A. Masat, C. Colombo, A. Boutonnet
Surfing Chaotic Perturbations in Interplanetary Multi-Flyby Trajectories: Augmented Picard-Chebyshev Integration for Parallel and GPU Computing Architectures
in: AIAA Scitech 2022 Forum, AIAA, 2022, ISBN: 9781624106316, p. 1-14, AIAA 2022-1275
[AIAA Scitech 2022 Forum, San Diego, CA, USA & Virtual Conference, 3-7 Jan. 2022]
doi:10.2514/6.2022-1275

Surfing Chaotic Perturbations in Interplanetary Multi-Flyby Trajectories: Augmented Picard-Chebyshev Integration for Parallel and GPU Computing Architectures

Alessandro Masat * and Camilla Colombo†
Politecnico di Milano, Milano, Italy, 20156

Arnaud Boutonnet ‡
European Space Agency (ESA/ESOC), Darmstadt, Germany, 64293

The computational intensity of the trajectory design problem severely affects the development time of any space mission, both in its preliminary phase and in the consequent optimization. This paper presents a formulation of the design problem that can account for any force source in the dynamical model through efficient Picard-Chebyshev numerical simulations. A two-level augmentation of the integration scheme is proposed, to run an arbitrary number of simulations within the same algorithm call, fully exploiting high performance and GPU computing facilities. The performances obtained with implementation in C and NVIDIA® CUDA® programming languages are shown, highlighting possible use cases and paradigms for the efficient use of GPU computing architectures.

I. Introduction

Complex trajectory solutions have become a standard choice for interplanetary missions, and often include several gravity assist maneuvers to reach remote space regions with limited fuel consumption. Two late missions, the ESA/NASA mission Solar Orbiter [1] and the ESA mission JUICE [2], are meaningful ongoing cases. The former first features preparatory flybys of Earth and Venus, then exploits several resonant encounters with Venus to raise the spacecraft inclination over the ecliptic and observe the Sun's polar regions. The latter requires multiple flybys to reach Jupiter, and then repeatedly swings by different Jupiter's moons to fulfill its scientific observational objectives. Both missions would not be practically feasible without all the designed flybys, as every saved kilogram of fuel means more mass available to board scientific equipment.

*PhD Candidate, Department of Aerospace Science and Technology, Via G. La Masa 34, 20156, Milano, Italy, alessandro.masat@polimi.it
†Associate Professor, Department of Aerospace Science and Technology, Via G. La Masa 34, 20156, Milano, Italy, camilla.colombo@polimi.it
‡PhD, Mission Analyst, Mission Analysis Section, Robert-Bosch Straße 5, 64293, Darmstadt, Germany, arnaud.boutonnet@esa.int

The common framework where high-energy multi-flyby trajectories are designed is the patched conics approximation, whose simple but effective model allows to perform a consistent preliminary mission analysis. Feasible gravity assist maneuvers are identified and the consequent optimization of the initial trajectory guess can be performed, by refining the preliminary orbital parameters and/or adding artificial corrections to meet the operational requirements and fit the full body dynamics.

Nevertheless, the real-life environment features more complex physical phenomena, whose perturbing effects may have a significant impact on the nominal mission. An increasing deviation from the designed trajectory is experienced, and would violate the mission operations if several artificial correction actions were not performed. Furthermore, the patched conics approximation becomes too broad in the vicinity of the sphere of influence boundaries: this mission phase is extremely sensitive on displacements from the nominal trajectory, that if not properly controlled or mitigated can be amplified of several orders of magnitude by the fast dynamics of the close approach.

The proposed work defines a systematic framework where space-time continuous trajectories are designed, accounting for an arbitrarily complex physical model of the orbital dynamics and including planetary flybys. Based on the results of a previous work [3], the whole trajectory is split into several interplanetary legs delimited by flyby events, designed directly with the completely perturbed physical model in a dynamic programming-like backward recursion logic, generalized in Figure 1.

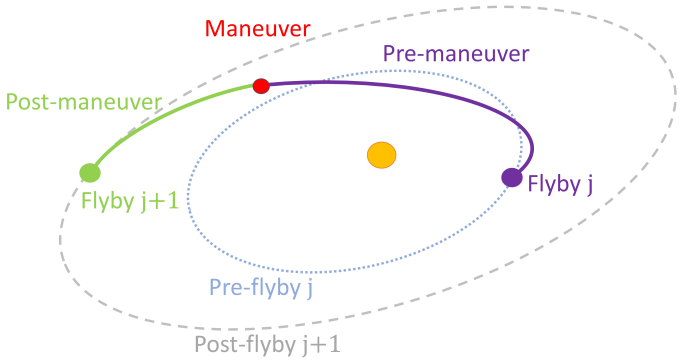


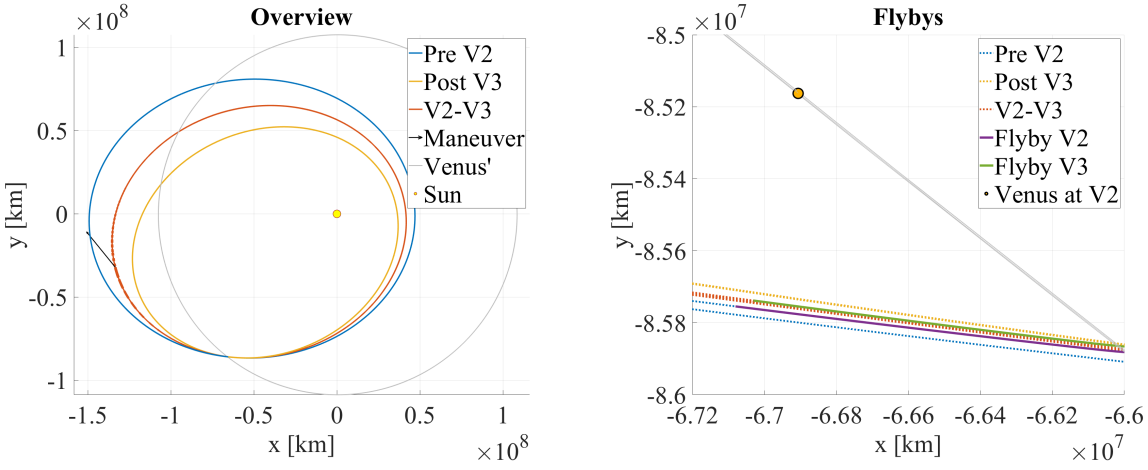
Fig. 1 Orbital scheme of the phase from flyby j to flyby $j+1$.

