GPU-BASED AUGMENTED TRAJECTORY PROPAGATION: ORBITAL REGULARIZATION INTERFACE AND NVIDIA CUDA TENSOR CORE PERFORMANCE

Alessandro Masat[(1)], **Camilla Colombo**[(2)], **Arnaud Boutonnet**[(3)]

[(1)]PhD Candidate, Department of Aerospace Science and Technology, Via G. La Masa 34, 20156, Milano, Italy, alessandro.masat@polimi.it
[(2)]Associate Professor, Department of Aerospace Science and Technology, Via G. La Masa 34, 20156, Milano, Italy, camilla.colombo@polimi.it
[(3)]Senior Mission Analyst, OPS-GFA, ESA-ESOC, Robert-Bosch-Str. 5, D-64293 Darmstadt, Germany, arnaud.boutonnet@esa.int

ABSTRACT

The high non-linearity of the Cartesian equations of the orbital motion represents a limit for the computational efficiency of high-fidelity numerical simulations. The implicitly straightforward interface with physical forces and perturbing accelerations has made the Cartesian formulation the most popular dynamics description in numerical simulations, allowing the accurate implementation of complex and high-precision force models. Yet, considering the trajectory propagation per se, Cartesian coordinates are far from being the best dynamics formulation, either for accuracy and computational efficiency. Element-based formulations that possibly include the regularization of the equations of motion are widely considered the best strategies to compute orbits. The lack of simplicity and the complex physical interpretation behind these techniques has perhaps represented the major limitation in the adoption thereof. Hence, this work implements, interfaces, and studies the performance of the Kustaanheimo-Stiefel (KS) formulation of the dynamics, in direct application to uncertainty propagation and planetary protection tasks, implemented in the CUDAjectory GPU software.

1 INTRODUCTION: THE KUSTAANHEIMO-STIEFEL FORMULATION

KS variables were created bringing together two concepts: the first, adding a fourth coordinate trying to describe the Kepler problem as a four dimensional harmonic oscillator, originally presented by Kustaanheimo [1], and the introduction of the time transformation of the Sundman type [2], operated later with the contribution of Stiefel [3]. The key concept is about converting the integration independent variable, switching from the physical time t to the fictitious time s. The generic Sundman transformation [2]:

$$\frac{ds}{dt} = \frac{\beta}{r} e^{\int K dt} \tag{1}$$

In general, β is an arbitrary constant coefficient and K is an arbitrary function of position, velocity and time. For instance, setting $\beta = 1$ makes the fictitious time s scale like the eccentric anomaly, whereas $\beta = 2$ introduces a true anomaly-like fictitious time evolution. The transformation of the independent variable from the physical time t to the fictitious time s is commonly called *regularization* of the orbital dynamics.

The Kustaanheimo-Stiefel (KS) formulation rewrites the two body problem as an isotropic[1], four-dimensional harmonic oscillator. The conservation of the orbital energy is also introduced, leading to a simple linear ordinary differential equation. The first formulation was proposed indeed by Kustaanheimo and Stiefel [3], [4], and extended the usual Cartesian position vector $\mathbf{r} = \{r_1, r_2, r_3\}^T$ into a four-vector, by adding the length $r = \sqrt{\mathbf{r} \cdot \mathbf{r}}$ as fourth coordinate. Using the initial formulation proposed by Kustaanheimo, the physical coordinates are linked to the spinor regularized coordinates $\mathbf{u} = \{u_1, u_2, u_3, u_4\}^T$ through:

$$\begin{aligned} r_1 &= u_1^2 - u_2^2 - u_3^2 + u_4^2 \\ r_2 &= 2\big(u_1 u_2 - u_3 u_4\big) \\ r_3 &= 2\big(u_1 u_3 + u_2 u_4\big) \\ r &= u_1^2 + u_2^2 + u_3^2 + u_4^2 \end{aligned} \tag{2}$$

later re-arranged in matrix-vector product as

$$\mathbf{r} = \mathbf{L}(\mathbf{u})\mathbf{u} \tag{3}$$

which gives $\mathbf{x} = \{r_1, r_2, r_3, 0\}^T$, with

$$\mathbf{L}(\mathbf{u}) = \begin{bmatrix} u_1 & -u_2 & -u_3 & u_4 \\ u_2 & u_1 & -u_4 & -u_3 \\ u_3 & u_4 & u_1 & u_2 \\ u_4 & -u_3 & u_2 & -u_1 \end{bmatrix} \tag{4}$$

