference frame can be inertial or non-inertial. Instead, a *coordinate system* specifies the mechanism to locate a point within a reference frame.

The International Celestial Reference System (ICRS) is the most adopted reference system for describing the motions inside the Solar System, and have also been used in CUDAjectory. Its origin is located at the Solar System barycentre and the direction of the axes is defined by the adopted position of 295 extra-galactic sources [29].

The International Celestial Reference Frame (ICRF) is the realization of the ICRS and it is used to describe the position and velocity of space objects. Furthermore, since the extra-galactic sources are so distant, the ICRF is practically fixed in time; so, no rotation is needed to describe the motion of a body inside the Solar System. The realization of the ICRF was made to coincide almost exactly with the J2000 (or EME2000) reference frame [29], which is based on distant stars.

2.3. Dynamics model and orbital perturbations

The equation of motion in CUDAjectory are integrated in its Cartesian form, so the state vector ($\boldsymbol{x}$) is expressed as:

$$\boldsymbol{x} = \begin{pmatrix} \boldsymbol{r} \\ \boldsymbol{v} \end{pmatrix} \tag{2.1}$$

where $\boldsymbol{r}$ is the position vector and $\boldsymbol{v}$ is the velocity vector, both expressed in the ICRF frame. Thus the state is a 6-components vector, denoted as:

$$\boldsymbol{x} = \begin{pmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{pmatrix} \tag{2.2}$$

The dynamics model adopted is the two-body perturbed model, that can be expressed in the following form [5]:

$$\ddot{\boldsymbol{r}} = -\nabla U + \boldsymbol{a_p} \tag{2.3}$$

where $\boldsymbol{a_p}$ is the acceleration induced by the orbital perturbations, and U is the potential. The user can set which perturbations are active during the integration thanks to the configuration file.

The propagation of the samples is carried out with the *patched conical approach*: the trajectory of each sample is propagated piece-wise (with the two-body perturbed dynamics) around the current centre of gravity, and then patched to the next if any change of gravitational centre is detected [5]. The current centre of gravity is defined through the classical Sphere Of Influence (SOI) parameter, which defines the radius around each celestial body inside which the gravity it is mostly dependent to. The SOI is computed by default as [5]:

$$R_{SOI} = a\frac{m}{M}^{0.4} \tag{2.4}$$

where M is the primary mass involved, while m is the secondary; a is the semi-major axes of the orbit of the secondary body around the primary.

2.3.1. Gravity models

The user can choose between three gravity models: *point masses*, *spherical harmonics* and *point mascon*.

Point masses

The default model is the point masses; thus, the gravity potential is expressed as:

$$U = -\frac{\mu}{\boldsymbol{r}} \tag{2.5}$$

So recalling the Eq. (2.3), the equation of motion can be written as:

$$\ddot{\boldsymbol{r}} = -\frac{\mu}{\boldsymbol{r}^3} + \boldsymbol{a_p} \tag{2.6}$$

where μ is the gravitational parameter of the gravity centre.

Spherical harmonics

The irregular distribution of the mass of a celestial body implies a dependency between the position vector and the gravity potential function [5]. In the body fixed frame, the gravity potential can be expressed as function of two $n \cdot m$ body-dependent sets of coefficients (S_{nm} and C_{cm}) and two $n \cdot m$ position-dependent coefficients (W_{nm} and V_{cm}) as follows [25]:

$$U = -\frac{\mu}{R}\sum_{n=0}^{+\infty}\sum_{m=0}^{+\infty}(S_{nm}W_{nm} + C_{nm}V_{nm}) \tag{2.7}$$

where n is the degree, while m is the order of the spherical harmonics. Exploiting this notation, the gradient of the gravity potential can be computed by recursive formulas. The complete mathematical dissertation and how it is implemented in CUDAjectory can be found in [17, 25].

The user has to provide a file containing the two $n \cdot m$ body-dependent coefficients (S_{nm} and C_{cm}) if this gravity model is selected. The spherical harmonics gravity model is used by the software to compute the gravity field around the initial central body.

Point mascon

The point mascon gravity model is based on representing the central body with discrete point masses, where the concentrated masses are placed in a spherical surface of radius R. It is an alternative model with respect to the spherical harmonics one, and the point mascon method is advantageous in terms of simplicity, computational effort and easiness to be parallelised. However, the difference with respect to spherical harmonics becomes more and more significant in lower orbits [25].

Dealing with n masscons, the gravity potential can be modeled as presented in [17]:

$$U(\boldsymbol{r}) = -\frac{\mu}{r}\left(1 - J_2\left(\frac{R}{r}\right)^2\left(\frac{3z^2}{2r^2} - \frac{1}{2}\right) + \sum_{j=1}^{n}\frac{\mu_j}{|\boldsymbol{\rho}_j - \boldsymbol{r}|}\right) \tag{2.8}$$

where μ_j is the potential of each mascon and $\boldsymbol{\rho}_j$ its location. The complete procedure to compute the unknowns can be found in [25]; the unknowns have to be given as inputs in CUDAjectory, passed with an additional input file.

2.3.2. Planet oblatness perturbation

The zonal harmonics contribution that mostly depends on the effect of the body oblateness is the second, and it is usually the highest perturbing term of the harmonics [5]. This effect is quantified by a coefficient called J_2 , which depends on the celestial body considered. The default J_2 values of the main celestial bodies are provided as input. The J_2 perturbation can be activated by the user in the configuration file, if the point masses gravity model is selected.

2.3.3. Solar radiation pressure

A body exposed to the Sun flux is accelerated by the resultant of a pressure distribution called Solar Radiation Pressure (SRP) [5]. The user can decide to activate the computation of this acceleration in the configuration file and it is computed as:

$$\boldsymbol{a}_{\boldsymbol{p}}^{SRP} = \nu P_0 AU^2 C_r \frac{A_r}{m} \frac{\boldsymbol{r}}{|\boldsymbol{r}|^3} \tag{2.9}$$

where $\boldsymbol{r}$ is the position of the body with respect to the sun, P_0 is the SRP at one Astronomical Unit (AU), C_r is the reflectively coefficient (body-dependent), A_r is the area exposed to the solar flux, m is the mass of the spacecraft and ν is the so-called shadow factor. The shadow factor is unitary if the body is completely exposed to the light; instead, if the body it is in a total umbra condition, the shadow factor is equal to zero, meaning that no SRP is acting to the body. Partial illuminated conditions, giving a shadow factor between zero and one, have been implemented and the algorithm scheme can be found in [17].

2.3.4. Third body perturbation

Celestial bodies besides the integration centre generates a perturbing gravity force acting on the samples. The user may select a list of celestial bodies which perturb the dynamics the samples. The perturbing acceleration generated by the selected bodies that are not the current propagation centre, generally called *third-body perturbation*, is computed as:

$$\boldsymbol{a}_{\boldsymbol{p}}^{3rd} = \mu_{3rd} \left(\frac{\boldsymbol{r}_{3rd} - \boldsymbol{r}}{|\boldsymbol{r}_{3rd} - \boldsymbol{r}|^3} - \frac{\boldsymbol{r}_{3rd}}{|\boldsymbol{r}_{3rd}|^3} \right) \tag{2.10}$$

where $\boldsymbol{r}_{3rd}$ is the position of the third body with respect to the centre and μ_{3rd} is its gravitational constant. The initial propagation centre is automatically used as third body if a sample is exiting from its SOI.

2.4. Events detection

CUDAjectory has been optimised to run on GPUs. To maximise the performance, it has been developed to store the final states of the samples only, which is the main output of the software. Avoiding the storage process of the intermediate integration steps of the samples allows to exploit the maximum throughput of the device. Indeed, the additional computational expense of handling the intermediate states implies a huge performance loss. In many applications some event detections during the integration are the most valuable output. In CUDAjectory the event detection algorithms are called *checks*; the checks available in the previous version are five: collision, close-approach, SOI, SOI-crossing and altitude. The user can choose which checks are actively used during the integration in the configuration file. Every check is implemented in a separate algorithm.

2.4.1. Collision check

The Collision algorithm controls whether each sample collides into a celestial body. The user can select which bodies are controlled at each time step by the algorithm; these bodies are called *activated*. Inside the algorithm, each activated body is approximated to be a perfect sphere: the radius of the body is constant, and this allows to easily address the eventual collisions. The user may also specifies an additional offset on top of the radius of the activated body, leading to a larger sphere of collision around it.

The rationale behind the algorithm is simple: at each time step, for each sample, the norm of the position vector and the radius of the collision sphere of the activated body are compared. The role of the gravity is essential in this logic: the samples flying close to the activated bodies have a smaller step size, allowing this type of control logic. The smaller step size is due to a rapid change of the state vector whenever the sample is close to a body. The reason of this is clarified in Section 2.5.

An additional user-defined flag lets the algorithm *refine* the collision. Refine here means that if a collision is detected, the algorithm iterates in order to improve the position of the collision point. The iterations are carried out with a bisection-like method, having the interval defined by the integrational step. The root to be searched is identified thanks to the function given by the difference between the norm of the position vector and the radius of the collision sphere.

The user may also tune how much the collisions are refined, specifying a tolerance. The tolerance constrains the minimum radial variation between two iterations, and so the time step.

2.4.2. Close-approach check

The goal of this algorithm is to compute the pericenter of the orbits with respect to a set of user-defined (activated) bodies; the set of activated bodies might be selected differently from the one of the collision check.

The algorithm works with a bisection scheme as well, with the interval specified by the integrational step. The function used to define the root is the radial velocity of the sample with respect to an activated body. The tolerance represents the same physical constraint as in the collision algorithm, and can be specified independently.

2.4.3. SOI and SOI-crossing checks

The SOI and SOI-crossing algorithms are almost identical; the goal is to detect the crossings over the SOIs and, if any, to update the current integration centre. If an event is detected, the SOI algorithm outputs the current SOI only, while the SOI-crossing both the incoming and current one.

It is important to note, that these algorithms are responsible to change the integration centre during the propagation. As a consequence the user must use one of the two checks in most of the simulations, even if the output is not strictly needed. As it is discussed in Chapter 6, this is a limiting factor for many analyses.

2.4.4. Altitude check

The altitude check in CUDAjectory is not a proper event algorithm. Indeed the goal of the algorithm is to output the altitude with respect to the current integration centre at each time step. Despite not controlling an event, its management inside the software is exactly the same as the other checks. The output can be useful for some analyses but since this algorithm produce data at each time step, it drastically impacts the performances.

2.5. Integration scheme

The equation of motion presented in Eq. (2.6) is a second order Ordinary Differential Equation (ODE), expressed in a vectorial form, forced by the perturbation term $\boldsymbol{a_p}$. The order can be reduced introducing the velocity vector $\boldsymbol{v}$, which is the first order derivative of the position vector $\boldsymbol{r}$:

$$\begin{cases} \dot{\boldsymbol{v}} = -\dfrac{\mu}{\boldsymbol{r}^3} + \boldsymbol{a_p} \\ \dot{\boldsymbol{r}} = \boldsymbol{v} \end{cases} \tag{2.11}$$

The Eq. (2.11) is a system of six ODEs, written in a vectorial form with respect to the state vector introduced in Eq. (2.2). The latter can be numerically integrated starting from the samples initial conditions, provided by the user. The software uses a Runge-Kutta 78 variable time-step integrator, with the Felhberg method to control the step size [15]. The integrator is of seventh order, while the eighth order is used to control the step size. The complete discussion about the implementation of this method can be found in [17]. The user can tune two parameters, called relative and absolute tolerance to tune the step size adjustments; default values are both 10^{-12}.

2.5.1. Step size control

The control on the step size enhances the stability of the integration [15]. The adjustment is regulated by an error parameter, computed with respect to the tolerances, the next state of the samples ($\boldsymbol{x_{n+1}}$), and the state of the sample computed with the eighth order method ($\boldsymbol{x^8_{n+1}}$) [15]:

$$\epsilon = \sqrt{\frac{1}{6}\sum_{i=1}^{6}\left(\frac{(\boldsymbol{x_{n+1}})_i - (\boldsymbol{x^8_{n+1}})_i}{R_{tol}|(\boldsymbol{x^8_{n+1}})_i| + A_{tol}}\right)^2} \tag{2.12}$$

The latter is a norm over each component of the state (denoted by the index i), where R_{tol} is the relative tolerance, A_{tol} is the absolute tolerance and n indicates the time step. The integration step ratio $\frac{h_{n+1}}{h_n}$ is adjusted with an inverse exponential function of ϵ. The higher the error, the smaller the step ratio become and vice-versa. The ratio is bounded between 0.2 and 5.0. This means that the step cannot increase or decrease more than five times the previous value. This boundaries played an important role in the development of this work as the next chapters discuss. It is also important to mention that the integrator can accept also a requested h_{n+1} imposed by the event algorithms. In this case the integrator directly assumes the proposed value, without computing the ϵ function.

Once the next time step is computed, the algorithm controls if its value is between a minimum and a maximum value; the default values of these quantities are $10^{-3}s$ and $10^{7}s$, and the user may adjust it in the configuration file.

2.6. Simulation management

In the previous sections, the main inputs/outputs of the software have been presented, together with the main features and its capabilities. This section gives an higher level overview of how the simulation is managed.

The software can be divided into three sequential parts:

- data preparation and allocations;
- kernel launch and execution
- data handling

2.6.1. Data preparation and allocations

First, the user defined settings are parsed and stored into different objects; the objects are then copied into the device shared memory, to be accessed by each block during the kernel execution.

Once the simulation has been set, the program parses the input states of the samples, and creates an object that contains the data accordingly. Each sample is an object itself, composed by multiple fields, such as the current integration centre, the state, the current epoch and its *status*. The status indicates whether a sample is running, the propagation is finished, or some events happened (e.g. a collision). The object containing all the data is copied in the device global memory, ready to be accessed by the kernel.

A CUBE file is composed by a lot of data, even up to few GB; the software selects the needed sub-set of it and rearranges the data as a 2D array. The complete dissertation on the implementation and storage of the ephemerides can be found in [38]. The resulting data set is stored in the texture memory; this memory has been found to grant the highest efficiency at running time [17, 38].

The *output array* contains the states and the epoch of each sample at every time step. If any check routine is activated by the user, the array also contains the relative output for each sample and time step. The number of steps used to allocate the array is explained in the next subsections. The array is allocated in the device global memory, to be accessed during the kernel execution.

The output array contains many data that are not needed after the simulation is finished. Few *result arrays* are allocated in the host memory, to store the final, parsed, results. One result array is dedicated to store the final states of the samples (the main result of the sofware), and the others are devoted to the event detections, one per check. The Result arrays are 2D NumPy arrays [31], with the size defined by the number of stored variables per detection and a user-defined number of rows. The default values is 1000 rows per sample, and the user might have to adjust it in the case of long simulations.

2.6.2. Kernel launch and execution

Each sample is integrated in a separate device thread. Calling the number of samples to be propagated N, and the number of available threads n, typically $N \gg n$. A single kernel launches all the samples to be integrated, block by block, until all the blocks have been propagated. The same instructions are applied to different samples, and each output is stored in the output array, allocated in the global memory.