The j-th gravity assist maneuver is defined by the exit condition from the planetary sphere of influence, found minimizing the artificial correction effort required to reach flyby $j+1$, already determined from either boundary conditions (e.g end-of-life) or a previous step of this design algorithm. The back-integration of flyby j to the sphere of influence entrance provides a new target state for the design of flyby $j-1$. Proceeding backwards mitigates the

combinatorial nature of the problem if the j-th flyby body is determined within the recursion: having already defined the $j+1$-th flyby allows not to deal with branched cases, and preserves the designed trajectory thereafter regardless the choices made for flyby j.

Numerical simulations are required to use the completely perturbed dynamical model. Two separate concepts are brought together in [3] to achieve a feasible computational load for the single leg optimization: the b-plane description of close approaches [4] and the Picard-Chebyshev (PC) integration method [5], implemented in its modified version [6]. The former provides an analytic solution to map the post-flyby orbital parameters at the boundaries of the sphere of influence, used to generate a suitable initial trajectory guess, the latter as the integration scheme to obtain the perturbed path. The b-plane formalism was also used for the description of the exit state to be optimized, allowing to bound the search space only to a narrow region nearby the pruning unperturbed orbit.

The fixed time nodes of PC allow to read ephemerides data only once, task known to be the most expensive part of numerical simulations accounting for the effects of other bodies in the Solar System [7]. In the optimization process, different trajectory guesses computed on a common set of time nodes would then require the same ephemerides data to be sampled again only once, drastically reducing the computational burden required by the repeated numerical simulations. A Matlab® sequential implementation of the proposed optimization required only a few minutes to reach the optimal solution for a Solar Orbiter-like first resonant phase with Venus shown in Figures 2a and 2b, surfing the perturbations from the N bodies and general relativity with just a few meters per second of artificial $\Delta\mathbf{v}$ needed.



(a) Deep space correction maneuver and overview. **(b) Zoom over Venus' flybys V2 and V3.**

Fig. 2 Solar Orbiter's continuous first resonant phase with Venus.

Recent development on PC introduced the second order technique [8] and adaptive capabilities [9]. The integrator was applied to Earth orbits for the design of low-thrust trajectories [10, 11], the solution of the perturbed Lambert problem [12, 13], and the integration of the Circular Restricted Three-Body problem via differential corrections [14]. The optimal number of Chebyshev nodes per period was found to be in the order of dozens to hundreds [9], and numerical instabilties arise if multiple orbital revolutions are integrated in a single run [5] even with increasing number of nodes. Despite the parallelization possibilities, these last two aspects become serious disadvantages for the massively parallel/GPU implementation of the integration scheme.

The proposed work extends the results obtained in [3] discussing the parallelization capabilities for the presented optimization of single trajectory legs in deep detail. Within a single run, the common time nodes of all the trajectory arcs allow to rework the iterative refinement process of PC. Instead of the six-dimensional Cartesian state of the single trajectory, the propagation is performed for a two-level augmented state, made by a properly sorted collection of the states of different trajectories at the same time node. Regardless the sorting choices of the single states within the augmented system, the fundamental mathematical structure of the PC process remains unaltered, without the need of re-defining the method's steps, coefficients and matrices. The integration of the newest developments of the method, particularly the error feedback and the adaptive capabilities [9], would therefore be applicable to the augmented version as well. The two different augmentation levels allow to preserve the fine grain flexibility that, if feasible, simulating each single trajectory alone would have, without a too large sacrifice in computational efficiency. The augmented formulation enhances the parallel nature of the integration algorithm, despite keeping short trajectory legs to avoid the known multi-revolution instability issues [5]. The single leg design cost drops to the order of seconds for the case analyzed in [3] (Figures 2a and 2b).

The paper is outlined as follows: Section II presents the features of the PC numerical scheme and Section III introduces the modifications required to perform the scheme augmentation, whereas the fundamental concepts of GPU computing and of the NVIDIA® CUDA® languages are given in Section IV. Finally, implementations and performances are discussed in Section V.

II. Picard-Chebyshev integration

Picard iterations [15] are a method that can be used to obtain an approximation of the solution of initial/boundary value problems. Denoting the state of dimension n with $\mathbf{x}$, the independent variable with t, the initial/boundary condition

with $\mathbf{x}_0$ and the dynamics function with $\mathbf{f}(\mathbf{x}, t)$, the problem is defined as:

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, t), \qquad \mathbf{x}_0 = \mathbf{x}(t_0) \tag{1}$$

Starting from an initial approximation $\mathbf{x}^{(0)}(t)$ of the actual solution $\mathbf{x}(t)$ in the interval $[t_0, t]$ of the initial/boundary value problem presented in Equation (1), the i-th Picard iteration improves the previous approximation $\mathbf{x}^{(i-1)}(t)$ of $\mathbf{x}(t)$ with $\mathbf{x}^{(i)}(t)$ as in [15]:

$$\mathbf{x}^{(i)}(t) = \mathbf{x}^{(0)}(t) + \int_{t_0}^{t} \mathbf{f}\big(\mathbf{x}^{(i-1)}(s), s\big) ds \tag{2}$$

The method converges for a good enough initial approximation $\mathbf{x}^{(0)}(t)$ and for $i \longrightarrow +\infty$ [15].

In the analytical Picard iteration context, performing more than one iteration is in general hard. The increasingly complex expressions for $\mathbf{x}^{(i)}(t)$ make it difficult to retrieve closed form solutions after the first 2-3 steps [16]. At the same time, numerically computing the integral functions by quadrature might not suffice in accuracy, as only the first few iterations in general improve the function approximation. In the attempt to develop parallelizable routines for the integration of the dynamical motion, the PC method was built combining the Picard iterations with the Chebyshev polynomial approximation [17]. A possible derivation of the method that follows the work of Fukushima [5] is reported in Appendix, and can be summarized in three steps:

1) Select a good enough initial guess $\mathbf{x}^{(0)}(t)$.

2) Approximate $\mathbf{f}(\mathbf{x}, t)$ and $\mathbf{x}^{(0)}(t)$ with their Chebyshev polynomial expansion.

3) Perform a Picard iteration to update the coefficients of the interpolating Chebyshev polynomials.

The Picard iterations halt when the stopping conditions are met, based on the maximum difference between two consecutive iterations dropping below some user-specified tolerance.

The so defined method allows to easily perform several more Picard iterations than the analytical case. The involved expressions remains always of the same type, i.e. the Chebyshev polynomials, thus delegating the function approximation to a finer level. Furthermore, few iterations suffice to drop below a low tolerance if the real solution $\mathbf{x}(t)$ differs from the initial guess $\mathbf{x}^{(0)}(t)$ only because of small perturbations [15]. Starting from the unperturbed Keplerian solution for the generic weakly perturbed two body problem, a relatively fast convergence of the method is ensured [5].