Setting $\beta = 1$ and $K = 0$ in Equation (1) to operate the time transformation, replacing $\mathbf{r} = \mathbf{L}(\mathbf{u})\mathbf{u}$ and its derivatives, the KS transformation converts the Kepler two-body problem

$$\ddot{\mathbf{r}} = -\frac{\mu}{r^3}\mathbf{r} \tag{5}$$

into

$$\mathbf{u}'' = \frac{\epsilon}{2}\,\mathbf{u} \tag{6}$$

where $\ddot{(\cdot)}$ and $(\cdot)''$ stand for second t and s derivatives respectively, and ϵ denotes the two-body orbital energy. Because the adopted Sundman regularization is of the first order, for unperturbed orbits the new independent variable s follows the evolution of the eccentric anomaly. The achieved behavior reflects a sort of slow motion movie for the near-pericenter part of the orbit, with smaller physical time steps the closer the trajectory gets to the attractor.

1.1 Perturbed problem

Perturbing physical accelerations $\mathbf{f}$ are accounted for simply by an additional non-null term on the right hand side of the equations of motion. The generic perturbed two-body problem in Cartesian coordinates is

$$\ddot{\mathbf{r}} = -\frac{\mu}{r^3}\mathbf{r} + \mathbf{f}(\mathbf{r}, t) \tag{7}$$

And following [5] the final expression in KS coordinates becomes

$$\mathbf{u}'' - \frac{\epsilon}{2}\mathbf{u} = \frac{r}{2}\mathbf{L}(\overline{\mathbf{u}})\mathbf{f}(\mathbf{r}, t) \tag{8}$$

with $\overline{\mathbf{u}} = \{u_1, -u_2, -u_3, -u_4\}^T$.

[1] All the four components oscillate with the same frequency and phase.

Since the physical time t can appear in $\mathbf{f}(\mathbf{r}, t)$ either implicitly or explicitly, it should still be tracked, even though its removal was necessary to reach a simpler form of the equations of motion. Closed form expressions for $t = t(s)$ cannot be found for the perturbed case, whereas s evolves like the eccentric anomaly [5], up to a constant, in the unperturbed two-body problem. However, the relation $dt/ds = r$ can be used to add the physical time as another state element for numerical integrations accounting for perturbations. The two-body energy ϵ can either be computed at each time step with

$$\epsilon = -\frac{1}{r}\left(\mu - 2|\mathbf{u}'|^2\right) \tag{9}$$

or be added as another state element as well, and its derivatives are defined as:

$$\begin{aligned} \epsilon' &= \mathbf{r}' \cdot \mathbf{f}(\mathbf{r}, t) \\ \dot{\epsilon} &= \dot{\mathbf{r}} \cdot \mathbf{f}(\mathbf{r}, t) \end{aligned} \tag{10}$$

1.2 Barycentric KS formulation

The original KS formulation of the orbital dynamics requires the reference frame to be centered in one body, that serves both as regularization point and primary attractor for the computation of the orbital energy. However, a more complex dynamics that does not necessarily follow a dominantly two-body trajectory could not benefit from the KS regularization if it was kept in its standard form. Moreover, the barycentric formulation of the dynamics builds a more efficient simulation already in the Cartesian form, since tidal terms are not present [6]. With the purpose of building a simulation setup that remains as general as possible in the cases it can efficiently tackle although still featuring the core benefits of the KS formulation, a barycentric formulation of the KS equations of motion is derived. For the sake of conciseness only the perturbing effects of the N bodies are presented.
Keeping the frame centered on one of the N-bodies brings:

$$\mathbf{u}'' - \frac{\epsilon}{2}\mathbf{u} = -\frac{r}{2}\sum_{\substack{i=1 \\ i\neq i_p}}^{N}\left(\frac{\mu_i(\mathbf{r}-\mathbf{r}_i)}{|\mathbf{r}-\mathbf{r}_i|^3} + \frac{\mu_i \mathbf{r}_i}{|\mathbf{r}_i|^3}\right)\overline{\mathbf{u}}^* \tag{11}$$