The kernel collects the data from the settings object and from the samples objects. The samples are propagated and the next time step is computed using the procedure briefly presented in Section 2.5. At each time step, the kernel runs the requested check algorithms and stores the relative data in the output array. The checks utilise the state just computed to perform the calculations. This procedure is iterated until the propagation of each sample is *terminated*.

2.6.3. Samples clustering and terminal conditions

The above paragraph does not clarify the terminal condition of the samples integration. The kernel could be developed to propagate each sample until the requested final epoch is reached. Instead, the kernel has been implemented to optimise the ephemerides evaluation, which is one of the most computationally expensive task performed. If the threads inside a warp have to read the ephemerides data at epochs close to each other, the read operations are more efficient thanks to the locality of the texture memory addresses, as presented in Section 1.6.1. For this reason, before the kernel launch, the samples are sorted for epoch, in order to exploit this efficiency peak granted by the texture caching [17]. However, even if the initial epochs are similar, the more the simulation goes on and the more the samples epochs may diverge from each other, leading to a performance loss. To overcome this issue the kernel is stopped after a *maximum amount of steps*. Afterwards, the samples are sorted again. The kernel is launched, starting from the intermediate states found with the previous iteration of the kernel. The maximum number of steps is a user-defined parameter with default value of 100. The kernel is sequentially launched multiple times, until each sample reaches its final state.

The terminal condition is meet when a sample reaches the final epoch or if it collides with a body (if the collision check is active). The software keeps track of the samples that are terminal thanks to the status field of the samples objects, as said in Section 2.6.1. The samples that are not terminal are called *active*.

2.6.4. Data handling

Each run of the kernel produces and stores the partial output inside the global memory. The size of the output array is computed such that the states, the epochs, and the checks output can be stored at any time step. So, the size depends on the number of active samples, the maximum amount of steps, the size of the output produced by the user-selected checks, and the state-epoch vector size.

After the execution of a kernel, the output array contains all the output produced. If the sequential kernel starts immediately afterwards, it overwrites the data inside the output array. For this reason, the software has to duplicate the output array inside the global memory, such that the next kernel can be launched without losing the output. During the next kernel execution, the CPU parses the output, to search for any event detections. In order to process the data with the CPU, the duplicate of the array has to be copied again from the global memory to the host memory. Once this operation is completed, the kernel can restart. The parsing operation performed by the CPU consists in checking if the default values of the output array duplicate have been overwritten. If any value differs from the default one, it corresponds to an event detection. The parsed event detections are stored into the final result arrays.

When the simulation is over the program frees all the allocations, on both host and device.

3 | Close-approach event

This chapter presents the improvements made on the close-approach algorithm. As discussed in Section 2.4, the close-approach was already developed before this work. Since an optimisation margin was present, this part of the work focuses on the development process to improve the running time of the procedure.

The aim of this check is to detect the close-approaches between each sample and a set of the user-defined celestial bodies, called activated bodies. The user may select both a primary body and a secondary body (e.g. the Earth and the Moon) and the algorithm shall return the *closest* close-approaches with respect to each body. The following example clarifies this last sentence. Consider a sample that flies in an elliptical closed-orbit around the Earth with an apogee below 150000 km. Since the Earth-Moon distance is approximately 484000 km, the sample is closer to the Earth than to the Moon at any time. In order to propagate this sample in CUDAjectory, the user has to specify the initial state vector, to activate the close-approach in the configuration file, and to set the Earth and the Moon as active bodies. The sample flies past its perigee (i.e. the close-approach with respect to the Earth) once per orbit. Moreover, once per orbit, the sample is situated in its close-approach with respect to the Moon. However, since the Earth is closer than the Moon at every time step, the close approach with respect to the Earth is always the closest one.

3.1. Previous implementation

The framework of the algorithm is depicted in Fig. 3.1, where A_i are the activated bodies with $i = 1, 2, ..., N, \quad C$ is the current integration centre, and S is a sample.

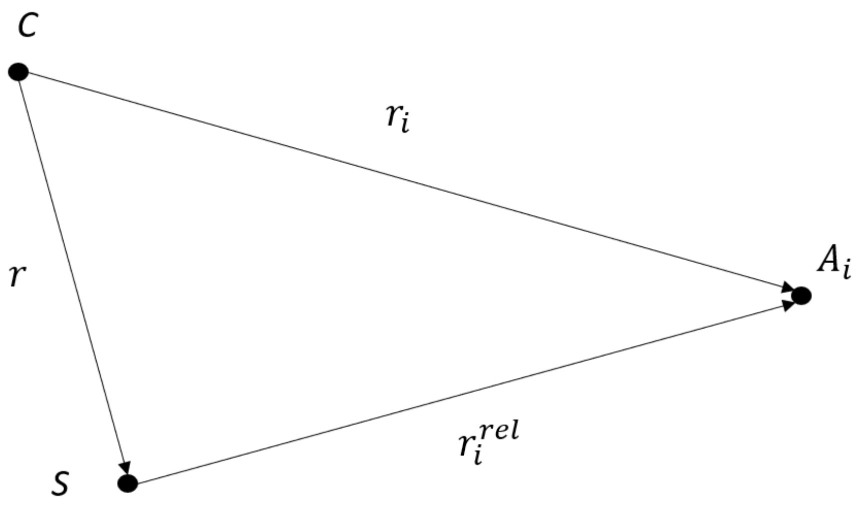


Figure 3.1: Framework of the close-approach algorithm.

$\boldsymbol{r}_i$ and $\boldsymbol{r}$ indicates respectively the i-th activated body and the sample position vectors with respect to the integration centre. The relative position vector is denoted as $\boldsymbol{r}_i^{rel}$. Using the same notation $\boldsymbol{v}_i$, $\boldsymbol{v}$ and $\boldsymbol{v}_i^{rel}$ are respectively the A_i, the sample, and the relative velocity vectors. Thus, the relative velocity and position vector can be computed as:

$$\boldsymbol{r}_i^{rel} = \boldsymbol{r}_i - \boldsymbol{r}$$
$$\boldsymbol{v}_i^{rel} = \boldsymbol{v}_i - \boldsymbol{v}$$

3.1.1. Closest celestial body retrieval

The routine starts identifying the closest activated body at the current time step. This part of the algorithm is here shown as pseudo-algorithm in Algorithm 3.1. The algorithm compares the distance of the sample with respect to all the activated bodies to find the closest one. The pseudo-function `centreIsActive()` at line 3 is used to control if the current integration centre is an active body. Inside the for loop the distance of each activated body is computed. At the end of the cycle, the closest body is saved in the k variable.

It can be noted that the procedures to compute the position and velocity vectors are not included. Indeed those are estimated thanks to the ephemerides functions, which are not the focus of this work.

Algorithm 3.1 Closest body

```
1: $dist = 10^{30}$                      % initialization
2:
3: if $centreIsActive()$ then
4:     $dist = \boldsymbol{r}.norm()$
5:     $k = centre$                      % storing the closest body
6: end if
7:
8: for $i = 1 : N$ do
9:     if $\boldsymbol{r}_i^{rel}.norm() < dist$ then
10:        $dist = \boldsymbol{r}_i^{rel}.norm()$
11:        $k = i$                       % storing the closest body
12:    end if
13: end for
```

3.1.2. Close-approach event condition

The close-approach algorithm runs at each time step. The routine has to identify whenever the sample is at the close-approach condition with respect to the closest body. To do so, the radial component of the relative velocity, called *range rate* is used; the same method is also used in Suite for the Numerical Analysis of Planetary Protection (SNAPPshot) [22]. The range rate (rr) is computed as:

$$rr = \frac{\boldsymbol{r}_k^{rel} \cdot \boldsymbol{v}_k^{rel}}{dist} \tag{3.1}$$

When the range rate is equal to zero, the sample is situated in its nearest point of the orbit with respect to the k-th body.

The range rate is computed at each time step. Since it is inside a numerical integration, the exact condition in which $rr = 0$ is never reached. However, the time interval in which the close-approach is located can be identified thanks to the sign of the range rate. The Fig. 3.2 shows the timeline during the integration of a sample, together with its sign of the range rate function. $t_1, t_2..t_{100}$ are the time steps during the integration.

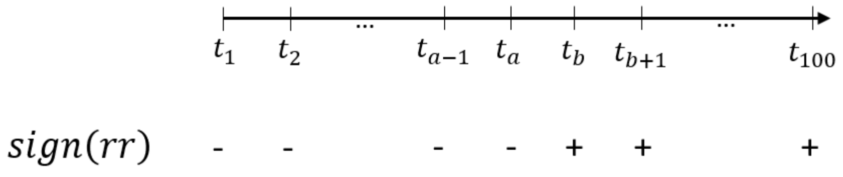


Figure 3.2: Timeline of the close-approach event during the integration.

Initially, the sign of the range rate is negative, because the sample is approaching the body. At t_b, the sign of the range rate becomes positive. As a consequence, the event is located between t_a and t_b. In order to find the interval t_a-t_b, the algorithm stores the range rate at each time step. The current range rate is here denoted as rr, while the previous range rate as rr_{prev}.

3.1.3. Sample status

Once the interval that locates the event is found, the algorithm performs a bisection-like method in order to refine the close-approach, iterating on the time step. Proposing the next time step, the algorithm rules the integration in order to converge at the event. As presented in Chapter 2, each sample is an object and its status is a field of the relative class instance. The algorithm uses this value to keep track of the bisection status. This algorithm uses three status values: *running*, *before* and *after*. The status is equal to running when the sample is before t_a, while statuses before and after are assigned during the bisection. These two statuses refer to the conditions in which the sample is before and after the epoch of close-approach event.

3.1.4. Bisection scheme

In a bisection scheme, the interval is halved at every iteration, and the algorithm stops when the tolerance is meet. The goal of this particular bisection scheme is to converge to the close-approach condition exploiting the possibility to propose the next time step, as discussed in Section 2.5. The Algorithm 3.2 shows the bisection scheme as a pseudo-algorithm.

The line 18 is the control that stops the loop. The bisection stops once the product between the range rate and the time step is less than the specified tolerance, expressed in kilometers. Indeed, this tolerance represents an approximation of the radial distance variation between the sample and the k-th body. Its default value is 50km.

An example is here provided to help in understanding the algorithm. A sample is propagated with a single activated body (i.e. it is the k-th body at each time step). During the integration, the algorithm finds the interval (t_a, t_b) in which the event is contained, thanks to the control at line 3 of the Algorithm 3.2.

Algorithm 3.2 Bisection scheme

```
 1: if $sample.status == running$ then
 2:
 3:     if $rr > 0$ && $rr_{prev} < 0$ then
 4:         $sample.status = after$
 5:         $h = -0.5 \cdot h$
 6:     end if
 7:
 8: else if $sample.status == after$ then
 9:
10:     if $rr < 0$ then
11:         $sample.status = before$
12:         $h = -1 \cdot h$ % revert the time step to forward integrate
13:     end if
14:
15: else if $sample.status == before$ then
16:
17:     if $rr > 0$ then
18:         if $|rr \cdot h| < tolerance$ then
19:             $sample.status = running$
20:             save detection
21:         else
22:             $sample.status = after$
23:             $h = -0.5 \cdot h$
24:         end if
25:     end if
26:
27: end if
28:
29: $rr_{prev} = rr$
```

The timeline in Fig. 3.3 shows the behaviour of the range rate function in time of the example. In the figure, the interval (t_a, t_b) is mapped into a unitary length interval. As a consequence, the origin represents the condition at t_a and the value $t = 1$ is associated with t_b.

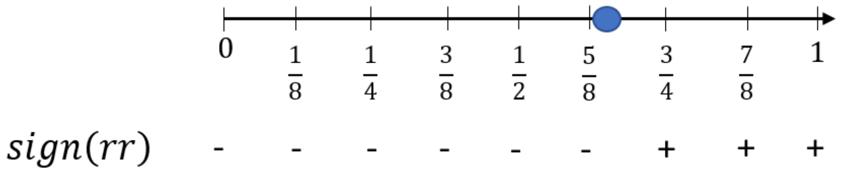


Figure 3.3: Timeline of the example.

Looking at Fig. 3.3 it is possible to note that the close-approach of the sample (indicated with the blue marker), is approached between point $\frac{5}{8}$ and $\frac{3}{4}$, closer to $\frac{5}{8}$. The following list shows the iteration that the algorithm performs:

1. the bisection starts at $t = t_b$ and $h = 1$. The condition at line 1 and 3 are verified, so, the status is changed to after and the time step is computed as $h = -0.5 \cdot 1$. Consequently, the sample is propagated, arriving to $t = 0.5$;
2. at $t = 0.5$ the range rate is negative. Since the status is equal to after, the conditions at line 8 and line 10 are verified. The status is changed to before and the time step is adjusted with $h = -1 \cdot (-0.5) = 0.5$. The sample is propagated back to point 1;
3. the triggered controls are at line 15 and 17. So, the status is updated once again to after and $h = -0.5 \cdot 0.5 = -\frac{1}{4}$. The sample is integrated to point $\frac{3}{4}$.
4. line 8 is verified, but line 10 is not. The status and the time step are not modified and so the sample is propagated to point $\frac{1}{2}$.
5. the range rate switches the sign and lines 10 becomes true. The interval is reversed at line 12 as $h = -1 \cdot (-\frac{1}{4}) = \frac{1}{4}$ and the status is saved to before. The sample goes to point $\frac{3}{4}$.
6. the modus operandi should be clear now; if the tolerance is not meet yet, the sample goes to point $\frac{5}{8}$
7. at point $\frac{5}{8}$ the interval is reversed and the sample come back again at $\frac{3}{4}$;
8. etc.. until the tolerance condition is meet.

3.1.5. Optimisation margin

Thanks to the previous example, it is clear that during multiple iterations (2, 5 and 7) the algorithm does not half the time interval. The algorithm converges to the event condition, but with a non-constant convergence speed, lower with respect to a classic bisection. The bisection methods are stable and robust algorithms, but slower than other root-finding methods (e.g. the Newton-Rapson method). In this case it is even slower than a classic bisection. The goal of this part of the work is to implement a new bisection-like scheme (in order to exploit the high robustness of the method) that avoids any zero-convergence velocity iteration.

The explanation of why the previous algorithm must revert the interval at line 12 is discussed in Section 3.2.3.

3.2. New implementation

As discussed in the previous section, the goal of the new implementation is to force the algorithm on halving the interval at every time step, improving the rate of convergence of the check, and so the computational time. The first part of the algorithm (the one used to retrieve the closest body) has been kept, whereas the bisection scheme has been modified.

3.2.1. Sample status

In order to reach the discussed aim, the first change introduced is related to the sample status. Indeed, the time interval has to be halved at each time step independently from the sample being after or before the event epoch. A new status called *CloseAppr* is introduced, denoting that the sample is performing the bisection.

3.2.2. Bisection scheme

The first iteration starts as in the previous implementation: if the sample status is equal to running and the a change in the range rate is detected, the status is modified to start the bisection (to CloseAppr this time).