A. Matrix form for vectorized and parallel computation

The method is suitable for parallel or vector implementation, indeed Fukushima also proposed a vectorized version [18]. More recent works over this technique by Bai and Junkins developed the modified PC method [6] and a CUDA® implementation for NVIDIA® GPUs [16]. For compactness and to better highlight the parallelization possibilities, the method is presented following the matrix formulation by Koblick et al [19].

For N Chebyshev nodes and the integration interval $[t_0, t_{N-1}]$, the independent variable t is sampled for $j = 0, 1, ..., N-1$ up-front as

$$t_j = \omega_2 \tau_j + \omega_1 \tag{3}$$

with

$$\tau_j = -\cos\left(\frac{j\pi}{N-1}\right), \quad \omega_1 = \frac{t_{N-1} + t_0}{2}, \quad \omega_2 = \frac{t_{N-1} - t_0}{2} \tag{4}$$

The values of the Chebyshev polynomials and their derivatives at the selected N nodes do not directly enter the Picard iterations, but require some manipulation to obtain the constant matrices $\mathbf{C}$ and $\mathbf{A}$, which can be computed up-front as well. $\mathbf{C}$ is $N \times N$ and its elements are defined as [19]

$$\begin{aligned} \mathbf{C}_{j+1,1} &= \frac{1}{2} T_0(\tau_j), \quad j = 0, ..., N-1 \\ \mathbf{C}_{j+1,k+1} &= T_k(\tau_j), \qquad j = 0, ..., N-1 \quad \text{and} \quad k = 1, ..., N-1 \end{aligned} \tag{5}$$

whereas $\mathbf{A}$ is $N-1 \times N$, with elements

$$\begin{aligned} \mathbf{A}_{k+1,j+1} &= \frac{T_k(\tau_j) - T_{k+2}(\tau_j)}{(N-1)(j+1)}, \quad j = 0, ..., N-2, \;\; k = 1, ..., N-3 \\ \mathbf{A}_{k+1,j+1} &= \frac{T_k(\tau_j) - T_{k+2}(\tau_j)}{2(N-1)(j+1)}, \quad j = 0, ..., N-2, \;\; k = 0, N-2 \\ \mathbf{A}_{k+1,N} &= \frac{T_k(\tau_{N-1})}{(N-1)^2}, \qquad\qquad k = 1, ..., N-3 \\ \mathbf{A}_{k+1,N} &= \frac{T_k(\tau_{N-1})}{2(N-1)^2}, \qquad\qquad k = 0, \; N-2 \end{aligned} \tag{6}$$

where $T_k(\tau) = \cos(k \arccos(\tau))$ is the Chebyshev polynomial of degree k [5]. Another constant quantity that is involved

in the iteration process is the row matrix $\mathbf{S}$ of dimension $1 \times N-1$, whose elements are [19]

$$\mathbf{S}_k = 2(-1)^{k+1}, \qquad k = 1, ..., N-1 \tag{7}$$

Finally, given the n-dimensional sampled states $\mathbf{y}^{(i-1)}(t_j) = \mathbf{y}_j^{(i-1)}, \quad j = 0, ..., N$ as a matrix $\mathbf{y}^{(i-1)}$ of dimension $N \times n$ computed at the Picard iteration $i-1$, the whole process can be summarized in three sequential steps to obtain the states at the iteration i. The first one collects the evaluations of the dynamics function $\mathbf{f}$ in the $N \times n$ force matrix $\mathbf{F}$ [19]:

$$\mathbf{F}_{j+1}^{(i)} = \omega_2 \ \mathbf{f}\big(\mathbf{y}_{j+1}^{(i-1)}, t_j\big), \quad j = 0, ..., N-1 \tag{8}$$

each of which can be performed in parallel, as well as all the upcoming matrix elementary operations.

Secondly, the $N \times n$ matrix $\mathbf{B}$ is obtained by rows as [19]

$$\mathbf{B}_1 = \mathbf{SAF} + 2\mathbf{y}_0, \quad \mathbf{B}_j = \mathbf{AF}, \quad j = 2, ..., N \tag{9}$$

Note that the boundary values $\mathbf{y}_0$ remain constant for all the Picard iterations. Third and last, the $N \times n$ matrix of the state guesses $\mathbf{y}^{(i)}$ for the i-th Picard iteration is

$$\mathbf{y}^{(i)} = \mathbf{CB} \tag{10}$$

The iteration process stops when the maximum state difference between two consecutive Picard iterations $\mathbf{y}^{(i)}$ and $\mathbf{y}^{(i-1)}$ drops below a specified relative or absolute tolerance, upon user's choice.

As already underlined by Fukushima [5] and Bai and Junkins [6, 16], despite the proved theoretical convergence, large integration spans may lead to numerical instabilities, due to the cumulation of round-off errors even with large N as multiple orbital revolutions take place. Fukushima [5] suggests a piece-wise approach as a workaround, which was implemented in [3] and uses the modified PC method to integrate orbit by orbit in sequence* until the end of the span.

The core steps of the proposed algorithm follow the presented scheme [6, 16], together with the automatic generation of the Keplerian initial guess spanning one nominal orbital period. The proposed implementation interfaces with SPICE ephemerides [20] considering them a parameter of the Picard iteration process, to be sampled in advance on the time

*The proposed implementation automatically handles either forward or backward integration.

nodes and to be shared by all the iterations and the states being simulated.