with i_p identifying the primary body, included in the definition of the two-body energy ϵ, $\mathbf{r}_i$ and μ_i position vector with respect to the primary and gravitational parameter of the i-th body, respectively. A few modifications are required to write the dynamics centered in the barycenter of the N bodies involved [6]. In general, through the regularization, smaller physical time steps are implicitly taken in the proximity of the center of the reference frame. Its correspondence with the main attractor in the Keplerian problem is not necessary, but becomes convenient when combined with the expression for the orbital energy ϵ. In the barycentric case, the state $\mathbf{r}$ does not identify the position with respect to the primary, therefore every single gravitational contribution must be included among the right hand side terms. Equation (8) for the barycentric state becomes:

$$\mathbf{u}'' = \frac{|\mathbf{u}'|^2}{r}\mathbf{u} - \frac{r}{2}\sum_{i=1}^{N}\frac{\mu_i(\mathbf{r}-\mathbf{r}_i)}{|\mathbf{r}-\mathbf{r}_i|^3}\overline{\mathbf{u}}^* \tag{12}$$

2 CUDAJECTORY GENERAL ARCHITECTURE

CUDAjectory [7] is a GPU object-oriented software for massively parallel trajectory propagation, using the Runge-Kutta-Fehlberg RKF78 integration method. As the name suggests, CUDAjectory is written in CUDA® C++, making it thus suitable to be run with NVIDIA® graphics cards.

CUDAjectory has been developed and is maintained by the Mission Analysis section at the European Space Operations Centre (ESOC), at ESA, and is available to the general public by an ESA Community License. The first implementation follows the Master's thesis work of Geda [8], including most of the model features, including gravitational effects from the Solar System bodies, solar radiation pressure, spherical harmonics perturbation for Earth orbits, and high-order perturbations based on point mascons [8]. CUDAjectory manages ephemeris data in a unique way, with data written in the `CUBE` (acronym for CUdajectory Binary Ephemeris) format and accessed from texture memory, improving the performance of the software. In this way, CUDA® threads can access ephemeris data more efficienly than in lookup-based approaches (commonly adopted with SPICE's `spk` files [9]). More recent works have widened the force pool with an atmospheric model and aerodynamic drag acceleration, as well as the capability of interrupting simulations based on the total radiation absorption in the Jovian environment. Additionally, Inno [10] improved the efficiency of memory management and event detection routines.
Figure 1 shows the overall software structure. Other than the pure command line interface, CUDAjectory includes a PyBind® [11] interface, that exposes CUDAjectory's simulators to Python programs and Python Numpy®[12] input/output array types, providing Python users a simple interface to the GPU parallelized tools. For the command line execution of CUDAjectory, input/output samples and trajectories are given in `csv` format, while the only configuration file required can be written either in `json` [13] or `yml` [14] format, for the Python library implementation as well.

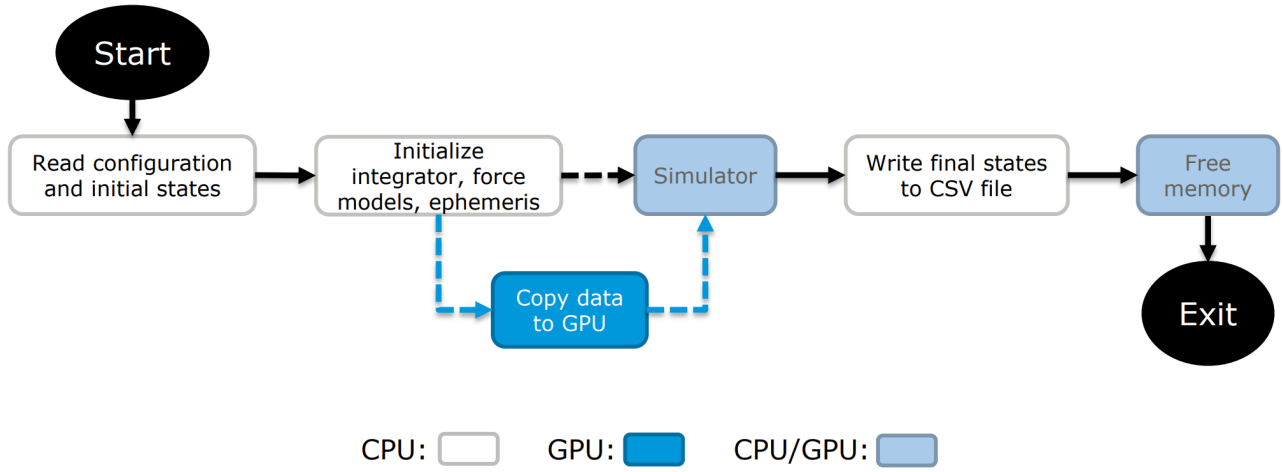

Figure 1: CUDAjectory overall structure. Picture from [8].