Starting from the second iteration, the interval must be halved at every iteration, so the absolute value of the time step can be computed as:

$$|h| = 0.5 \cdot |h| \tag{3.2}$$

In order to converge at the event condition, the sign of the time step can be computed accordingly to the sign of the range rate:

- if the range rate is positive, h has to be negative in order to backward integrate;
- if the range rate is negative, h has to be positive in order to forward integrate.

The two conditions stated above, together with Eq. (3.2), can be computed as:

$$h = -sign(rr) \cdot 0.5 \cdot |h| \tag{3.3}$$

Thanks to logical arithmetic the sign function of the range rate can be expressed as:

$$sign(rr) = (rr > 0) - (rr < 0) \tag{3.4}$$

3.2.3. Infinite looping

The previous bisection scheme has been developed in order to enforce the sample to be after the event epoch at the last iteration (i.e. sample status equal to after). The rationale behind this choice was to prevent an infinite loop. Indeed, if the sample is before the event epoch, the range rate is negative, but at the next integration step it becomes positive. This behaviour triggers the algorithm to start from scratch in order to detect again the same event. The consequence is an infinite loop.

The new bisection scheme, by exploiting Eq. (3.3) and Eq. (3.4), remains prone to the same issue. In order to prevent the infinite loop, once the bisection is complete, the time step is computed as:

$$h = 2 \cdot |h| \tag{3.5}$$

Moreover, the range rate is imposed to be positive ($rr = 1$). This condition, together with Eq. (3.5), ensures that the sample is after the detection point epoch and that the range rate does not trigger the algorithm again.

The Algorithm 3.3 shows the new bisection scheme as a pseudo-algorithm.

Algorithm 3.3 New bisection scheme

```
1: if $sample.status == running$ then
2:
3:     if $rr > 0$ && $rr_{prev} < 0$ then
4:         $sample.status = CloseAppr$
5:         $h = -0.5 \cdot h$
6:     end if
7:
8: else if $sample.status == CloseAppr$ then
9:
10:    if $|rr \cdot h| < tolerance$ then
11:        $sample.status = running$
12:        $h = 2 \cdot |h|$
13:        $rr = 1$
14:        save detection
15:    else
16:        $h = -0.5 \cdot ((rr > 0) - (rr < 0)) \cdot |h|$
17:    end if
18:
19: end if
20:
21: $rr_{prev} = rr$
```

3.3. Validation

The new developed algorithm has been validated using a dummy test case. The same samples have been propagated in CUDAjectory and in Godot. Godot is a software package for orbit propagation and optimisation available under ESA Community Licence [12]. The close-approach event detections obtained with CUDAjectory, have been compared to those obtained with Godot, in order to prove the validity of the algorithm.

3.3.1. Setup

The same model has been set in both software:

- two body perturbed dynamics;
- J_2 perturbation;

- Moon third-body perturbation;
- no SRP;
- propagation period, 1 day.

A set of 240 Low Earth Orbit (LEO) samples has been randomly generated with a uniform distribution. The range of the orbital parameters used to define the initial states of the samples are the following:

- **perigee**, $7000 < r_p < 8000$
- **eccentricity**, $0.5 < e < 0.9$
- **inclination**, $0° < i < 90°$
- **Right Ascension of the Ascending Node (RAAN)**, $0° < \Omega < 360°$
- **Argument of Pericentre (AoP)**, $\omega = 0°$
- **True Anomaly (TA)**, $0° < \theta < 360°$

The perigee has been used instead of the semi-major axes to constrain the samples in the LEO region. The semi-major axes (a) can be retrieved using the eccentricity as: $a = \frac{r_p}{1-e}$

3.3.2. Results

The results of the comparison between the events detected by Godot and CUDAjectory are shown in Fig. 3.4.

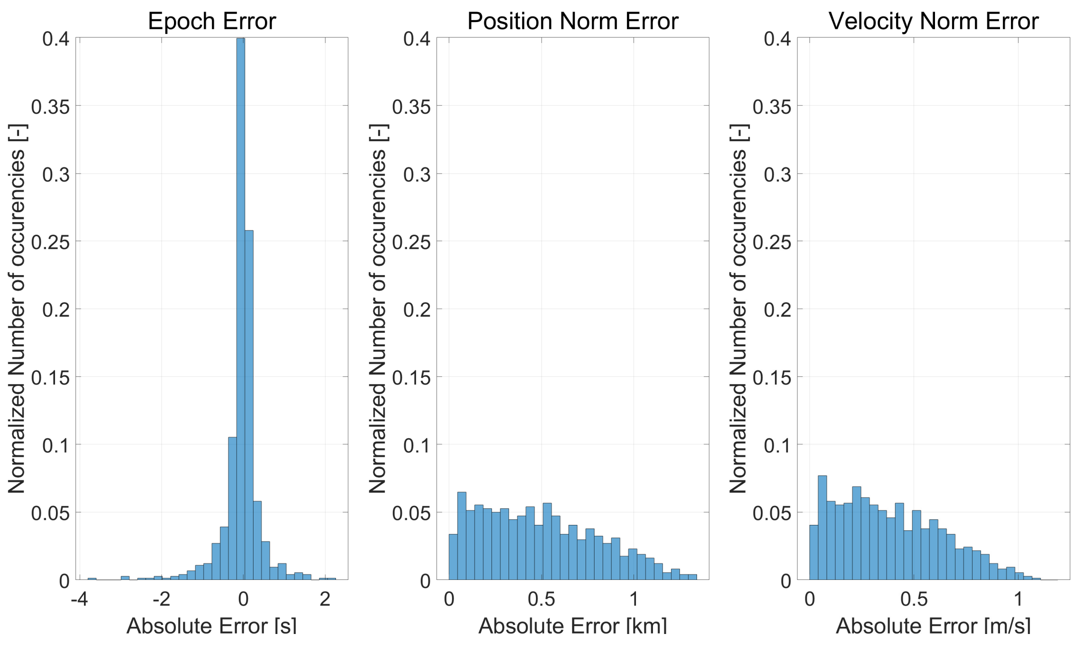


Figure 3.4: Results of the close-approach comparison.

This comparison has been taken supposing that Godot is the reference solution. For this reason, the difference between the two output is denoted as *error*. The three histograms show the error in terms of epoch, position norm and velocity norm respectively, computed at each detection. The results proves the reliability of the algorithm, keeping the epoch error bounded between -4 and 2 seconds, the position norm error between 0 and 1.5 kilometers and the velocity norm error between 0 and 1.5 meters per second.

3.4. Computational time comparison and conclusions

The computational time of the two algorithms (previous and new version) are compared in this section. Multiple tests have been ran, varying the tolerance, the number of samples, the integration period and the orbits to be propagated. All the simulations have been carried with just the close-approach check active. The results of this analyses show a decrease on the total computational time between 3% and 15%, depending on the simulation. The stricter the tolerance is, the higher the uplift in performance is.

This implementation improves the computational time in few aspects. The increased convergence velocity is directly impacting the performance. The example presented in Section 3.1.4 shows that the previous bisection scheme did not halved the time interval three times over seven iterations. Moreover, the reduction of the total number of computational steps has also the benefit of reducing the data handling time. Indeed, as discussed in Chapter 2, after every kernel launch the handler has to scroll the output array at each time step for all the samples. Thanks to the less time steps to be controlled, the new implementation lowers the time needed to perform the parsing operation. Moreover, the number of if-else branches in the new bisection scheme is lower than before, as can be seen in Algorithm 3.3, leading to less *warp divergence*.

4 | Earth protected regions events

The current space debris problem is getting worse year by year [23]. Space companies have to follow stricter mitigation guidelines with respect to the past. Additional preliminary studies and assessments are performed, such as statistical interference analyses with the near-Earth regions, crowded by space objects. CUDAjectory allows its users to propagate millions of samples in parallel, granting faster computational time than CPU software on large scale analyses. Consequently, the software is a candidate to perform the needed assessments, leading to the need of implementing algorithms to detect whether a sample is inside the critical regions. This chapter discusses the development of two algorithms: the LEO and Geostationary Earth Orbit (GEO) Earth protected region checks.

4.1. Protected region convention

Two near-Earth regions are particularly crowded of space debris; the first one is the LEO region, while the other one is situated across the Geostationary orbit, the GEO region. The Inter-Agency space Debris coordination Committee (IADC) is an international committee of space agencies, aimed at dealing with the current situation for the near-space debris. The committee published a document containing a set of guidelines in order to mitigate and prevent a steep increase of space debris. In the document, the following conventions are proposed [1]:

- LEO protected region - spherical region that extends on top of the Earth's surface up to an altitude of 2000 km;
- GEO protected region - a segment of spherical shell defined by an altitude between plus and minus 200 km from the geostationary altitude, and a latitude (l) between -15 a and 15 degrees. The altitude of the Geostationary orbit is 35786 km.

The two regions are shown in Fig. 4.1; the view is centred in the Earth equatorial plane and the altitude is called Z.

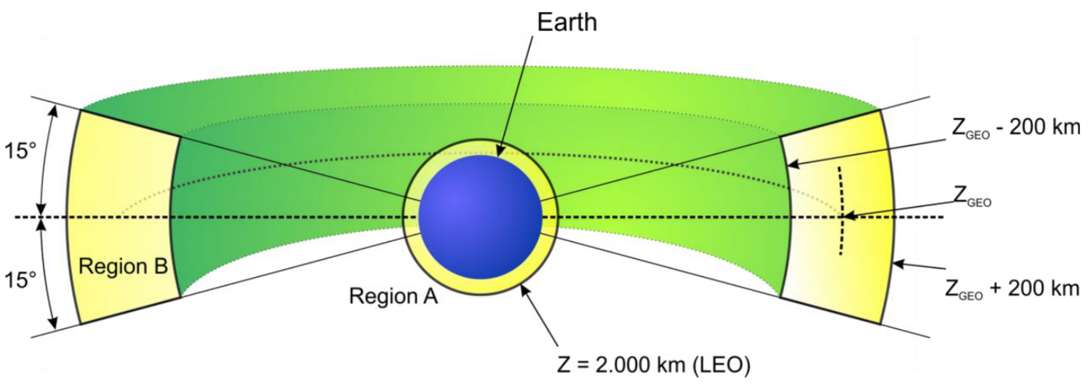


Figure 4.1: Earth protected regions - IADC conventions.

4.2. Requirements

The main requirement of the two algorithms is to detect both the entry and exit conditions from the protected regions. Having both conditions allows to estimate the time of flight spent inside each region, useful to perform statistical and interference analyses.

Since the two events cannot happen at the same time step (a sample cannot be simultaneously situated in both regions), two separate algorithms have been developed.

4.3. Bisection scheme limitations

As presented in Chapter 3, the bisection method is a very robust root-finding algorithm and can be exploited on GPUs as well. Once the interval that contains the root is found, the algorithm implementation is straight-forward.

The LEO region algorithm has to control if a samples crosses the space-boundary defined by the IADC conventions. Instead, the GEO region *check*, has to verify the crossings over four boundaries, once again, defined by the IADC conventions. These physical boundaries can be used to define the functions that characterize the events. For example, considering the LEO region, the event is identified by the root of the function:

$$f(t) = z(t) - z_{LEO} \tag{4.1}$$

where z is the altitude of the sample with respect to the Earth. Aiming to adopt a bisection-method, the algorithm shall robustly find the time interval which locates the zero of the function f.

Consider the integration of a given sample flying towards the LEO protected region. In order to find the time interval that contains the zero of f, in at least one step the sample has to be located inside the LEO protected region. Indeed the sign of the function f is negative inside the region and positive outside it, and if a change of sign of f is detected, the time interval is immediately defined. However, a given sample orbit might not cross the LEO boundary enough to have the sample (in a time step) located inside the region, and so to detect a change of sign of f. As a consequence, the robustness of a bisection method based on the function f it is dependent on the type of orbit itself. For analogy, the same concepts applies to the GEO protected region.

4.4. LEO protected region

The main issue involved in the use of a bisection scheme, as discussed above, is regard the difficulties of consistently having a time step inside the LEO region for each crossing orbit. Moreover, the number of branches has to be kept as low as possible, in order to minimise the warp divergence. Instead of relying on a bisection scheme, at each time step it is possible to predict the next altitude. Accordingly to the prediction, the time step can be adjusted to converge at the LEO boundary. The process of correcting the time step according to a prediction is here called *adaptive step refinement.*

4.4.1. Adaptive step refinement

The altitude at the next time step can be predicted as:

$$z_{n+1} \approx z_n + v_r h_{n+1} \tag{4.2}$$

where n indicates the current step. This prediction is based on a linear extrapolation with respect to the radial velocity v_r, which expresses the variation of the altitude in time. The smaller the next time step is and the more correct the prediction is. Assuming that a sample is orbiting around the Earth, its radial velocity can be easily computed by projecting the velocity vector on to the position vector of the sample:

$$v_r = \frac{\boldsymbol{r} \cdot \boldsymbol{v}}{||\boldsymbol{r}||} \tag{4.3}$$

The next time interval (h_{n+1}) is not known, and so the Eq. (4.2) cannot be directly used to predict the next altitude. However, as presented in Section 2.5, h_{n+1} is bounded between

0.2 and 5.0 times h_n. In the case under analysis, the 5.0 factor is the worst case value. Indeed, the higher the time interval, the higher the distance the sample travels and the higher the possibility to cross the LEO region boundary. According to this reasoning, it is possible to predict the worst-case next altitude as follows:

$$z^*_{n+1} = z_n + v_r 5 h_n \tag{4.4}$$

where z^*_{n+1} is the predicted worst-case altitude. It can be noted that using the instantaneous radial velocity to extrapolate the altitude at the next time step it is a linear approximation, and so an additional safety factor could be added in Eq. (4.4). However, since the check is related to near-Earth samples it is unlikely that the step is augmented by the whole 5.0 factor, and it has being avoided to add an additional penalty to the prediction. Indeed, the prediction is used to adjust the next time step accordingly, and so an additional penalty is unnecessarily slowing down the propagation.

4.4.2. Check implementation

The Eq. (4.4) can be used to control if z^*_{n+1} is below 2000 km. consequently, the sample is predicted to be inside the LEO protected region at the next time step. In order to find the entry point inside the region, the algorithm has to converge to the LEO boundary. Aiming to do so, the next time interval is imposed such that the sample is predicted to be located exactly at the boundary. To achieve it, z^*_{n+1} has to be equal to z_{LEO}, and so, using Eq. (4.4), the following relationship can be found:

$$h_{n+1} = \frac{z_{LEO} - z_n}{5 v_r} \tag{4.5}$$

The Eq. (4.5) is used if the sample is predicted to cross the LEO boundary. Since the prediction is computed in the worst case, the Eq. (4.5) implies that the sample is always before the boundary at t_{n+1}. The step adjustment is iterated until the time interval is such that a user-defined tolerance (tol) is meet. The tolerance constraints the step adjustment according to the following:

$$|v_r| * h_n >= tol \tag{4.6}$$

While Eq. (4.6) is verified, the step is adjusted as presented. Instead, if it is not verified,

h_{n+1} is computed imposing the tolerance:

$$h_{n+1} = \frac{tol}{|v_r|} \tag{4.7}$$

Once the sample crossed the boundary, the algorithm performs the same steps as above, but converging to the exit conditions. The Eq. (4.4) is used to predict the altitude; if it is above z_{LEO}, the time interval is computed with Eq. (4.5), Eq. (4.6) and Eq. (4.7).