III. Two-level system augmentation

A. One-level augmentation

Instead of the evolution of the sole trajectory determined by the initial condition $\mathbf{y}_0$, the system being integrated can be re-written so that M different trajectories sampled on the same N time nodes can be processed within a unique iterative process. At the iteration i, the matrix $\mathbf{Y}^{(i)}$ collects all the samples of all the trajectories, and its j-th row is related to the j-th time sample of the m-th trajectory by:

$$\mathbf{Y}_j^{(i)} = \left[\mathbf{y}_{j,1}^{(i)} \;\; \cdots \;\; \mathbf{y}_{j,m}^{(i)} \;\; \cdots \;\; \mathbf{y}_{j,M}^{(i)}\right], \quad j = 1, ..., N \tag{11}$$

and similarly for the dynamics function evaluations collected in the matrix $\mathbf{F}^{(i)}$:

$$\mathbf{F}_j^{(i)} = \left[\mathbf{F}_{j,1}^{(i)} \;\; \cdots \;\; \mathbf{F}_{j,m}^{(i)} \;\; \cdots \;\; \mathbf{F}_{j,M}^{(i)}\right], \quad j = 1, ..., N \tag{12}$$

whose elements are still computed per sample:

$$\mathbf{F}_{j,m}^{(i)} = \omega_2 \;\; \mathbf{f}\big(\mathbf{y}_{j,m}^{(i-1)}, t_{j-1}\big), \quad j = 1, ..., N \tag{13}$$

In principle, building the augmented system only requires to define $\mathbf{Y}^{(i)}$ by stacking the different M trajectory matrices along the columns, and similarly $\mathbf{F}^{(i)}$ undergoes the exact same modification. The structure of the PC iterations remains unchanged and features the steps:

1) Evaluate the dynamics function for all the N states of all the M trajectories with $\mathbf{F}_{j,m}^{(i)} = \omega_2 \;\; \mathbf{f}\big(\mathbf{y}_{j,m}^{(i-1)}, t_{j-1}\big)$.
2) Perform the matrix operations $\mathbf{B}_1 = \mathbf{SAF} + 2\mathbf{Y}_0$ and $\mathbf{B}_j = \mathbf{AF}$, for $j = 2, ..., N$.
3) Update the guesses for all the M trajectories with $\mathbf{Y}^{(i)} = \mathbf{CB}$.

B. Two-level augmentation

The stack-along-column rule can be applied again, this time collecting in one single matrix P groups of different M_p trajectories each. The augmented matrix $\mathbf{Y}^{(i)}$ is now built as

$$\mathbf{Y}_j^{(i)} = \left[\mathbf{Y}_{j,1}^{(i)} \ \cdots \ \mathbf{Y}_{j,p}^{(i)} \ \cdots \ \mathbf{Y}_{j,P}^{(i)} \right], \quad j = 1, ..., N \tag{14}$$

with

$$\mathbf{Y}_{j,p}^{(i)} = \left[\mathbf{y}_{j,p,1}^{(i)} \ \cdots \ \mathbf{y}_{j,p,m}^{(i)} \ \cdots \ \mathbf{y}_{j,p,M_p}^{(i)} \right], \quad j = 1, ..., N \tag{15}$$

In principle, infinite augmentation levels could be built relying on the same logic, and none of them would require modifications in the core PC algorithm structure. Nevertheless, a re-definition of the iteration error can be helpful for practical purposes, since the augmentation advantage is purely computational.

Two strategies can be addressed:

1) A traditional error definition, that treats the trajectory samples as if they were part of a unique system, whose maximum will be compared against the iteration stopping condition.
2) A more flexible per-block error definition, that treats the different trajectory blocks as independent, for which the augmentation has then a sole computational purpose.

Both the approaches have advantages and drawbacks. The former would allow a simpler implementation and is inevitably computationally more efficient than the latter, because of the reduced overhead compared to maintaining the group split. However, dissimilar trajectories requiring a significantly different number of iterations would keep the computational resources busy for already converged blocks, while the per-block definition would allow far more flexibility on this regard.

This work uses the two augmentation level to build an hybrid approach, that treats the outer blocks as independent, and the inner ones as a single system in a more strict sense. In this way, groups of similar trajectories can be considered as unique but separate augmented system, allowing to maximise the integration performances. Note that the two level augmentation also provides a framework to deal with single trajectories within the same high performance computing context, considering them as a group made of only one member.

IV. NVIDIA® CUDA® and GPU computing fundamentals

The features of GPU computing arise from the hardware architecture, which is profoundly different from the traditional compute units. Figure 3 shows a graphical representation of such differences: in summary, more transistors are devoted to pure data processing on GPUs, instead of flow control and memory management as in the CPU case [21].

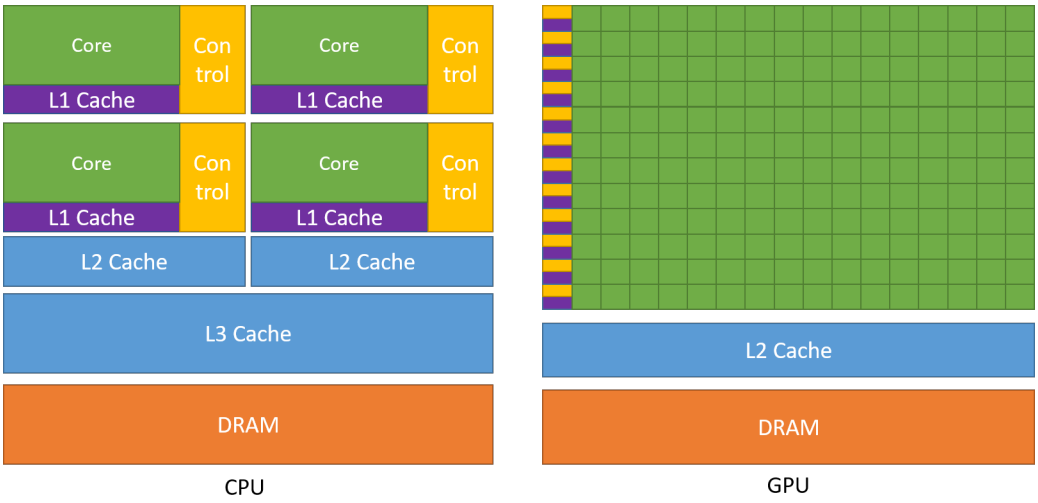


Fig. 3 CPU vs GPU architecture difference graphical scheme. Picture from [21].

The processing units are grouped in blocks, each controlled by the one controller, and processing units in the same block all execute the same instruction issued by the controller. This aspect, together with the normally hundreds to thousands of processing units available in modern graphics cards, makes GPUs prone to implement massive parallelism, although with lower flexibility and higher programming effort compared to CPU applications. Some key concepts are given in the following subsection, a comprehensive view can be found in the CUDA C++ programming guide [21].

A. Main programming paradigms and the CUDA® API

A brief nomenclature is introduced, to then better understand the programming paradigms. In particular:

- The fundamental execution unit is called *thread*.
- Threads can be grouped in *blocks*, and some shared memory is available to threads in the same block.
- The execution of a single instruction is always performed by groups of 32 threads at the same time, called a *warp*, regardless the number of threads in a block. Therefore, blocks with less than 32 threads do not exploit the full hardware resources.
- CPU and GPU are often called *host* and *device* respectively, and complex configurations can be achieved combining

multiple CPUs and GPUs to run the same program. The program flow is always controlled by the host, which also controls the execution of the device.