Figure 2 shows instead the simulation approach implemented in `ClusterSimulator`. A limited set of steps is taken on the GPU, then the simulation data are passed to the CPU for storage and processing. The process is repeated in a `while` loop until the end of the simulation is reached. This approach has the advantage of reducing the occupancy in the device memory, since, if the full trajectories are stored, they are gradually stored on the CPU and keep the limited GPU memory free. Additionally, the simulation samples can be sorted to maximize the software performance, for instance gradually removing early-ending samples (e.g. in case of detection of impacts).

3 KS VARIABLES IN CUDAJECTORY

Following the just outlined requirements, CUDAjectory has been extended completely hiding KS variables at the core of the software. All the `Simulator` classes in CUDAjectory have been extended with a new method that runs KS variables, with the user interface with the software that remains unchanged. In any case, CUDAjectory can still be run in the traditional Cartesian setup, with the

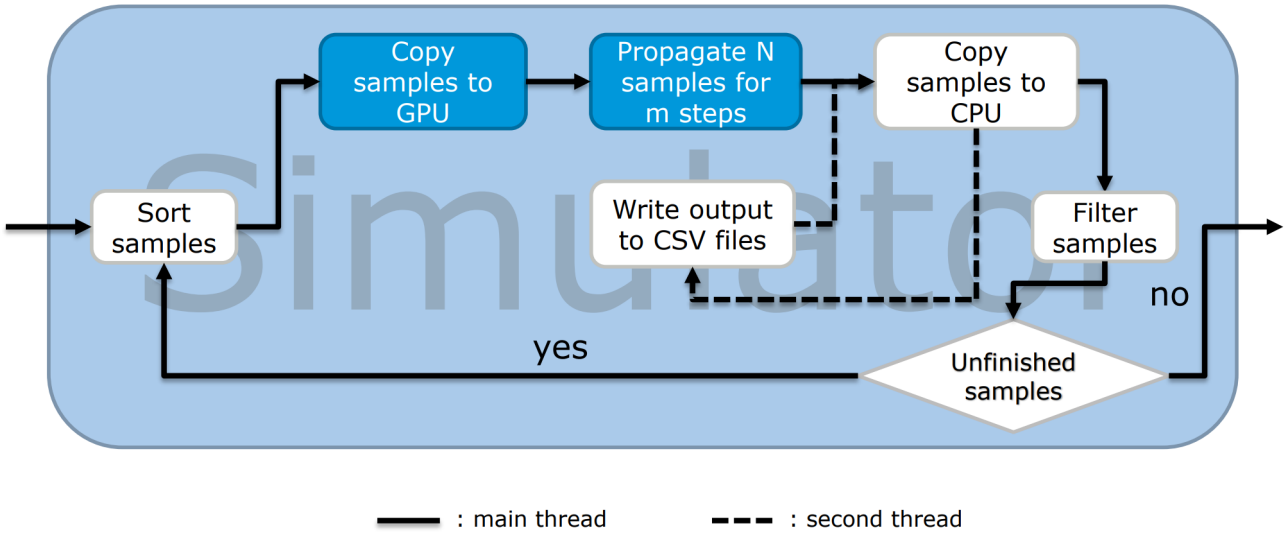

Figure 2: `ClusterSimulator` structure in CUDAjectory. Picture from [8].

usual `evolve()` method, while the KS run is launched with the `ksevolve()` routine. Figure 3 shows the conceptual extension that `Simulator` classes in CUDAjectory have undergone to run with KS variables.