The complete pseudo-algorithm is shown in Algorithm 4.1.

Algorithm 4.1 LEO protected region algorithm

```
1: if $((sampleInLeo() \ \&\& \ z >= z_{leo}) \ || \ (!sampleInLeo() \ \&\& \ z < z_{leo}))$ then
2:     save detection
3: else
4:     $z^*_{n+1} = z_h + v_r 5 h_n$
5:     if $((sampleInLeo() \ \&\& \ z^*_{n+1} >= z_{leo}) \ || \ (!sampleInLeo() \ \&\& \ z^*_{n+1} < z_{leo}))$
       then
6:         $h_{n+1} = (z_{LEO} - z_n)/(5v_r)$
7:         $h_{n+1} = h_{n+1} * (h_n|v_r| >= tol) + \frac{tol}{|v_r|} * (h_n|v_r| < tol)$
8:     end if
9: end if
```

The pseudo-function `sampleInLeo()` is needed to control if the sample was inside or outside the LEO protected region at the previous time step. Line 7 forces the next time interval to be compliant with Eq. (4.5) or Eq. (4.7) depending on Eq. (4.6).

4.5. GEO protected region

The LEO algorithm relies on Eq. (4.1) to detect the crossings over the boundary. The GEO protected region is defined by four boundaries and the four relative functions can be expressed as follows:

$$\begin{aligned}
f_1 &= z - z^+_{GEO} & l^-_{GEO} < l < l^+_{GEO} \\
f_2 &= z - z^-_{GEO} & l^-_{GEO} < l < l^+_{GEO} \\
f_3 &= l - l^+_{GEO} & z^-_{GEO} < z < z^+_{GEO} \\
f_4 &= l - l^-_{GEO} & z^-_{GEO} < z < z^+_{GEO}
\end{aligned}$$

where z^+_{GEO} and z^-_{GEO} are respectively the altitude of the geostationary orbit plus and minus 200 km, and l^+_{GEO}, l^-_{GEO} are respectively 15° and −15°. So, two prediction are now required in order to predict if the sample crosses one of the four boundaries:

$$\begin{aligned} z^*_{n+1} &= z_n + v_r 5h_n \\ l^*_{n+1} &= l_n + \dot{l}_n 5h_n \end{aligned} \tag{4.8}$$

using the 5.0 factor discussed above. The latitude at the current step (l_n) can be computed using the position vector of the sample as:

$$l_n = atan\left(\frac{z_n}{\sqrt{x_n^2 + y_n^2}}\right) \tag{4.9}$$

From Eq. (4.9) the derivative of the latitude ($\dot{l}_n$) can be found as:

$$\dot{l}_n = \frac{v_{z_n}\sqrt{x_n^2 + y_n^2} - \frac{z_n}{2}\frac{2x_n v_{x_n} + 2y v_{y_n}}{\sqrt{x_n^2+y_n^2}}}{1 + \frac{z_n^2}{x_n^2+y_n^2}} \tag{4.10}$$

The derivative of the latitude can also be estimated using a backward finite-difference:

$$\dot{l}_n \approx \frac{l_n - l_{n-1}}{h_n} \tag{4.11}$$

Since the prediction is a rough linear estimation, it is sufficient to use Eq. (4.11) to approximate the derivative of the latitude.

4.5.1. Check implementation

The logic of the adaptive step refinement has been developed in the same way of the LEO protected region algorithm, applied to all four boundaries. Using Eq. (4.8), if the sample is predicted to cross one of the four boundaries, the next time interval is imposed such that z^*_{n+1} or l^*_{n+1} are respectively equal to the condition of the crossed boundary.

4.6. Validation

The two developed algorithms have been validated using two dummy test cases (one per algorithm). The validation process used is the same as for the close-approach algorithm, presented in Section 3.3; a set of generated samples (per algorithm) has been propagated both with CUDAjectory and Godot.

4.6.1. Setup

The setup is the same as discussed in Section 3.3. The set of 240 samples to test the LEO algorithm has been generated with a random uniform distribution accordingly to the following ranges of orbital parameters:

- **perigee**, $7000 < r_p < 8000$
- **eccentricity**, $0.5 < e < 0.9$
- **inclination**, $0° < i < 90°$
- **RAAN**, $0° < \Omega < 360°$
- **AoP**, $\omega = 0°$
- **TA**, $0° < \theta < 360°$

Instead, the orbital parameters used for the second algorithm are defined by:

- **apogee**, $z_{GEO}^{-} < r_a < z_{GEO}^{+} + 500$;
- **eccentricity**, $0.5 < e <= 1$
- **inclination**, $-15° < i < 15°$
- **RAAN**, $0° < \Omega < 360°$
- **AoP**, $\omega = 0°$
- **TA**, $0° < \theta < 360°$

The apogee is here used in order constrain it to be inside or above GEO region.

For both tests, the samples have been propagated for 1 day.

4.6.2. Results

As in the previous chapter, the comparisons have been taken supposing that Godot is the reference solution, and so, the difference between the detections is denoted as error.

The Fig. 4.2 and Fig. 4.3 show the results of the two comparisons. The same figures of merit as in Section 3.3 have been used; so, the error with respect to the epoch, the position norm and the velocity norm vector, computed at each detection are plotted.

The results for the LEO protected region are better the close-approach ones, granting the validity of the procedure.

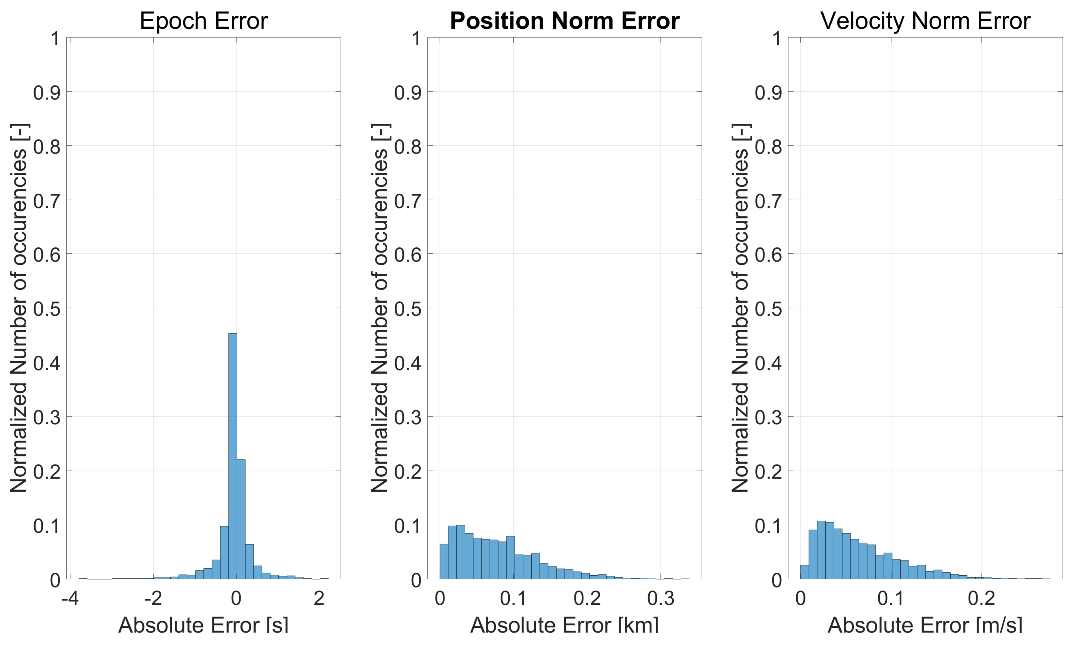


Figure 4.2: Results of the LEO protected region validation.

The Fig. 4.3 shows the comparison with respect to the GEO protected region. It can be noted that the three errors are significantly higher with respect to the LEO results. Since the two software used are different, the difference in terms of propagation may play an important role here, due to the integration of orbits with higher semi-major axes than in the LEO case. The results show that the maximum difference with respect to the position norm vector is below 50km, which is one order of magnitude less with respect to the radial dimension of the GEO protected region. The results are refereed to more than 1500 detections (half entry and half exit conditions). The two software found the same set of interfering samples. The algorithm has been implemented to be exploited in preliminary statistical assessments, and the most important result is to detect all the crossings over the boundaries. Moreover, the magnitude of the position vector at the detection is in the order of 40000 km, so the absolute position error is about 1000 times smaller than it. For these reasons, the validation process of the algorithm is considered success-full. The event detections are precise enough to be used for preliminary analyses.

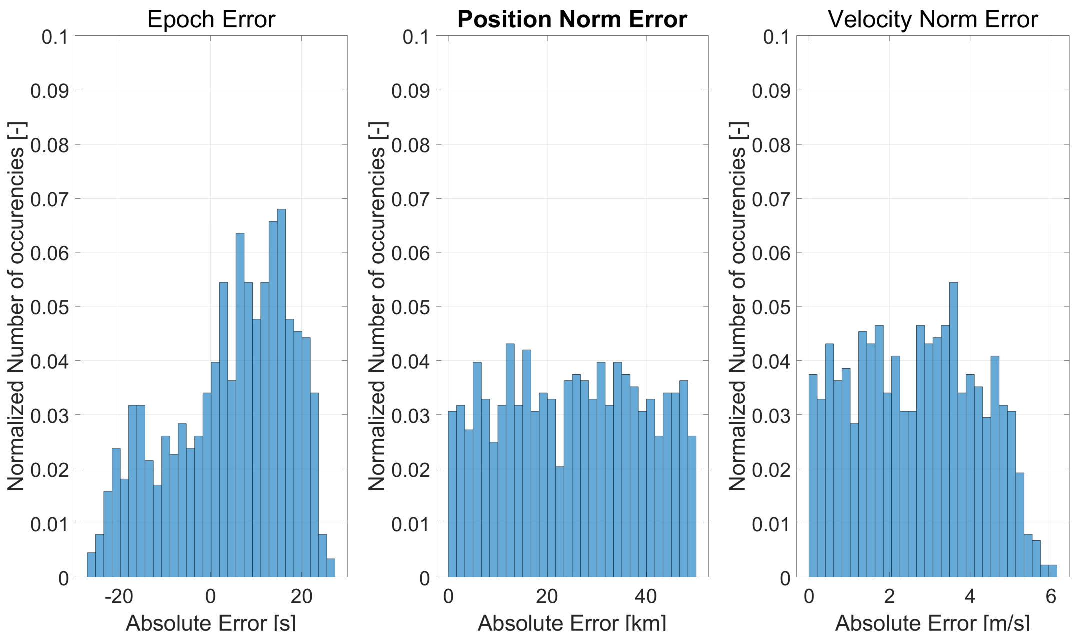


Figure 4.3: Results of the GEO protected region validation.

5 | Massless bodies collision event

This chapter presents the development of the massless bodies collision algorithm. The aim of the algorithm is to detect the collisions between the samples and a set of user-selected *massless* bodies. The term massless here means that the bodies are modelled without a mass. As a consequence, the gravity acceleration is not perturbed by the selected bodies, and the integrational step-size is not intrinsically refined by the integrator during the interference. For each selected body, the user must also specify the radius of the collision sphere around it.

The increased interest in exploiting the Libration points for scientific missions, mainly the Sun-Earth's ones, leads to many interesting applications of the algorithm in discussion, for example interference analyses. For this reason, both entry and exit conditions inside the collision sphere has to be detected, like in the protected regions algorithms.

5.1. Current limitations

The massless bodies are not propagated by the software. The goal is to utilise a user-prompted ephemerides file to evaluate the position and velocity of the requested massless bodies. As discussed in Section 2.1.3, CUDAjectory is currently compatible with Satellite and Planet Kernel (SPK) types 2 and 3 only. Usually, spacecrafts and Libration points ephemerides are not given in these formats. The current compatibility of CUDAjectory with respect to SPK types limits the possibility to provide data for non-planetary bodies. For this reason, the motion of the Sun-Earth Libration (SEL) points has been analytically implemented, in order to be utilised for the massless bodies collision algorithm. The appendix A gives a brief explanation of the format and the usage of these Spice-kernel files.

The goal of the massless collision algorithm implementation is to develop it to be properly expanded to all the sets of Libration points and to user-defined bodies in the feature.

5.2. Libration points

This section, explains the proposed analytical implementation of the Libration points. The ephemerides toolkit of CUDAjectory allows to compute the position and the velocity of the celestial bodies with respect the solar system baricenter at each time step. For this reason, instead of using the classical Circular Restricted three Body Problem (CR3BP) definition (explained in [5]), it is easier to implement the Libration points using the instantaneous distance between the two attractors. The reference frame used for the description is the so called *roto-pulsating frame*, shown in Fig. 5.1, centred in the primary mass (m_1).

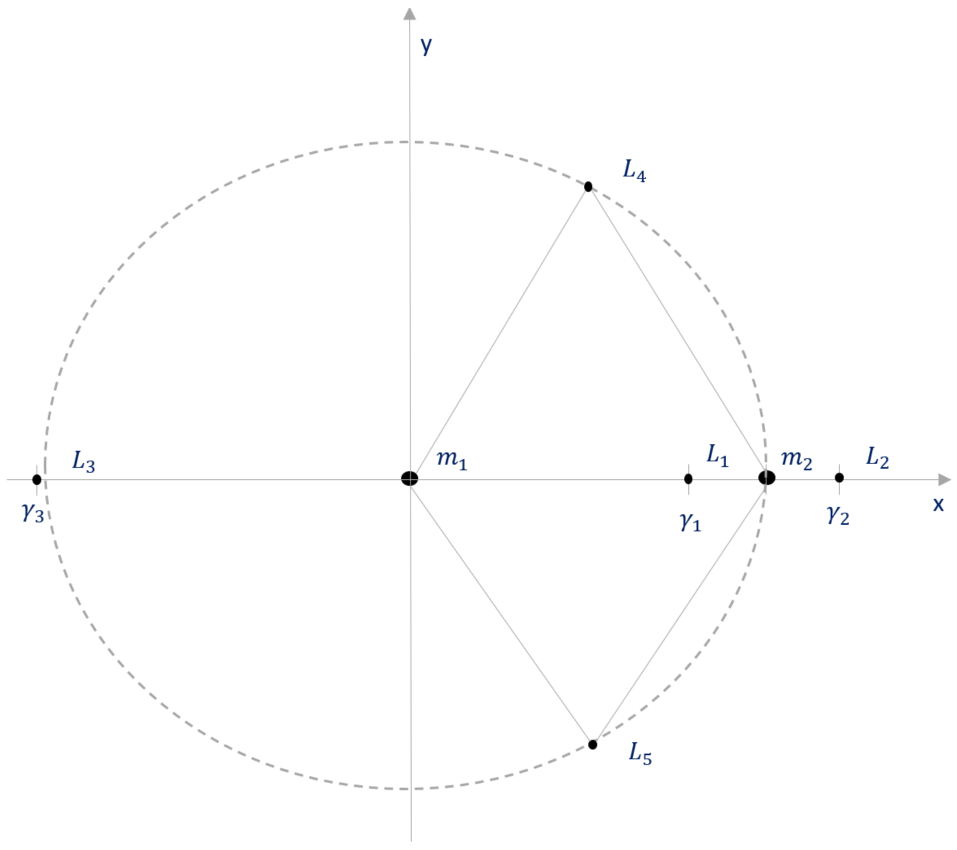


Figure 5.1: Roto-pulsating reference system.