- A function that is invoked by the host but executed on the device is called *kernel*.

The first difference compared to standard programming is that the device does not have normal access to the the usual compute memory, but the data must be loaded on the dedicated device memory before any kernel run. In a similar manner, the data must be retrieved to the host after the kernel has completed its execution for consequent use. Therefore the program flow always follows:

1) Initialisation on the host.
2) Data movement to the device.
3) Kernel execution.
4) Data retrieval on the host.

Kernels must be programmed in a warp-oriented manner, including the data access by the various threads, and complex functions typically require the programmer to optimize memory access, array element sorting, and cache usage by hand.

CUDA® is a programming language developed and maintained by NVIDIA®, which remarkably simplifies the use of NVIDIA® GPUs in computer programs. It is built as a C++ extension, with a set of keywords and API functions that allow programmers to build their own kernels and control the device execution flow. A set of optimized libraries is also available, for instance the basic linear algebra cuBLAS® functions worth mentioning for the purposes of this work [21].

B. Concurrency and advanced features

The host-device duality and cooperation exposes other programming possibilities than the simple acceleration of intensive portions of the code. In general, kernel calls are asynchronous with respect to the host, which allows the host to process other tasks while the kernel is still executing. Furthermore, modern GPUs can manage at the same time two saturated memory transfers (one per direction, host to device and device to host) while saturating also its computing units for one or more concurrent kernel execution [21].

CUDA® allows to enqueue a series of sequentially dependent device function calls with the use of *streams*: for instance, an application may require some data to be transferred to the device before the execution of a custom kernel, which must be completed before calling a cuBLAS® function, at the end of which the processed data should be

transferred back to the host. All it takes is assigning the sequentially dependent device function calls to the same stream. Multiple streams can be created and used at the same time, the obtained behaviour would be analogous to the batch job submission to supercomputing facilities. CUDA® guarantees the synchronization within the same stream, different streams must instead synchronized by hand. The compiler will typically schedule executions and memory transfers so that the GPU use is maximized, superposing different device function calls from separate streams [21].

V. Integration implementation and performances

The Matlab®† implementation proposed in [3] was re-run on a single core of a local workstation equipped with an Intel® Core™ i7-7700 CPU (3.60 GHz), running Windows® 10 Pro. The algorithm converged running 13509 PC integrations in 506.3 seconds, to the residual $\Delta\mathbf{r}^*$ and impulsive action $\Delta\mathbf{v}^*$ for the required maneuver presented in Table 1. To better detail the embedded PC integrations, they all feature 200 Chebyshev nodes spanning the arc connecting the flyby exit and the maneuver point, long about 0.87 orbital periods.

Table 1 Optimization results, in terms of position difference residual $\Delta\mathbf{r}^*$ and correction effort $\Delta\mathbf{v}^*$ at the maneuvering time corresponding to the apocenter of the post-flyby orbit.

$\Delta\mathbf{r}^*$ [m]			$\Delta\mathbf{v}^*$ [m/s]		
x	y	z	x	y	z
−0.52	−0.52	−1.19	−1.28	1.57	0.22

To build a common framework for the pure algorithm performance evaluations, the same initial conditions that led to the results of Table 1 are re-run using a C and a CUDA implementation of the PC integration. The setup considered as benchmark is the execution of the 13509 independent runs with the C implementation of the algorithm, both completely sequential and parallelized with OpenMP [22]. The matrix operations featured in the PC iterations are performed using the OpenBLAS library [23–25]. All the presented runs of the C algorithm have been executed on a machine running Ubuntu Linux 20.04 equipped with 40 physical / 80 logical cores of the type Intel® Xeon™ CPU E5-4620 V4 running at 2.1 GHz, with varying number of OpenMP threads and the "`o3`" `gcc` compiler optimization enabled. Because of the physical machine where the GPU was available, the CUDA™ code has been run on the same workstation of the Matlab® optimal solution computation. In this case a four core OpenMP® parallelization on a Intel® Core™ i7-7700 CPU (3.60 GHz), is combined for concurrent executions with a NVIDIA® GTX 1050 (1.3GHz) graphics card‡. Finally, in all the

†Following the update to Matlab® R2021b higher runtimes have been experienced for the same final results, although not affecting the presented discussion since the performance analysis is only made on the C and the CUDA algorithm versions.

‡This is a 2016 low-end gaming card model, whose design purpose is far from the double precision computing of this work. Modern

cases that will be presented, the relative error among the different implementations for corresponding initial conditions always falls below the specified PC relative tolerance (10^{-12}).

The basic workflow of the independent PC runs is given in the block-scheme of Figure 4. It is clear that the only parallelization possibilities, for high numbers of trajectories, apply at the highest level, inevitably introducing a considerable overhead for both the inner sequential execution and the still parallelizable inner functions. In fact, all the per-trajectory steps of the PC process would still be parallelizable algorithms per se.

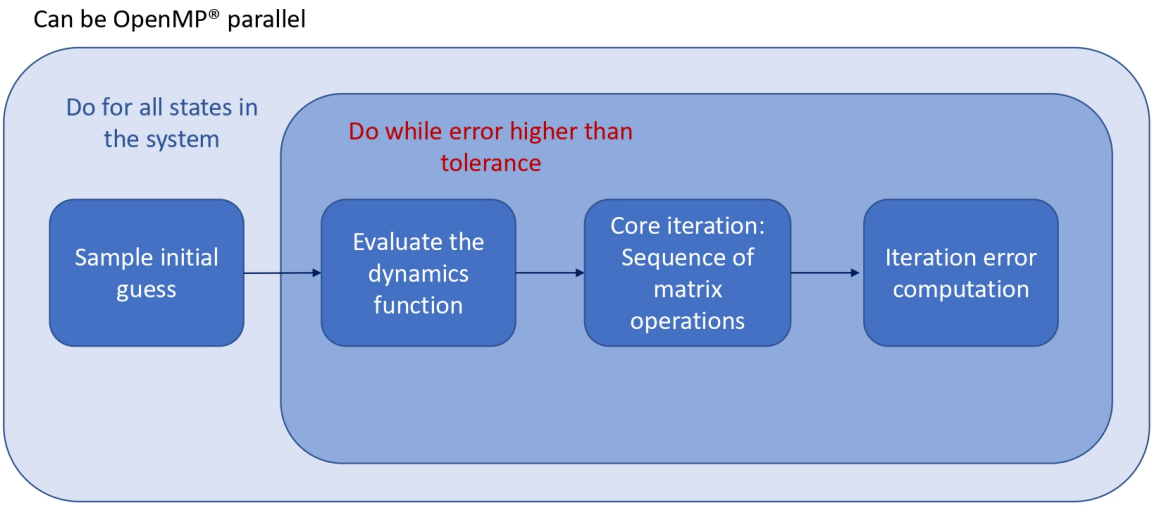


Fig. 4 Standard PC workflow.