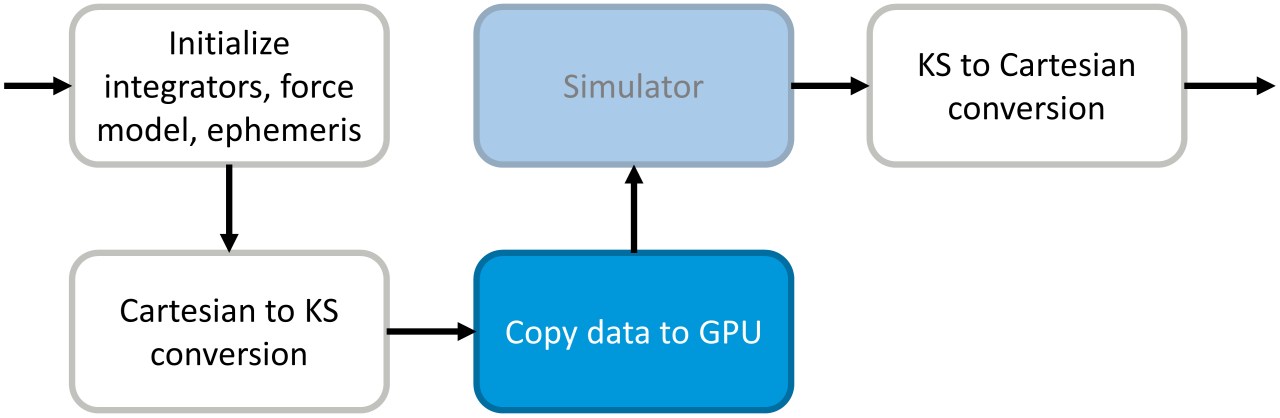

Figure 3: CUDAjectory overall extension logic.

Taking a closer look to the `Simulator` block, the transformation from KS to Cartesian coordinates needs to be implemented also for the storage of intermediate simulation steps, as highlighted in red in Figure 4. Internally in the "Propagate N samples for m steps" block, the given force model is interfaced with KS variables according to Equations (8) and (12). Events are instead computed by retrieving the Cartesian states, at each step.

4 RESULTS AND PERFORMANCE

The proposed implementation of KS variables in CUDAjectory is tested with the propagation of the dispersion of Ariane 5's upper stage of launcher, for the JUICE [15] mission case. Boutonnet and Rocchi used CUDAjectory [16] to assess the compliance of different possible launch scenarios with space debris mitigation policies, computing the impact probability of this dispersion with Earth. The initial dispersion data are given in terms of nominal interplanetary injection state and the related covariance matrix, both covered by a non-disclosure agreement between the ESA and Arianespace.

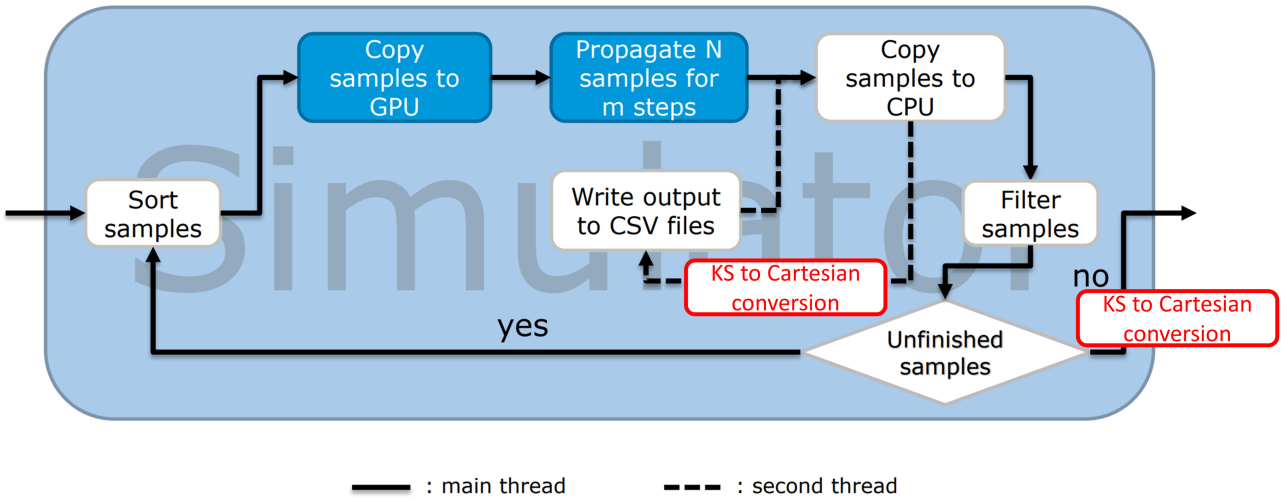

Figure 4: CUDAjectory `Simulator` extension logic. Picture modified from [8].