As in the CR3BP, the x coordinate is pointing towards the secondary mass m_2. Therefore this reference frame is rotating with respect to the ICRF centred in m_1, with the same angular velocity (ω) of m_2 around m_1. The axes are normalized with respect to the instantaneous distance between m_1 and m_2, i.e. m_2 is located at coordinates (1, 0).

L1, L2 and L3 points are called collinear points, while L4 and L5 are called triangular points.

5.2.1. Collinear points

The collinear points lie on the x axes, and the distance with respect to the primary body is defined thanks to the γ_i magnitudes as:

$$\boldsymbol{r}_i = \gamma_i \boldsymbol{r}_2 \delta_i \tag{5.1}$$

where $\boldsymbol{r}_2$ is the position vector of the secondary mass with respect to the primary and δ_i is an operator equal to $+1$ at $i = 1, 2$ and -1 at $i = 3$.

The instantaneous velocity vectors are computed supposing to be always tangential:

$$\boldsymbol{v}_i = \omega ||\boldsymbol{r}_i|| \delta_i \hat{\boldsymbol{y}} \tag{5.2}$$

and the versor $\hat{\boldsymbol{y}}$, since the position and the velocity vectors of m_2 lie in the roto-pulsating xy plane, can be computed as:

$$\hat{\boldsymbol{y}} = \frac{\hat{\boldsymbol{z}} \times \boldsymbol{r}_2}{||\hat{\boldsymbol{z}} \times \boldsymbol{r}_2||} \tag{5.3}$$

where $\hat{\boldsymbol{z}}$ is the versor of the z-axis, computed as:

$$\hat{\boldsymbol{z}} = \frac{(\boldsymbol{r}_2 \times \boldsymbol{v}_E)}{||(\boldsymbol{r}_2 \times \boldsymbol{v}_2)||} \tag{5.4}$$

The magnitudes γ_i are the root of the following function [5]:

$$f(\gamma, \pi_2) = \frac{1 - \pi_2}{|\gamma + \pi_2|^3}(\gamma + \pi_2) + \frac{\pi_2}{|\gamma + \pi_2 - 1|^3}(\gamma + \pi_2 - 1) - \gamma \tag{5.5}$$

where π_2 depends on the primary and secondary mass as:

$$\pi_2 = \frac{m_2}{m_1 + m_2} \tag{5.6}$$

5.2.2. Triangular points

The triangular points lie on the vertex of the two equilateral triangles with the bases defined by the m_1-m_2 line.

The position vector of the triangular points is computed as:

$$\boldsymbol{r}_i = \frac{1}{2}\boldsymbol{r}_2\hat{\boldsymbol{x}} + \frac{\sqrt{3}}{2}\boldsymbol{r}_2\hat{\boldsymbol{y}}\delta_i \tag{5.7}$$

where δ_i is equal to $+1$ for $i = 4$ and -1 for $i = 5$, and the versor of the x-axis is defined by the direction of m_2:

$$\hat{\boldsymbol{x}} = \frac{\boldsymbol{r}_2}{||\boldsymbol{r}_2||} \tag{5.8}$$

The velocity vectors can be computed as:

$$\boldsymbol{v}_i = \omega||\boldsymbol{r}_i||\frac{\hat{\boldsymbol{z}} \times \boldsymbol{r}_i}{||\hat{\boldsymbol{z}} \times \boldsymbol{r}_i||} \tag{5.9}$$

5.3. Check implementation

As discussed in Section 5.1, just the SEL points have been implemented. The γ_i for the SEL points have been found as the root of Eq. (5.5) and have been hard-coded in CUDAjectory. The user can select which SEL points are controlled by the algorithm, called activated bodies, and the radius of the collision sphere around each selected point, R_i. The framework of the algorithm is shown in Fig. 5.2, where C is the current integration centre, A_i are the activated bodies and S is a sample.

The algorithm starts identifying the closest activated body, like in the close-approach procedure, shown in Algorithm 3.1. The closest body is then called k.

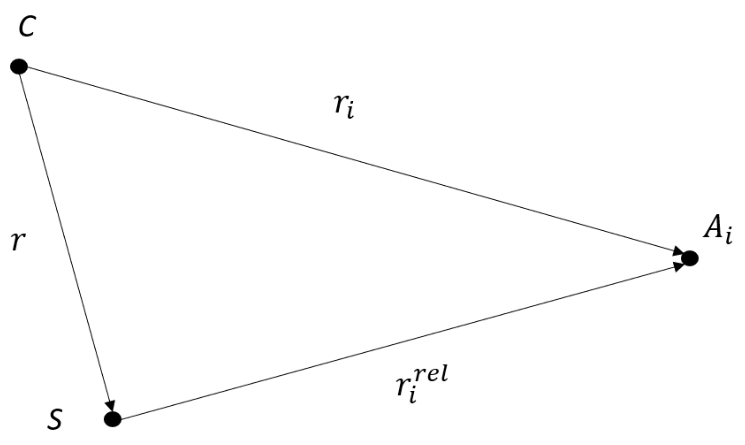


Figure 5.2: Framework of the massless bodies collision algorithm.

5.3.1. LEO protected region similarities

The massless bodies collision event presents some analogies with respect to the LEO protected region event. Indeed, the physical boundary defining the two events is a sphere centred around a body. Instead of the altitude, the physical quantity used to detect the event is the relative distance between the k-th body and the sample (d). Its variation is given by the range rate, computed as:

$$rr = \frac{\boldsymbol{r}_k^{rel} \cdot \boldsymbol{v}_k^{rel}}{d} \tag{5.10}$$

As analogy, the algorithm can be implemented with the same equations of the LEO protected region, substituting d to z, rr to v_r and R_i to z_{LEO}.

The LEO algorithm is based on the linear altitude prediction. Thanks to the gravitational force of the Earth, which is relatively close to the samples, the time steps are typically in the order of few minutes around the perigee. Instead, since the Libration points are modelled as massless points, the resultant time steps are in the order of days. As a consequence, the linear extrapolation it is not very robust, because of the range rate variation in time.

In order to improve the robustness of the prediction, the magnitude of the sample velocity can be used instead of the range rate, being an upper boundary of the range rate itself. This leads to the following prediction:

$$d_{n+1}^{*} = d_n + 5 * ||\boldsymbol{v}||h_n \tag{5.11}$$

The Eq. (5.11) is quite redundant and so it is used to predict and to adapt the time step in the initial iterations, where the prediction using the range rate might not be accurate.

5.3.2. Adaptive step refinement

For the reasons discussed above, the same rationale of the LEO protected region algorithm to adapt the time interval is here used. The complete pseudo-algorithm is shown in Algorithm 5.1. Logical arithmetic is used to minimise the number of branches.

Algorithm 5.1 Massless bodies collision algorithm

```
1: if $((inCollSphere() \ \&\& \ d >= R) \ || \ (!inCollSphere() \ \&\& \ d < R))$ then
2:     save detection
3: else
4:     $d^*_{n+1} = d_n + rr * 5h_n$
5:     if $((inCollSphere() \ \&\& \ d^*_{n+1} >= R) \ || \ (!inCollSphere() \ \&\& \ d^*_{n+1} < R))$ then
6:         $h_{n+1} = (R - d_n)/(5rr)$
7:         $h_{n+1} = h_{n+1} * (h_n|rr| >= toll) + \frac{toll}{|rr|} * (h_n|rr| < toll)$
8:     else
9:         $d^*_{n+1} = d_n + ||\boldsymbol{v}|| * 5h_n$
10:        if $((inCollSphere() \ \&\& \ d^*_{n+1} >= R) \ || \ (!inCollSphere() \ \&\& \ d^*_{n+1} < R))$ then
11:            $h_{n+1} = (R - d_n)/(5||\boldsymbol{v}||)$
12:            $h_{n+1} = h_{n+1} * (h_n||\boldsymbol{v}|| >= toll) + \frac{toll}{||\boldsymbol{v}||} * (h_n||\boldsymbol{v}|| < toll)$
13:        end if
14:    end if
15: end if
```

At line 5, the prediction using the range rate is utilised to check if the boundary might be crossed at the next iteration. If the prediction does not trigger the crossing condition, then the robust condition (Eq. (5.11)) at line 10 is used.

The pseudo-function `inCollSphere()` is needed to establish if at the previous time step the sample is inside the collision sphere.

5.4. Validation

Aiming to validate the algorithm shown in Algorithm 5.1 a dummy test has been used; the input samples have been generated to target SEL1. Once again, the same samples have been propagated in CUDAjectory and in Godot, and the event detections are compared in this section.

The same model has been set in both software:

- two body perturbed dynamics;
- Moon third body perturbation;
- no SRP.

The range of the orbital parameters used to define the initial states of the samples are:

- **perigee**, $r_p = 8000$
- **eccentricity**, e to target SEL1
- **inclination**, $i = 23.4°$
- **RAAN**, $\Omega = 0°$
- **AoP**, $\omega = 0°$
- **TA**, $\theta = 0°$

The initial epoch of the samples is between 7000 and 8000 Modified Julian Date 2000 (MJD2000). For each sample, the eccentricity has been found such to reach SEL1. The integration period is 120 days, in order to reach the SEL1 point. The radius of the collision sphere around the Libration point was set to 10000 km in both software, which is a typical value used to reach the target.

The Fig. 5.3 shows the results of the comparison. The error with respect to the position norm is higher than the previous test cases. However, the maximum difference with respect to the position norm vector is kept under 200km, which is 50 times less than the radius of the typical collision sphere. Moreover, since the integration period and the magnitude of the position vector is higher than in the other tests, this results are also more affected by the intrinsic propagation difference between CUDAjectory and Godot.

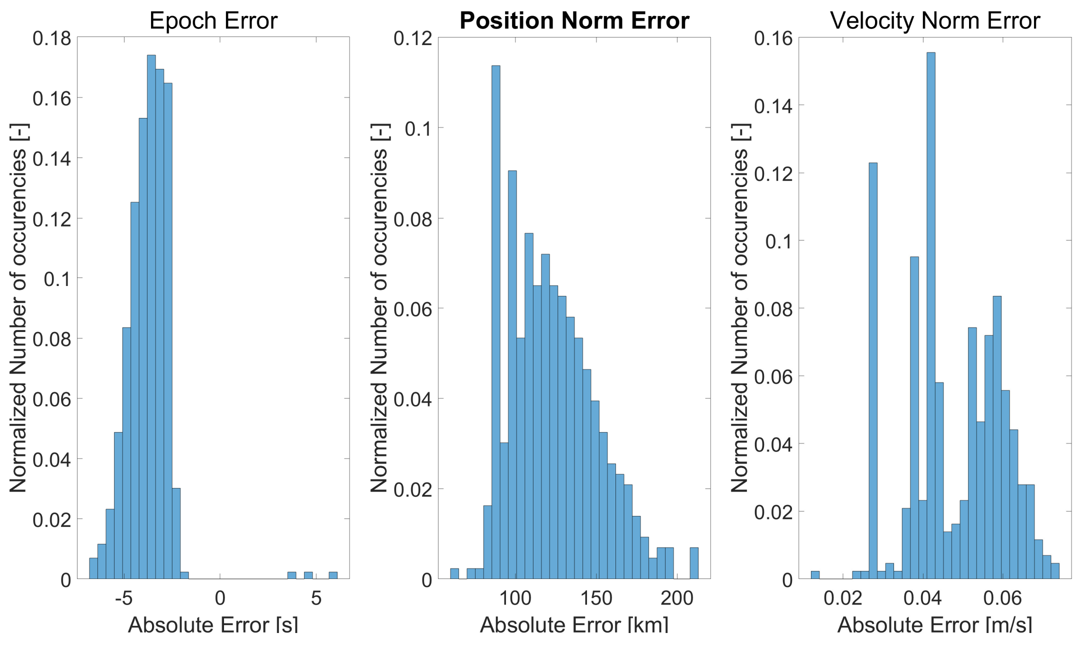


Figure 5.3: Results of the massless bodies collision validation - SEL1 test case.

6 | Output handling optimisation

This chapter presents the development carried out to improve the running time of CUDAjectory. A significant part of the work was devoted to lower the computational effort of the output handling, which is the main bottleneck of CUDAjectory. Depending on the configuration, the output handling may took more than the 99% of the total computational time.

As discussed in Chapter 2, thanks to the configuration file, the user can set the list of events to be controlled during the propagation. The updated set of checks is here listed:

- collision;
- SOI and SOI-crossing;
- altitude;
- close-approach;
- LEO protected region;
- GEO protected region;
- massless bodies collision.

6.1. Output handling

As presented in Section 2.6.4, the output handling performs the following tasks:

1. output array allocation;
2. data-storage on the output array (every kernel launch);
3. output array copy on the VRAM;
4. second output array copy on the host RAM;
5. CPU parsing;
6. final storage inside the result array.

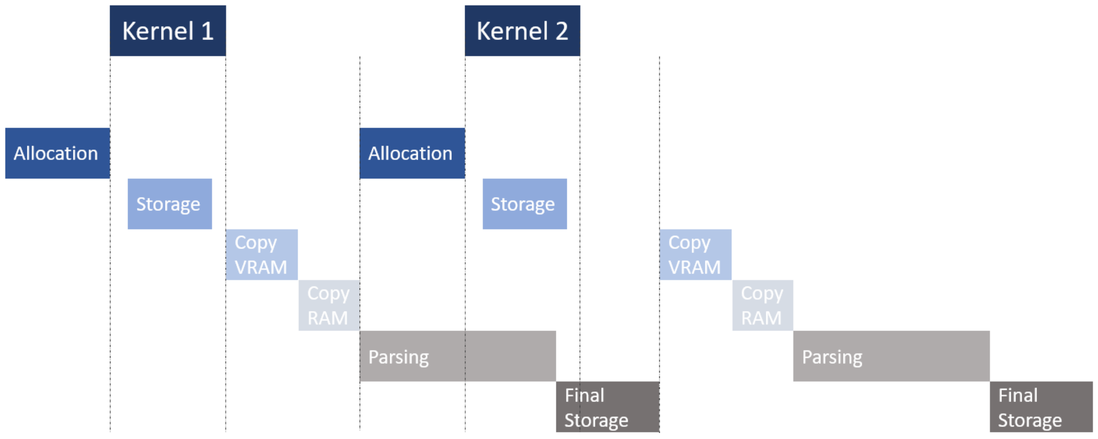


Figure 6.1: Timeline of the output handling during sequential kernels.