A. Sequential Augmented PC workflow

Although not yet introducing the parallelization, the implementation of the augmented PC integration follows a one-level augmentation only, to highlight the pipeline benefits in terms of overhead that this implementation introduces. A block-scheme representation of the augmented PC integration workflow is given in Figure 5. The conceptual change, from the PC iteration viewpoint, is only the initial sampling in a single array containing all the state vectors of all the trajectories of the augmented system.

gaming/professional cards could run up to 60 times faster, data center cards up to 400-500 times faster than this model for the presented application.

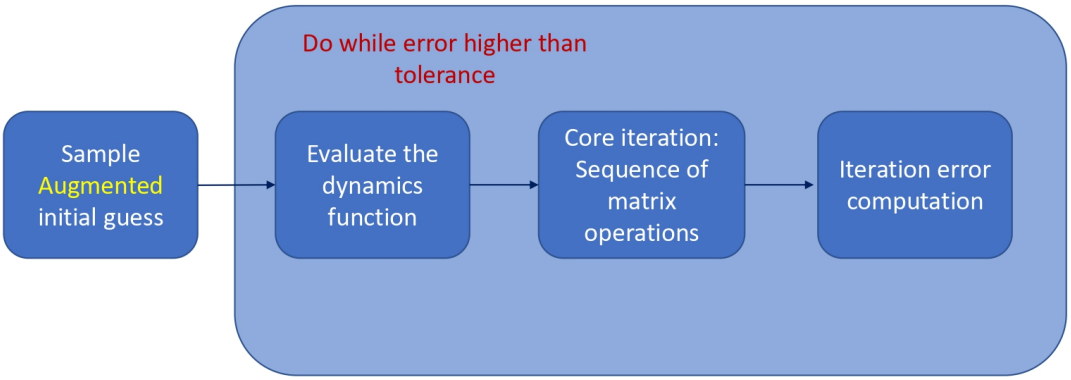


Fig. 5 Augmented PC workflow.

B. OpenMP® parallelized Augmented PC workflow

The parallelization of the augmented system integration, whose block-scheme representation is given in Figure 6, becomes fine-grained. It acts directly on the single state vectors for the dynamics function evaluation and on the elementary products of the matrix operations, moreover already implemented by the OpenBLAS developers [23]. In addition, reduction operations can be made through OpenMP® for a cooperated and parallel search of the maximum error, introducing further benefits to the algorithm runtime.

C. CUDA® Augmented PC workflow

The block-scheme representation of the CUDA® algorithm is given in Figure 7. Here the two-level augmentation is exploited, assigning one higher level augmented system to each CUDA® stream and using the thread-based parallelism on the lower level augmented systems.

The principal benefit of this implementation is the cooperation between host and device for the overall execution, with as many operations as possible executed concurrently. Each lower level augmented system is initially sampled by the host and then moved to the device. The CUDA® stream management API allows to overlap the host sampling of the next higher level augmented systems with memory transfers and kernel executions of the already launched ones. Similarly, the very last step of the PC iteration requires to retrieve the computed iteration error for each stream from

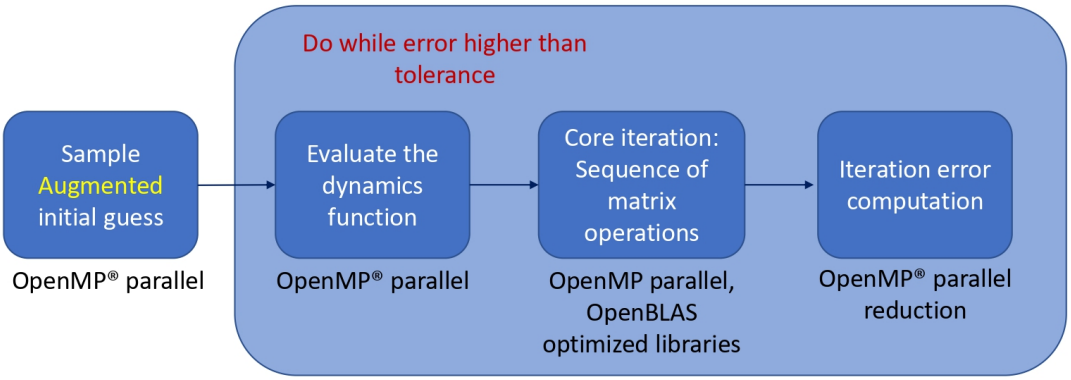


Fig. 6 Augmented and OpenMP® parallelized PC workflow.

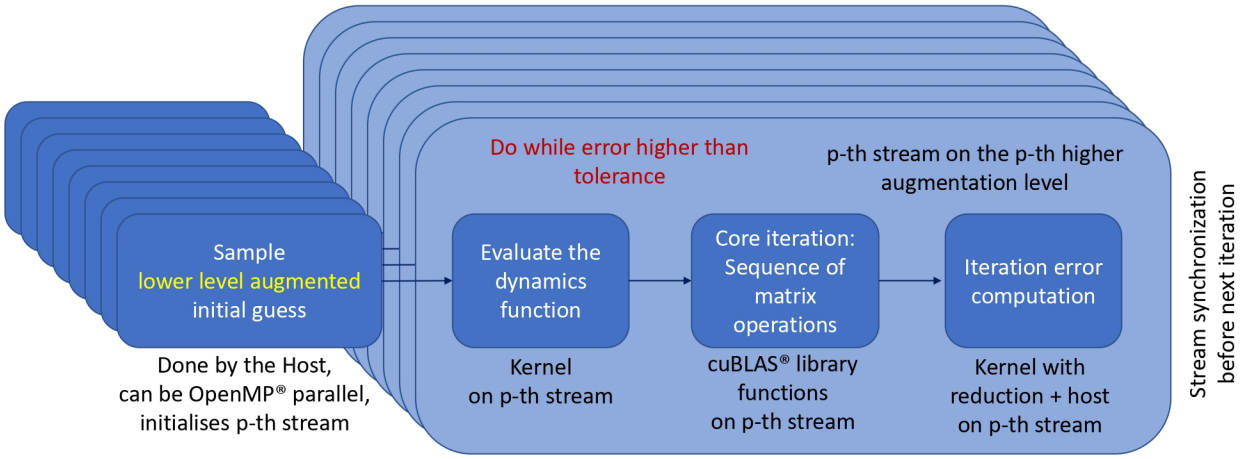


Fig. 7 Augmented CUDA® PC workflow.

the device to the host for the control of the while loop, which is also subject to the stream concurrency benefits. A stream synchronization at the end of each while loop iteration is necessary to achieve the overlapping behaviour of all the streams, because running independent loops for each higher level augmented system would make their execution completely sequential.