Nevertheless, the results of the CUDAjectory implementation can still be studied with a detailed analysis of the output trajectories.
The proposed analysis focus on the achieved runtime reduction, validating the computation of both `evolve()` and `ksevolve()` propagating the same initial conditions and comparing each sample outcome against a higher precsion `evolve()` simulation. The test cases have been run in a Ubuntu® Linux® 22.04 machine, equipped with two Intel® Xeon® Platinum 8352V (36 cores, 72 logical @ 2.1 GHz) and a Nvidia® RTX A6000 graphics card. Absolute and relative tolerances for the numerical schemes have been set equal to 10^{-12}, and the samples have been propagated 100 years forward in time.
Tables 1 and 2 show the runtimes for the different CUDAjectory calls, with and without storage of the full propagated trajectories, respectively. KS variables visibly improve the efficiency of the software, which increases for increasing samples. In case of stored trajectory, the improvements become even more relevant, achieving a runtime reduction of over 60% in the case of 20k propagated samples. Other than the reduction of the steps taken presented in [6], the reason for this additional performance enhancement is explained by the lower amount of states that CUDAjectory's output handlers need to manage, resulting in an overall reduced memory and output processing overhead. In fact, the runtime reduction without trajectory storage remains consistently around the expected 40 %.

Table 1: CUDAjectory runtimes for different configurations, with trajectory storage. Average of 5 different runs.

	Samples and Runtime [s]						
	1	10	100	1k	2k	10k	20k
`evolve()`	3.03	3.33	4.93	5.44	7.17	29.00	68.56
`ksevolve()`	1.91	2.09	2.90	3.13	3.24	12.20	23.95
Reduction	36.9%	37.3%	41.3%	42.5%	54.9%	57.9%	65.1%

Table 3 shows the number of different sample outcomes between the different CUDAjectory calls, against a higher precision `evolve()` call with 10^{-14} relative tolerance. The two simulation strategies appear to be equivalent in terms of accuracy. While in general KS simulations are more precise and numerically stable, as shown in [6], the low value of absolute tolerance may have allowed the Cartesian simulation to bridge the accuracy gap with the KS variables, in this particular setup. In any case, the lower runtime makes KS simulations the most suitable choice, given that the overall

Table 2: CUDAjectory runtimes for different configurations, without trajectory storage. Average of 5 different runs.

	Samples and Runtime [s]						
	1	10	100	1k	2k	10k	20k
`evolve()`	2.97	3.22	4.69	5.01	5.12	6.10	8.60
`ksevolve()`	1.83	2.05	2.83	3.02	3.06	3.60	5.24
Reduction	38.5%	36.4%	39.7%	39.7%	40.2%	41.0%	39.0%

accuracy is preserved.

Table 3: CUDAjectory outcome difference against a 10^{-14} relative tolerance `evolve()` simulation.

	Samples						
	1	10	100	1k	2k	10k	20k
`evolve()`	0	0	0	0	0	2	6
`ksevolve()`	0	0	0	0	0	2	5

5 CONCLUSION

The implementation of KS variables successfully introduces notable performance improvements in CUDAjectory. Future implementation works should mainly extend the software flexibility in KS variables. Storing the full trajectories in KS variables would allow future researchers to benefit from the renewed computational efficiency, while having a full KS output at their disposal.
CUDAjectory is a propagation-only related software, thereby limiting the extension possibility to improved input/output management, propagation techniques, and other dynamical formulations. Nonetheless, CUDAjectory may become the simulator at the core of more advanced uncertainty software tools, that may exploit the potential of GPU computing even beyond the trajectory propagation block.
While Tensor Cores represent a novel computational resource embedded in GPUs, the use thereof in CUDAjectory is not straightforward. Tensor cores require a unique memory management strategy, and are designed to perform 2×2 matrix multiplications in one clock cycle [17]. Hence, unless new APIs are developed by Nvidia® in the upcoming CUDA® release to efficiently manage Tensor Cores, existing algorithms would need to be completely rewritten to exploit this computational resource.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme as part of project COMPASS (Grant agreement No 679086), www.compass.polimi.it.
The research is co-funded by the European Space Agency (ESA) through the Open Space Innovation Platform (OSIP) co-funded research project "Robust trajectory design accounting for generic evolving uncertainties", Contract No. 4000135476/21/NL/GLC/my.
This research was supported by grants from NVIDIA® and utilized the NVIDIA RTX A6000 graphics card, awarded by NVIDIA®'s Applied Research Accelerator Programme to the project "GPU-accelerated algorithms for orbital uncertainty simulation applications".