In Fig. 6.1, an example of the output handling timeline during the first two launches of the kernel is shown. Task 5 and 6 are executed by the host in parallel with respect to the device, that is executing the following kernel. These two tasks must be completed before the copy operations of the output array of the next kernel, otherwise the data are over-written, as discussed in Section 2.6.4. This two phases are also the most time consuming, leading to the huge performance bottleneck.

6.2. Pinned memory

Nowadays, the operating systems deal with the so called *virtual memory* to store the applications data. The required memory is divided in small chunks of few kilobytes, called *pages*. When the required memory is higher than the physical memory available, the operating system relocates some pages from the physical memory into additional virtual pages. The additional pages are temporarily stored in the computer storage and not in the memory; for this reason, this pages are called virtual, and can be moved back to the physical memory once there is enough space for it.

Host memory allocations are *pageable* by default, which means that can be moved into the storage if needed. Moving pages to the storage may imply a huge performance drop, depending on the application and on the hardware involved. It is possible to request physical, non-pageable memory, which is called *pinned* or *page-locked* memory. The bandwidth between host memory and device memory is higher if host memory is allocated as page-locked [33].

Different types of pinned memory allocations exist. The *mapped* pinned memory maps the locked pages into the device address space. As a consequence, the device can directly store data into the mapped memory. This memory can be also read/wrote by the CPU. Memory transactions from the device to mapped memory communicates over the PCI-express bus, which adds a large amount of latency with respect to utilise the GPU global memory.

6.3. Improved output handler

The mapped memory can be used to allocate the output array; in this way, the kernel can directly store data into the host memory. As a consequence, one less copy-operation of the output array is needed. The higher latency of this type of memory transaction is hidden by the kernel itself. Indeed, this transactions are performed asynchronously with respect to the other computations, leading to almost a 100% of performance uplift during the copies.

Using the mapped memory allocation for the output array leads to the following schedule:

1. output array allocation into the mapped memory;
2. direct storage from the kernel to the host memory;
3. output array duplicate on the RAM;
4. CPU parsing;
5. final storage inside the result array.

6.3.1. New output array structure

As seen in Section 2.6.4 the size of the output array is dependent on the checks selected by the user. Inside the array, every state and epoch, for every active sample and integration step is stored. Moreover, additional space is allocated for each active sample and each step to store the results of the selected checks.

In the previous implementation, each event used to store a different amount of variables per check (e.g. the close-approach stores two doubles while the SOI, only one). During the output array parsing operation, this memory management leads to multiple controls in order to establish to which sample, step, and check a given data is related to. A new output array layout has been designed in order facilitate the parsing operations. A common data-type structure has been implemented, suitable to contain the output of every check. Moreover, the events are mutually exclusive, which means that there is no need to allocate space for each one of them at every time steps for all the samples. So calling the number of samples N, the maximum number of steps n and the size of the common data structure m, the size of the new output array is $N \cdot n \cdot m$.

6.3.2. Data structure choice

The designed structure of data contains the following fields:

- position vector, 3 double, 3x8 bytes;
- velocity vector, 3 double, 3x8 bytes;
- epoch, 1 double, 8 bytes;
- last time interval, 1 float, 4 bytes;
- body-ID, 1 short, 2 bytes;
- check-ID, 1 char, 1 byte;
- type-ID, 1 char, 1 byte.

The state and the epoch are a clear choice; it is crucial to have this information for each detection. The last time interval value has been added in order to have the possibility to further refine the detections once the simulation is completed. A float is sufficient to start a refinement process on the results. The body-ID specifies the celestial body with respect to which the event is verified (e.g. the body at which the sample is collided, or the body with respect to which the close-approach is verified). A short data-type is sufficient to contain its value. In CUDAjectory the checks are distinguished thanks to an integer-check mapping; the check-ID univocally correlates the check that detected an event. The type-ID is a multi-purpose variable. It has been introduced exploiting the single byte that remains free due to data-padding (data are stored in blocks of 8 bytes). This variable has been used to distinguish the entry or exit conditions with respect to the LEO/GEO protected region and massless bodies collision algorithms.

The size of the structure fields has been minimised to lower the amount of RAM required by the simulation. The size of the structure is 64 bytes. The typical number of samples of a statistical analysis can be in the order of millions. Considering ten millions of samples, propagated with the maximum number of steps set to 200, its default value, the size of the output array is $200 \cdot 10000000 \cdot 64$ bytes, corresponding to 128 GB. If the structure is composed by double data type only, the size of the output array is 176 GB. Moreover, this is just the size of the output array, additional space is required for the storage of the samples data and many other variables. The minimisation process of the output array gives the possibility to run simulations with 30% more samples, given the same RAM capacity.

6.4. Graphic card compatibility

Thanks to the mapped pinned memory allocations, as presented in the previous section, the GPU can directly store data into the host RAM from the kernel. The mapped pinned memory can be allocated without any restrictions, however, RAM pinned pages cannot be mapped into the device addresses on every GPUs; modern GPUs are allowed to perform such operation, and the compatibility can be controlled looking at the field `canMapHostMemory` of the `deviceProperties` structure.

Since compatibility across graphic cards might be an issue, both the previous and current handling implementations have been blended into the source code. The user can select and exploit the desired version in the configuration file thanks to a flag, called *mappedMode*. The software also makes sure that the device is compatible with the new handling version, if selected. This implementation grants complete compatibility with respect to all Nvidia CUDA graphic cards.

6.5. RAM allocation compatibility

As said in the previous section, mapped pinned memory can be allocated without any restrictions. However, the developer should carefully perform this type of allocations. Since mapped memory cannot be paged, if an application tries to allocate too much memory, the operating system might crash. Due to the huge potential size of the mapped output array, a control algorithm for over-allocations has been implemented. The algorithm is based on controlling the number of available physical pages to ensure that the buffer fits in it. If an over-allocation is predicted, the user is prompted to reduce the number of samples or the maximum number of steps, in order to reduce the size of the output array.

6.6. Checks support

The output of the altitude and the SOI algorithms, briefly presented in Section 2.4, are managed with the previous output handler even in the case the new version is selected. This is due to the possible coupling with the other event detections (i.e. multiple detections at the same time step). For example, the altitude produces output at every time step. Since the buffer can store one single structure of data at each time step (for every sample), the detections cannot overlap. So the improvements of the new handler is not exploited in simulations in which only the altitude or the SOI algorithms output is needed. However, with the new developed algorithms, the altitude check is deprecated, and it will be removed or revisited soon.

6.7. SOI output-handler decoupling

As discussed in Section 2.4, one of the SOI algorithms (SOI and SOI-crossing) has to be activated in the configuration file in order to update the integration centre when a change of SOI happens. Refreshing the integration centre is needed in the majority of the analyses. However, both algorithms also produce output data when a change of SOI is detected, other than just updating the integration centre.

In most of the analyses, the user is interested in having the possibility to update the integration centre without the need of the output of the algorithm (i.e. knowing the epoch and the state of the samples at the SOI changes). For this reason, in the new development the update of the integration centre has been decoupled from the check output. This decoupling perfectly blends with the new output handling version that is not able to speed-up the output produced by the SOI algorithms.

6.8. Performance comparison

In Chapter 5, a set of 240 samples targeting SEL1 has been used to validate the massless bodies collision algorithm. The epoch of the generated samples spans between 7000 and 8000 MJD2000. In order to find the proper eccentricity of the orbits to reach SEL1, a grid search has been performed. The epochs interval has been sampled with a time interval of 10 days. For each epoch, 10000 eccentricities have been used, uniformly sampled between 2 and 10. This leads to a total of 1.1 millions of combinations (samples). The samples have been propagated in CUDAjectory for 120 days, in order to extract a smaller set that was colliding at SEL1.

This grid search analysis is presented as an example to compare the performances of the new development with respect to the previous version of CUDAjectory. The same set of samples has been propagated three times with the following setups:

- **1st run**, old handler version;
- **2nd run**, new handler version with the SOI-crossing algorithm activated;

- **3rd run**, new handler version with the decoupled version of the SOI algorithm, without producing the output but updating the integration centre.

The second and third run are useful to appreciate the difference in terms of computational time induced by the SOI output decoupling. Indeed, in this analysis, there is no need to have the SOI-crossings output, but it is essential to update the integration centre, when needed.

The three simulations of the grid search have been performed on the main mission-analysis server at European Space Operation Center (ESOC). The main specifications of the server are:

- Intel Xeon Platinum 8176, base clock 2.1 Ghz, boost clock 3.8 Ghz, 28C/56T;
- Nvidia Tesla V100, base clock 1245 Mhz, boost clock 2380 Mhz, 5120 CUDA cores, 80 SM
- 384GB DDR4, 32x12GB

The Table 6.1 shows the computational time of the three simulations. It can be noted that with the old version of CUDAjectory (1st run) more than 99% of the total time is spent to handle the output. With the new development, keeping the SOI-crossing acivated, the total computational time is almost halved. This shows the significant uplift in performance achieved thanks to the new memory buffer type. Since the SOI-crossing output are managed with the previous version of the handler, almost 95% of the total time (i.e. 429 over 456 seconds) is spent during the old handling operations.

The new decoupled SOI algorithm allows to reach a massive uplift in performance. It can be noted that the 3rd run is almost 33 times faster than the 1st run. Clearly, with the new development the output handler does create a huge bottleneck as before.

	total time [s]	**preparation time** [s]	**integration time** [s]	**old handler** [s]	**new handler** [s]
1st run	946	4	5	938	0
2nd run	456	22	5	429	8
3rd run	29	22	5	0	8

Table 6.1: Grid search analysis - performance comparison.

The results of Table 6.1 are confirmed in all the other simulations performed. Using the same server, whenever the analysis can be carried on with the decoupled version of the

SOI algorithm and with the mapped output array, the simulation runs between 20 and 50 times faster than the previous implementation, depending on the settings.

7 | Study cases

This chapter presents two analysis performed with CUDAjectory that highlight the capabilities of the software. As discussed in this work, CUDAjectory does not directly output post-processed results and figures of merit to the user. The events and interference detection implementations allows to perform many types of analysis, depending on the settings and on the post-processing. The aim of the Chapter is to present two preliminary assessments study cases, for pure academic, testing, and validation purposes, without excessive focus on the results themselves.

7.1. Lunar mission: collision assessment

In the context of a lunar landing mission, it is vital to verify the Earth re-entry probability as well as the Moon impact probability in case of failures. The request of this type of assessment comes from planetary protection and orbital debris mitigation guidelines. The analysis is focused on finding the collision probabilities with respect the Moon and the Earth in the case of a post-launch separation failure. The case study follows a previous launch-optimisation for a lunar-landing mission. The launch vehicle is expected to directly inject the spacecraft into a Lunar Transfer Orbit (LTO) with an apogee of 400000 km; the details of the lunch window analysis are not part of this work. At the injection, a separation maneuver between the launcher and the spacecraft is performed. Supposing a failure during the actuation, the spacecraft is forced into an undesired orbit-change. A Monte-Carlo analysis based on the dispersion data with respect to the nominal trajectories has been performed, with the aim of retrieving the probability of collision with the Earth and the Moon.

The analysis focuses on 73 consecutive launch windows, long 3 to 5 days each. The nominal trajectories of each launch opportunity have been perturbed with the launcher dispersion data. Each nominal trajectory has been used to generate 1000 Monte-Carlo samples, leading to 30 millions of samples in total. The set of generated samples has been propagated for 500 years. CUDAjectory is perfectly suited for such large scale analyses as discussed throughout this work. The computational time required to run the simulation, using the test platform presented in Section 6.8, is about 12 hours. The expected computational time required by the previous version of CUDAjectory is about 3 weeks (without the possibility to control additional events as the LEO/GEO protected region crossings), while in a CPU-based environment the simulation is predicted to require about 6 months. The speed-up given by the current version of CUDAjectory with respect to a CPU software is massive, highlighting its computational capabilities.

An offset of 120 km with respect to the Earth surface has been considered in order to account for the atmosphere. The samples have been propagated considering the point-mass gravitational influence of the Earth, the Sun, and the Moon only, without the SRP perturbation.

The results show that the Earth re-entry probability over 500 years is less than 4%. As it can be seen in Fig. 7.1, the collisions are consistently distributed among 500 years. No correlation between the Time Of Flight (TOF) and the earth re-entry velocity has been found.

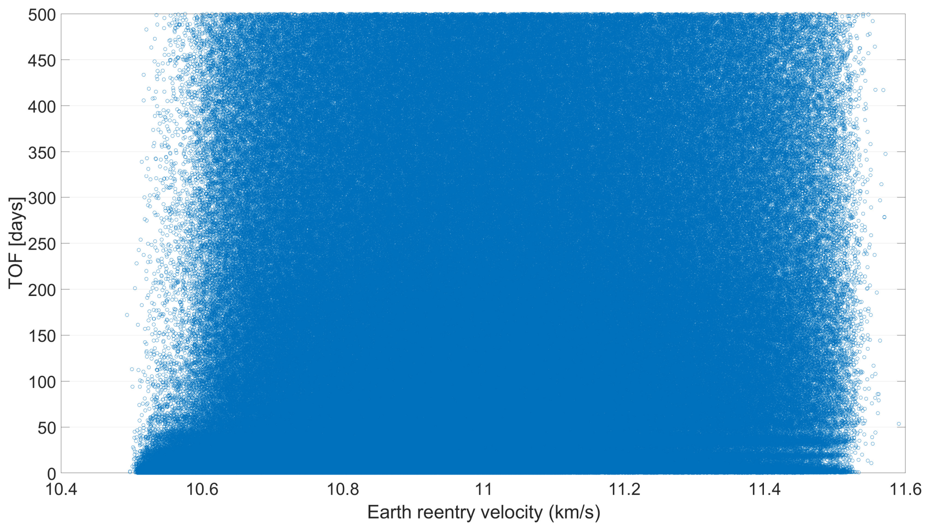


Figure 7.1: Earth re-entry velocity against the time of flight.

The collision probability with respect to the Moon instead is less than 2%. The Fig. 7.2 shows that the samples mostly collided during the first 50 years of propagation, leading to about 0.3% of collision probability after 50 years.