The warp-centric programming model of CUDA® kernels requires a small modification on the lower level augmented

system definition. Contiguous array elements should be of the same component type (i.e. contiguous x coordinates, then contiguous y coordinates, and so on), instead of storing state vector by state vector. This aspect might seem an implementation detail, however it is fundamental for the kernel not to run even slower than a sequential fully CPU code. This is because the memory access in CUDA® is optimized if all the threads work on contiguous array elements [21], whose latency would be otherwise too high to be hidden even by intensive parallelized device computations. The just discussed modification has no effect on the overall algorithm structure, all it requires is the dynamics and error kernels to be implemented following this array element logic.

D. Performance comparison

Table 2 shows the difference between the non-parallelized runtimes. The improved efficiency of the augmented integration can be immediately seen even in this sequential case with the augmented system running 23.87% faster, because of the minimized overhead experienced by sharing the outer while loop. Note that in the considered application the various trajectories all require 41 or 42 PC iterations, making it negligible to keep running trajectories even if their PC process has already converged.

Table 2 Sequential runtimes for the independent runs and the augmented system executions.

Case	Runtime [s]
Independent runs	245.02
Augmented system	186.54

The scalability properties of the integration of independent trajectories and the augmented system are studied on the C implementations, that is assessing how well the execution of the two programs increases with increasing number of OpenMP® threads. Figure 8 shows that the augmented system features excellent scalability properties, for a runtime that keeps decreasing for increasing number of OpenMP® threads. On the contrary, the integration of independent trajectories experiences even higher runtimes after a certain point. This happens because the parallelization itself introduces some overhead to the overall program execution, which in this case is not any more made up for by the increased number of threads.

Figure 9 shows the achieved speedup, defined as the ratio between the sequential runtime and the parallel runtime, varying the number of OpenMP® threads. It provides a measure of how parallelizable the algorithm is. The augmented system integration shows once again excellent scalability properties even at high number of threads, suggesting the

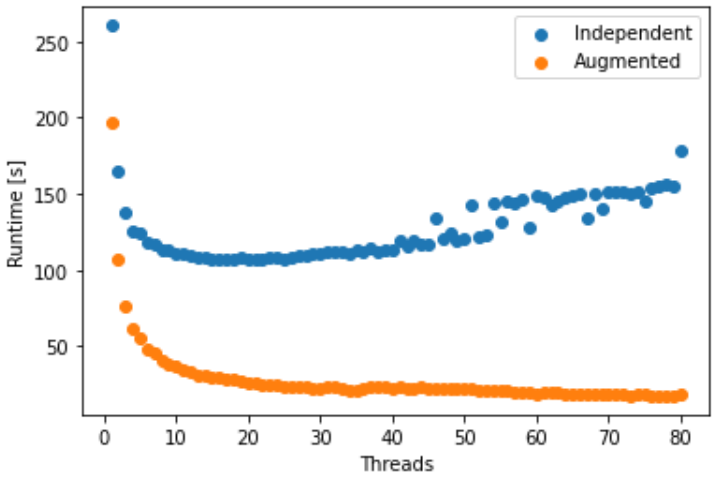


Fig. 8 Augmented system and independent integrations C code runtime comparison with OpenMP® parallelization and varying number of threads.

feasibility of the GPU acceleration even without having assessed the performances of the CUDA® implementation yet.

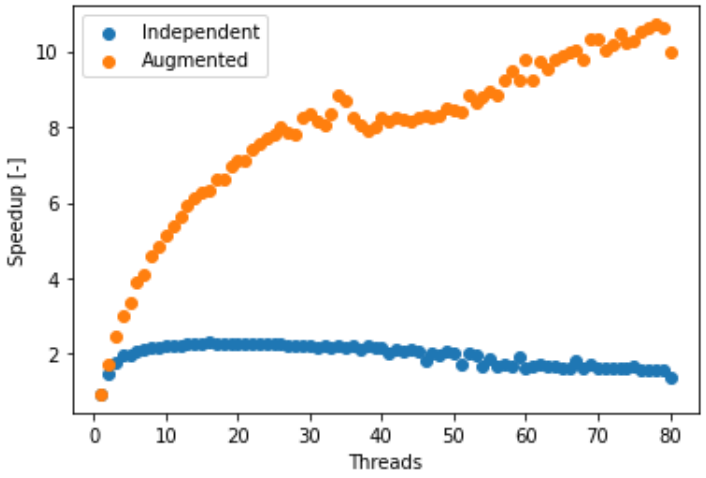


Fig. 9 Augmented system and independent integrations C code speedup comparison with OpenMP® parallelization and varying number of threads.

Finally, the execution of the CUDA® implementation results the faster overall, taking 15.74 seconds. The whole trajectory set has been almost evenly[§] split into 10 streams, although this number does not affect the analysis made in this work. The stream definition guideline should in practice fit the application the propagator would run on, being the unique flexibility degree left by the implementation. However, from the GPU viewpoint, larger kernels always imply a better device exploitation, thus too many streams with too few trajectories each would result in a performance degradation, at the limit of what already observed with the independent integration cases. Profiling the execution of the dynamics kernel

[§]The tenth stream has one less element, because of the floor result of the integer division.

showed a 97.03% saturation of the device compute multiprocessors and minimized memory transactions, suggesting that further acceleration could be achieved only with hardware upgrade. The cuBLAS® functions are already optimized in this direction, whereas the occupancy required by error computation remains negligible.

Table 3 summarizes the runtime results discussed in the previous lines for the different cases, for selected number of cores in the C implementation cases.

Table 3 Sequential runtimes for the independent runs and the augmented system executions.