REFERENCES

[1] P. Kustaanheimo, *Spinor Regularization of the Kepler Motion*. Turun Yliopisto, 1964. [Online]. Available: https://books.google.it/books?id=ZhQ9ywAACAAJ.

[2] K. F. Sundman, "Mémoire sur le problème des trois corps," *Acta Mathematica*, vol. 36, 105–179, 1913. DOI: 10.1007/BF02422379.

[3] P. Kustaanheimo, H. Schinzel, and E. Stiefel, "Perturbation theory of Kepler motion based on spinor regularization," *Journal fur die reine und angewandte Mathematik*, vol. 1965, no. 218, pp. 204–219, 1965. DOI: 10.1515/crll.1965.218.204.

[4] E. L. Stiefel and G. Scheifele, *Linear and Regular Celestial Mechanics* (Grundlehren der mathematischen Wissenschaften). Springer, Berlin, 1971, ISBN: 978-3-642-65029-1.

[5] J. Waldvogel, "Quaternions and the perturbed Kepler problem," *Celestial Mechanics and Dynamical Astronomy*, vol. 95, no. 1, pp. 201–212, May 2006, ISSN: 1572-9478. DOI: 10.1007/s10569-005-5663-7.

[6] A. Masat, M. Romano, and C. Colombo, "Kustaanheimo–Stiefel Variables for Planetary Protection Compliance Analysis," *Journal of Guidance, Control, and Dynamics*, vol. 45, no. 7, pp. 1286–1298, 2022. DOI: 10.2514/1.G006255.

[7] *CUDAjectory documentation*, Last access: May 2023. [Online]. Available: https://ad.space-codev.org/cudajectory/.

[8] M. Geda, R. Noomen, and F. Renk, "Massive Parallelization of Trajectory Propagations using GPUs," Master's thesis, Delft University of Technology, 2019. [Online]. Available: http://resolver.tudelft.nl/uuid:1db3f2d1-c2bb-4188-bd1e-dac67bfd9dab.

[9] C. H. Acton, "Ancillary data services of NASA's Navigation and Ancillary Information Facility," *Planetary and Space Science*, vol. 44, no. 1, pp. 65–70, Jan. 1996, ISSN: 0032-0633. DOI: 10.1016/0032-0633(95)00107-7.

[10] A. F. Inno, L. Bucci, A. Masat, and C. Colombo, "Orbital interference, planetary close-approaches detection and memory handling on GPUs," M.S. thesis, Politecnico di Milano, 2022. [Online]. Available: http://hdl.handle.net/10589/188499.

[11] *PyBind Python/C++ interface*, Last access: May 2023. [Online]. Available: https://github.com/pybind/pybind11.

[12] *Numpy Python package*, Last access: May 2023. [Online]. Available: https://numpy.org/.

[13] *JSON file format*, Last access: May 2023. [Online]. Available: https://www.json.org/json-en.html.

[14] *YML file format*, Last access: May 2023. [Online]. Available: https://yaml.org/.

[15] European Space Agency (ESA), "Jupiter ICy moons Explorer Exploring the emergence of habitable worlds around gas giants. Definition Study Report.," Tech. Rep. 1.0, 2014, p. 128. [Online]. Available: https://sci.esa.int/web/juice/-/54994-juice-definition-study-report.

[16] A. Boutonnet and A. Rocchi, "Taxonomy of Earth Impacting Trajectories: the Case of JUICE Launcher Upper Stage," in *AIAA SciTech 2022 Forum*, San Diego (CA), USA, 2022. DOI: 10.2514/6.2022-2460.

[17] NVIDIA corporation, *CUDA C++ Programming Guide*, Nov. 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.