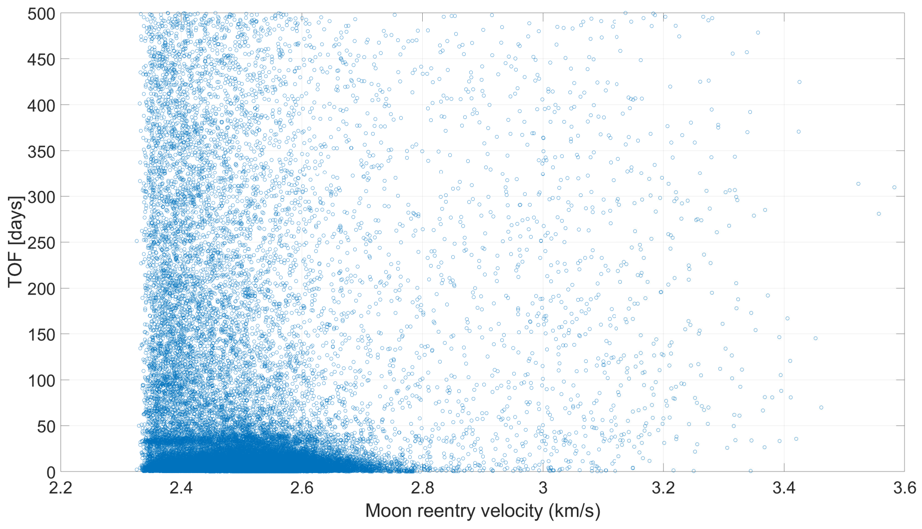


Figure 7.2: Moon impact velocity against the time of flight.

The Fig. 7.3 and Fig. 7.4 show the cumulative density functions (cdf) of the impact velocity with respect to the Earth and the Moon respectively. The impact velocity with respect to the Moon is mostly bounded within 2.4 and 2.6 km/s, which is consistent with respect to the typical Low Lunar Orbit (LLO) velocity. The 99th percentile is 2.62 km/s.

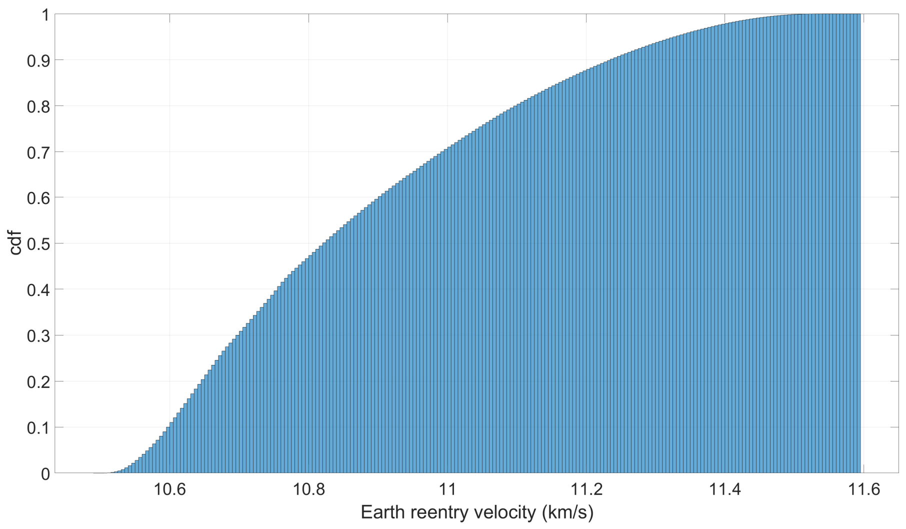


Figure 7.3: Cumulative density function of the Earth re-entry velocity.

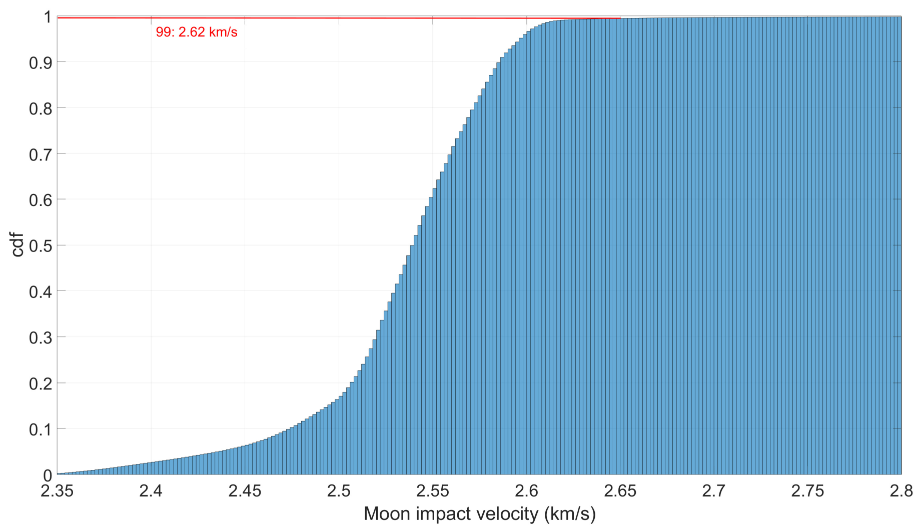


Figure 7.4: Cumulative density function of the Moon impact velocity.

The Fig. 7.5 and Fig. 7.6 show the bivariate distributions of the collision velocity and flight path angle for the Earth re-entries and Moon impacts respectively. It can be noted that the Earth re-entries with a lower velocity are correlated to an higher flight path angle. Indeed, considering a re-entry trajectory with a given infinity velocity, the lower the flight path angle at the collision, the nearer with respect to the perigee, and the higher the collision velocity.

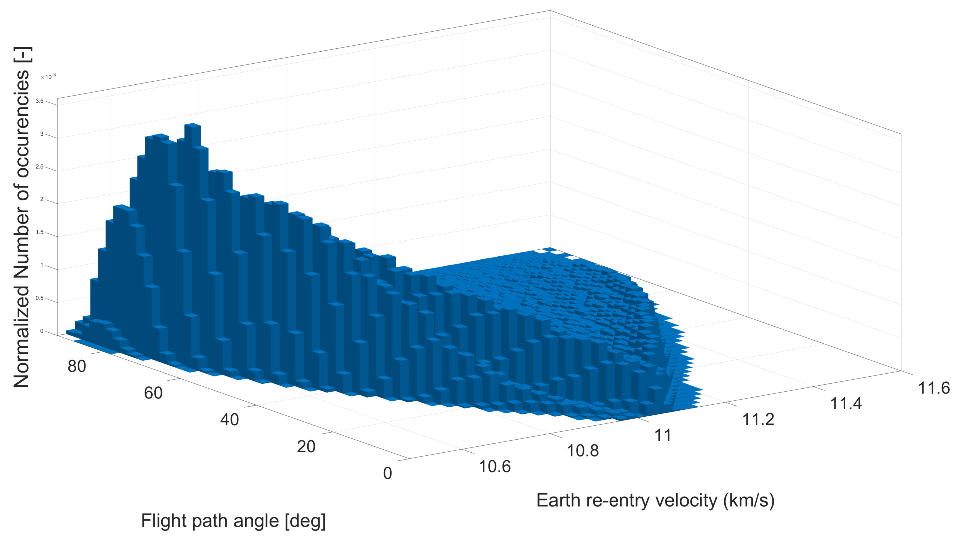


Figure 7.5: Bivariate distribution on the Earth re-entry velocity and flight path angle.

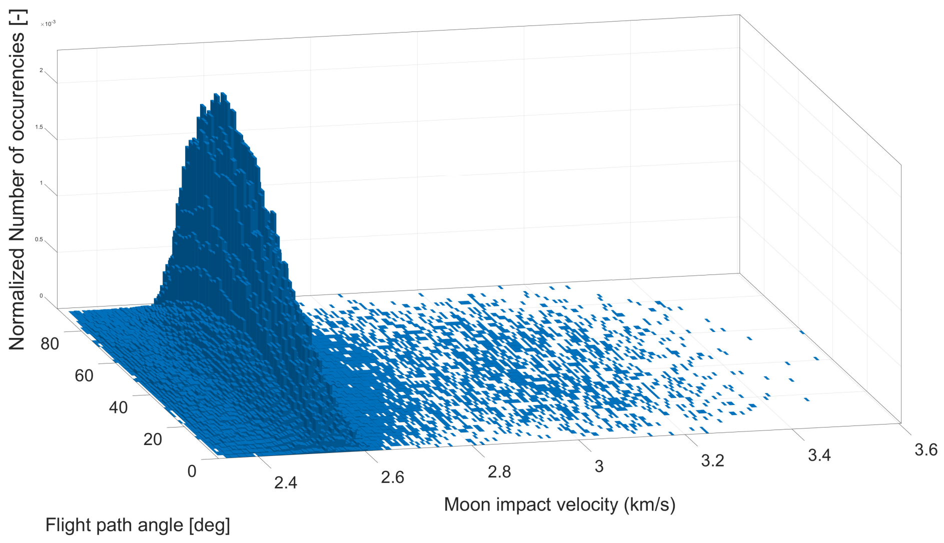


Figure 7.6: Bivariate distribution on the Moon impact velocity and flight path angle.

In Fig. 7.7 the detected impacts are plotted over the Moon surface. The impacts are more clustered in certain patterns, and more scattered in other regions like the polar bands, implying a lower collision probability. Thanks to the retrieved data, it would also be possible to compute the collision probability with respect to specific lunar regions, which could be required for some future lunar mission designs.

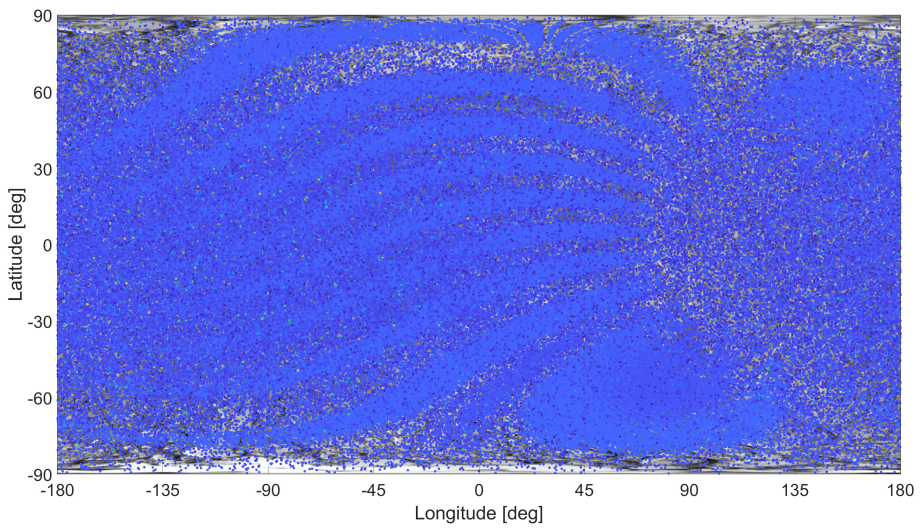


Figure 7.7: Impact positions on the Moon surface.

The propagation has been carried on with both the LEO and GEO protected region algorithms active, discussed in Chapter 4. Although the Earth re-entry probability is about 4%, the percentage of samples that cross at least one time the LEO boundary is 16%. Instead, 99% of the samples cross the GEO boundaries due to the path of the nominal trajectories, which passes through the GEO protected region. Less than 1% of the samples cross the boundaries after the initial outgoing conic. The outcomes show the TOF spent inside the regions ($TOF^{protected}$) is less than a minute for most of the samples; for the LEO region this indicates that most of the perigees of the re-entering orbits are just slightly under the boundary. The statistical interference (I) with respect to the LEO/GEO protected region has been computed as:

$$I = \frac{\sum_{i=1}^{N_{detections}} TOF_i^{protected}}{\sum_{j=1}^{N_{samples}} TOF_j} \tag{7.1}$$

it represents the probability of a sample to be located inside the region at any time. The interference with respect to the protected regions resulted in the order of 10^{-7} for the LEO, and $8 * 10^{-8}$ for the GEO. The cumulative TOF spent inside the LEO and GEO protected regions is respectively about 2500 and 1100 years.

7.2. SEL5 dispersion analysis after a collision

In the context of a mission at SEL5 (e.g. the Vigil Mission [10]), it is important to study the developments after an eventual collision of the spacecraft with respect to a space object. After the collision, the spacecraft is fragmented, generating multiple debris. Assuming a protected region around the Libration point, the study focuses on computing the statistical interference with respect to the protected region. In order to perform a comprehensive and exhaustive analysis, the fragmentation must be statistically computed modelling the break-up of the spacecraft. This is not the focus of the analysis and consequently, the results are not intended to provide any consistent scientific outcome. The aim is to test and show the capabilities of the software, with a realistic use case.

The analysis focuses on a mission at SEL5, assuming a protected region around the Libration point of 1000000 km. The simple adopted break-up model consists in perturbing each component of the spacecraft velocity with a variation (Δv), randomly uniform distributed between -1 and 1 km/s. The study is based on the propagation of 1 millions of samples, generated sampling a fictitious operative orbit around SEL5 into 1000 points, and distributing each instance into 1000 samples, with the break-up model. The samples propagation for 500 years gives an in-depth overview of the developments after the fragmentation. The simulation has been carried on using the test platform presented in Section 6.8, taking 4 hours to run. The expected computational effort to execute the same simulation with a CPU software is 2 months, showing the massive speed-up granted by CUDAjectory against a CPU-based software.

The distribution of the TOFs before the outgoing crossing detections with respect to the assumed protected region shows that most of the events are detected within the first 10 years of propagation. These detections are related to the immediate developments after the fragmentation, causing most of the samples to escape from the protected region within 10 years. In Fig. 7.8 the trend of the TOF spent inside the protected region ($TOF^{protected}$) by the samples is depicted. This set of TOFs has been used to compute the statistical interference with respect to the region itself with Eq. (7.1), which results in the order of 10^{-4}. The cumulative TOF spent inside the region is about 23000 years.

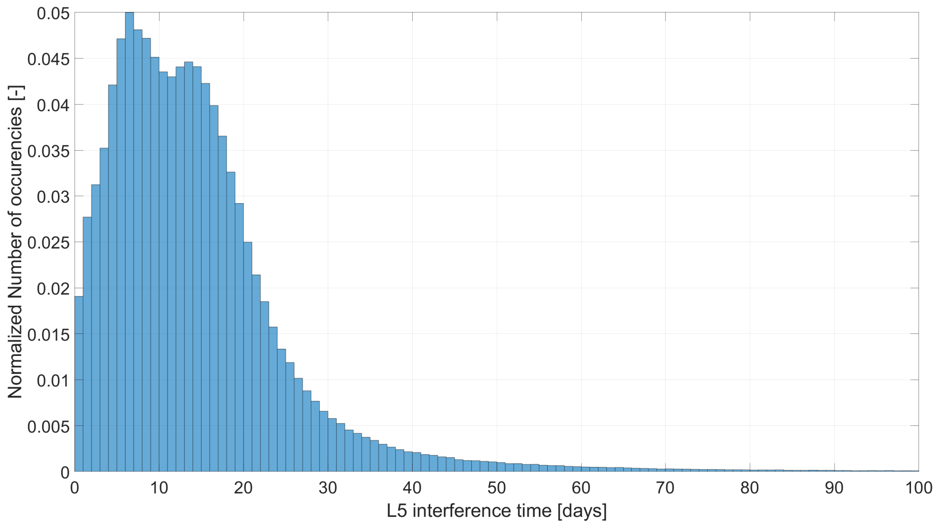


Figure 7.8: Time of flight spent inside the SEL5 protected region.

8 | Conclusions and future developments

Four different event detection algorithms have been shown in chapters 3, 4 and 5. This part of the work proves that is possible to efficiently design GPU routines for interference detections, despite the difficulties of avoiding branch-dependent equations. The usage of adaptive step refinement procedures to adjust the integrational time intervals accordingly to the prediction of interference has been proved to be a valuable and robust programming choice.

Exploiting the pinned mapped memory to directly store the output data in the host RAM has been proven to grant a huge performance uplift, in the case the added latency can be hidden by the kernel operations. Moreover, a different output handling approach and the improvement to the accessibility of the integration centre update, contributed to reach a massive performance uplift, granting up to fifty times faster analyses.

The developed algorithms capabilities have been dictated by the needs of the Mission Analysis team at ESOC, in which the work of this thesis has been carried out remotely. The algorithms have been successfully validated using Godot, a CPU based orbit propagation tool available in ESOC.

The software is still under development, and the work done during this thesis sets the basis for the future developments. In particular, as presented in chapter 5, the current ephemerides implementation of CUDAjectory can be improved, by expanding its capabilities to other SPK-kernel types. Moreover, an interesting follow-up to to improve the software efficiency could be to parallelise the output array parsing operation. Indeed, the array can be divided into smaller segments, and each of them can be read by a different CPU thread. This development shall also leave the possibility to choose the utilised number of CPU threads to the user, allowing to flexibly manage the resource of the server.

Bibliography

[1] IADC. Space debris mitigation guidelines, 7 2021. URL https://iadc-home.org/documents_public/view/id/172#u.

[2] AMD. *Website*. URL https://www.amd.com/.

[3] Archer, Branden, and E. W. Weisstein. *Lagrange Interpolating Polynomial*, 2015. URL https://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html.

[4] L. F. Chaparro. *Chebyshev Polynomials and Lissajous Figures*, 2011. URL https://www.sciencedirect.com/topics/computer-science/chebyshev-polynomial.

[5] H. D. Curtis. *Orbital Mechanics for Engineering Students*. Elsevier, The Boulevard, Langford Lane, Kidlington, Oxford, 2010. ISBN 9780080977478.

[6] ESA. *Mars Express Mission*, Manifest. URL https://www.esa.int/Science_Exploration/Space_Science/Mars_Express?msclkid=b3dad8b1b41511ec99d928ec23b36541.

[7] ESA. *Rosetta Mission*, Manifest. URL https://www.esa.int/Science_Exploration/Space_Science/Rosetta?msclkid=d070fdc7b41511ec8ed66778a389f9c3.

[8] ESA. *SMART1 Mission*, Manifest. URL https://www.esa.int/Science_Exploration/Space_Science/SMART-1?msclkid=dc0d3461b41511eca79f72559b009148.

[9] ESA. *Venus Expreess Mission*, Manifest. URL https://www.esa.int/Enabling_Support/Operations/Venus_Express?msclkid=e5a41ba8b41511ec97ae70bd4735b14d.

[10] ESA. *Vigil Mission*, Manifest. URL https://www.esa.int/Safety_Security/Space_weather/Introducing_ESA_Vigil_Earth_s_devoted_solar_defender.

[11] ESOC. *CUDAjectory repository*, Gitlab. URL https://gitlab.space-codev.org/ad/cudajectory.

[12] ESOC. *Godot repository*, Gitlab. URL https://gitlab.esa.int/godot.

[13] EVGA. *Website*. URL https://www.evga.com.

[14] EVGA. *RTX 3060ti*, ftw3. URL https://www.evga.com/products/product.aspx?pn=08G-P5-3667-KL.

[15] E. Fehlberg. *CLASSICAL FIFTH-,SIXTH-, SEVENTH-, AND EIGHTH-ORDER RUNGE-KUTTA FORMULAS WITH STEP SIZE CONTROL*, 10 1968. URL https://ntrs.nasa.gov/api/citations/19680027281/downloads/19680027281.pdf.

[16] M. Flynn. Encyclopedia of parallel computing, 2011. URL https://doi.org/10.1007/978-0-387-09766-4_2.

[17] M. Geda. Massive parallelization of trajectory propagations using gpus. Master's thesis, Delft University of Technology, 1 2019. URL https://repository.tudelft.nl/islandora/object/uuid%3A1db3f2d1-c2bb-4188-bd1e-dac67bfd9dab.

[18] F. R. Hoots and R. L. Roehrich. *Models for Propagation of NORAD Element Sets*, 12 1988. URL https://celestrak.com/NORAD/documentation/spacetrk.pdf.

[19] *Website*. Institute of Applied Astronomy of the Russian Academy of Sciences. URL https://iaaras.ru/en/.

[20] *EPM Ephemeris*. Institute of Applied Astronomy of the Russian Academy of Sciences, catalogue. URL https://iaaras.ru/en/dept/ephemeris/epm/?msclkid=98635c3ab41611ec99751fdb3da5a630.

[21] R. W. Keyes. Physical limits of silicon transistors and circuits, 9 2005. URL http://stacks.iop.org/RoPP/68/2701.

[22] F. Letizia, C. Colombo, J. P. J. P. V. den Eynde, and R. Armellin. *SNAPP-shot: Suite for the Numerical Analysis of Planetary Protection*, 3 2016. URL https://indico.esa.int/event/111/contributions/331/attachments/430/475/ICATTsnappshot_paper.pdf.

[23] J. C. Liou. Risks from orbital debris and space situational awareness, 1 2020. URL https://ntrs.nasa.gov/api/citations/20200000450/downloads/20200000450.pdf.

[24] Microsoft. *Website*. URL https://www.microsoft.com/.

[25] O. Montenbruck and E. Gill. *Satellite Orbits.* Springer Berlin, 2001.

[26] M. Mudawar. *Main Memory.* Computer Engineering Department, King Fahd University of Petroleum and Minerals, 2019. URL https://faculty.kfupm.edu.sa/COE/mudawar/coe501/lectures/05-MainMemory.pdf.

[27] NAIF-JPL. *DAF required reading.* NASA. URL https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/daf.html.

[28] NAIF-JPL. *SPK Required Reading.* NASA, 10 1999. URL https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/spk.html.

[29] NAIF-JPL. An overview of reference frames and coordinate systems in the spice context, 2020. URL https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/Tutorials/pdf/individual_docs/17_frames_and_coordinate_systems.pdf.

[30] NAIF-JPL. *Website.* NASA, development ephemerides. URL https://naif.jpl.nasa.gov/naif/data.html.

[31] NumPy. *Website.* URL https://www.numpy.org/.

[32] Nvidia. *Website.* URL www.nvidia.com.

[33] *CUDA C++ Programming Guide.* Nvidia, 11 edition, 6 2021. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[34] *CUDA Compiler Driver NVCC.* Nvidia, 11 edition, 3 2022. URL https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf.

[35] S. D. Office. Esa's annual space environment report, 5 2021. URL https://www.sdo.esoc.esa.int/environment_report/Space_Environment_Report_latest.pdf.

[36] openGL. *Website.* URL https://www.opengl.org/.

[37] J. Sanders and E. Kandrot. *CUDA by example : an introduction to general-purpose GPU programming.* Addison-Weasley, Upper Saddle River, NJ, 6 2010. ISBN 9780131387683. URL https://developer.download.nvidia.com/books/cuda-by-example/cuda-by-example-sample.pdf.

[38] F. Schrammel. Alternative ephemeris representations for astrodynamical simulations on accelerators. Master's thesis, Technisch Univeritat Darmstadt, 1 2019. URL https://www.hkhlr.de/en/projects/2040.

[39] T. Soyata. *GPU Parallel Program Development Using CUDA.* Chapman and Hall/CRC, Boca Raton, Florida, 2018. ISBN 9781315368290.

[40] E. W. Weisstein. *Hermite's Interpolating Polynomial*, 2015. URL https://mathworld.wolfram.com/HermitesInterpolatingPolynomial.html.

A | Spice kernel types

This appendix explains the data format, and the different interpolation techniques related to the ephemerides Spice Kernels. The purpose of the Spice SPK files is to combine the ephemerides of solar system bodies under a common file format. These file contain the required set of data to retrieve the state of one or more tracked bodies in time, between an initial and a final epoch.

The SPK files are written in a Spice-standard format, called DAF. The complete explanation of the DAF architecture can be found in [27]. DAF are composed by multiple parts, called *records*. The initial records are used to store data describing the file content and layout, and for additional comments. The following records contain the array of ephmerides data. Each record contains up to 128 double precision numbers.

The purpose of having multiple SPK types is to efficiently describe the motion of different bodies. Indeed, the various types contain different data formats. The data of each type is designed to be exploited with a peculiar numerical interpolation technique, in order to retrieve the state of a certain space object. The following enumeration lists each SPK type and the its relative interpolation technique, together with the typology of celestial bodies that the SPK type is more suited to describe:

1. **Modified Difference Arrays**. This data format has been created by the JPL Orbit Determination Program. It is a unique form used by the JPL orbit determination software for spacecraft ephemerides [28];

2. **Chebyshev polynomials - constant interval** [4] (position only). This SPK type furnish the set of polynomials coefficients to interpolate the position of the body, exploiting the Chebyshev polynomials interpolation technique. Furthermore, the velocity can be retrieved with the derivative formulation of the interpolation technique [17]. The coefficients are given to interpolate within time intervals of constant lengths. This data type is normally used for planetary ephemerides [28];

3. **Chebyshev polynomials - constant interval** (position and velocity). Same as type 2, but also gives the polynomials coefficients to directly retrieve the velocity

vector. Generally used for satellites whose orbits are computed with theoretical models and not observed [28];

4. reserved for future use;

5. **Discrete states - two body propagation**. This SPK format contains discrete state vectors at multiple epochs and the respective epochs. The retrieve process of the intermediate states from this format is designed to give similar results to a two-body motion. The intermediate states can be approximated thanks to a weighting function computed as function of the time interval and the epoch to be evaluated. This data type is typically used for comets and asteroids whose orbits are integrated from an initial state[28];

6. reserved for future use;

7. reserved for future use;

8. **Lagrange Interpolation - constant interval** [3]. The Lagrange interpolation technique is used to retrieve the intermediate states, starting from the given sampled states. This data format is rarely used [28];

9. **Lagrange Interpolation - variable interval** [3]. Same as type 8 but with variable time intervals. So, both the sampled states and the relative epoch are given. This SPK type is versatile to describe all the bodies [28];

10. **Space Command Two-Line Elements** [18]. The SPK data Type 10 stores a collection of packets each of which models the trajectory of some near-Earth satellites using the Space Command two-line element sets. Typically is used for Earths orbiters [28];

11. reserved for future use;

12. **Hermite Interpolation - constant interval** [40]. This SPK type contains discrete state vectors referred to equally spaced epochs. The intermediate states can be computed with the Hermite Interpolation technique. This SPK type is rarely used [28];

13. **Hermite Interpolation - variable interval**. Same as type 12 but with variable time intervals. This SPK type is versatile to interpolate all the motions [28];

14. **Chebyshev polynomials - variable interval** (position and velocity). Same as type 3 but with variable time intervals. The most flexible Chebyshev interpolation, suitable for all the bodies;

15. **Precessing Conic Propagation**. This SPK type represents a continuous ephemeris using an analytic model. The object is modeled as orbiting around a central body under the influence of a central mass plus first order secular effects of the J2 term [28]. The data contain the coefficient needed to define the model. Rarely used in the distributed kernels [28];

16. reserved for future use;

17. **Equinoctial Elements**. This SPK type represents a continuous ephemeris using an analytic model. The object described is in an elliptic orbit with precessing line of nodes and AoP relative to the equatorial frame of a central body [28]. The file contains the terms to define the equinoctial elements;

18. **ESOC/DDID Hermite/Lagrange Interpolation**. Spacecraft ephemerides used by the European Space Agency (ESA) on the Mars Express [6], Rosetta[7], SMART-1[8], and Venus Express missions[9] [28];

19. **ESOC/DDID Piecewise Interpolation**. Spacecraft ephemerides used by the European Space Agency (ESA) on the Mars Express, Rosetta, SMART-1, and Venus Express missions [28];

20. **Chebyshev** (velocity only). This data type is used to retrieve the velocity of a given body. It is used to represent the EPM ephemerides [20] developed by the Institute of Applied Astronomy [19]. The position of the body is obtained by integrating the velocity using a specified integration constant [28];

21. **Extended Modified Difference Arrays**. These data structures use the same representation as type 1, but with higher order of degree. It is a unique form used by JPL orbit determination software for spacecraft ephemerides [28];

In CUDAjectory, the SPK types 2 and 3 are currently implemented, because the set of bodies whose state has to be estimated through the ephemerides evaluation were the planetary bodies.

List of Figures

1.1 EVGA RTX 3060ti ftw3 graphic card and its PCB layout [14]. 7
1.2 Example of a Nvidia Fermi GPU architecture [39]. 8
1.3 Fermi GF110 microarchitecture [39]. 9
1.4 Graphical view of a bi-dimensional block and grid arrangement [33]. 11

3.1 Framework of the close-approach algorithm. 28
3.2 Timeline of the close-approach event during the integration. 30
3.3 Timeline of the example. 32
3.4 Results of the close-approach comparison. 36

4.1 Earth protected regions - IADC conventions. 40
4.2 Results of the LEO protected region validation. 46
4.3 Results of the GEO protected region validation. 47

5.1 Roto-pulsating reference system. 50
5.2 Framework of the massless bodies collision algorithm. 52
5.3 Results of the massless bodies collision validation - SEL1 test case. 55

6.1 Timeline of the output handling during sequential kernels. 58

7.1 Earth re-entry velocity against the time of flight. 66
7.2 Moon impact velocity against the time of flight. 67
7.3 Cumulative density function of the Earth re-entry velocity. 67
7.4 Cumulative density function of the Moon impact velocity. 68
7.5 Bivariate distribution on the Earth re-entry velocity and flight path angle. 68
7.6 Bivariate distribution on the Moon impact velocity and flight path angle. . 69
7.7 Impact positions on the Moon surface. 69
7.8 Time of flight spent inside the SEL5 protected region. 71

Acknowledgements

I would like to express my gratitude to my advisor, Camilla Colombo, who proposed me to follow this project and guided me throughout it. I wish to show my appreciation to Florian Renk for giving me the opportunity to carry out this work. I wish to acknowledge the support provided by the mission-analysis team at ESOC, I always felt part of the team. I would also like to show my very deep appreciation to Lorenzo Bucci who helped me throughout my project and always been present and available. I wish to extend my special thanks to Alessandro Masat for supervising me during the work, and helping me in writing this thesis. I wish to credit the student association Skyward Experimental Rocketry that infinitely helped me in opening my mind and growing my programming skills, I would never been able to carry out this project without the years spent in Skyward. I would also like to warmly thank all of my friends, my family, and my girlfriend who supported me throughout my studies and this work.