Case	OS	CPU	Threads	GPU	Runtime [s]
C Independent	Ubuntu® 20.04	E5-4620 V4 @ 2.1 GHz	1	-	245.02
C Augmented	Ubuntu® 20.04	E5-4620 V4 @ 2.1 GHz	1	-	186.54
C Independent	Ubuntu® 20.04	E5-4620 V4 @ 2.1 GHz	8	-	113.09
C Augmented	Ubuntu® 20.04	E5-4620 V4 @ 2.1 GHz	8	-	40.60
C Independent	Ubuntu® 20.04	E5-4620 V4 @ 2.1 GHz	40	-	113.54
C Augmented	Ubuntu® 20.04	E5-4620 V4 @ 2.1 GHz	40	-	22.67
C Independent	Ubuntu® 20.04	E5-4620 V4 @ 2.1 GHz	80	-	178.18
C Augmented	Ubuntu® 20.04	E5-4620 V4 @ 2.1 GHz	80	-	18.70
CUDA® Augmented	Windows® 10 Pro	i7-7700 @ 3.6 GHz	4	GTX 1050	15.74

Figure 10 shows the increasing speedup achieved by the augmented system when compared to the sequential and independent simulations of all the samples. The CUDA® implementation runs more than 15 times faster compared to the baseline case, proving again the suitability of the augmented PC algorithm to high performance and GPU computing facilities.

VI. Conclusion

This work explores the benefits that high performance and GPU computing architectures bring to the short leg orbital propagation of large sets of initial condition. The tested case includes the runs required to optimize a Solar Orbiter-like resonant phase with Venus in the attempt to surf the chaotic perturbed interplanetary environment, proposing a two-level augmentation strategy implemented in the C and the CUDA® programming languages.

Propagating the augmented system always outperforms the single propagations of the independent trajectories, both in the sequential and parallelized case. The augmentation benefits appear in two different aspects, the first being the reduced overhead compared to the repeated independent runs, the second being a finer grain parallelization also exploiting optimized libraries.

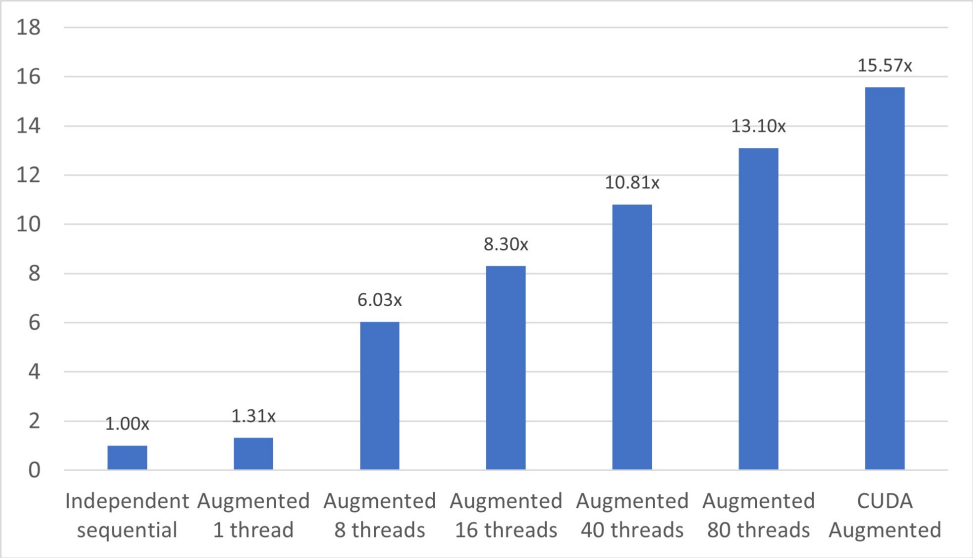


Fig. 10 Speedup comparison among C and CUDA® implementations.

The approach scalability allows its implementation on GPU architectures, with low end and old graphics card already capable of matching the performance of a common-sized cluster node. The proposed algorithm detaches from the trajectory optimization application it was built on, being a completely general propagator per se. Any application requiring the propagation of large sets of initial conditions could benefit of the high computational efficiency this algorithm provides, not necessarily requiring any more the use of supercomputing facilities in favour of use of common gaming graphics cards.

Acknowledgments

The research leading to these results has received funding from the European Research Council (ERC) under the European Union's Horizon2020 research and innovation programme as part of project COMPASS (Grant agreement No 679086), `www.compass.polimi.it`.

References

[1] European Space Agency (ESA), "Solar Orbiter Definition Study Report (Red Book)," Tech. Rep. July, 2011. URL `https://sci.esa.int/s/w7yO4P8`.

[2] 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. URL `https://sci.esa.int/web/juice/-/54994-juice-definition-`

study-report.

[3] Masat, A., Romano, M., and Colombo, C., "Combined B-plane and Picard-Chebyshev approach for the continuous design of perturbed interplanetary resonant trajectories," *31st AAS/AIAA Space Flight Mechanics Meeting*, Vol. AAS-21-289, Charlotte, NC, USA, 2021.

[4] J. Opik, E., *Interplanetary Encounters: Close-Range Gravitational Interactions*, Vol. 2, Amsterdam, 1976.

[5] Fukushima, T., "Picard Iteration method, Chebyshev Polynomial Approximation, and Global Numerical Integration of Dynamical Motions," *The Astronomical Journal*, Vol. 113, 1997, pp. 1909–1914. https://doi.org/10.1086/118404.

[6] Bai, X., and Junkins, J. L., "Modified Chebyshev-Picard Iteration Methods for Orbit Propagation," *Journal of the Astronautical Sciences*, Vol. 58, No. 4, 2011, pp. 583–613. https://doi.org/10.1007/BF03321533, URL https://doi.org/10.1007/BF03321533.

[7] Colombo, C., Letizia, F., and Van Der Eynde, J., "SNAPPshot ESA planetary protection compliance verification software Final report V1.0, Technical Report ESA-IPL-POM-MB-LE-2015-315," Tech. rep., University of Southampton, 2016.

[8] Junkins, J. L., Bani Younes, A., Woollands, R. M., and Bai, X., "Picard Iteration, Chebyshev Polynomials and Chebyshev-Picard Methods: Application in Astrodynamics," *Journal of the Astronautical Sciences*, Vol. 60, No. 3, 2013, pp. 623–653. https://doi.org/10.1007/s40295-015-0061-1, URL https://doi.org/10.1007/s40295-015-0061-1.

[9] Woollands, R., and Junkins, J. L., "Nonlinear Differential Equation Solvers via Adaptive Picard-Chebyshev Iteration: Applications in Astrodynamics," *Journal of Guidance, Control, and Dynamics*, Vol. 42, No. 5, 2019, pp. 1007–1022. https://doi.org/10.2514/1.G003318, URL https://doi.org/10.2514/1.G003318.

[10] Koblick, D., Xu, S., Fogel, J., and Shankar, P., "Low Thrust Minimum Time Orbit Transfer Nonlinear Optimization Using Impulse Discretization via the Modified Picard-Chebyshev Method," *Computer Modeling in Engineering & Sciences*, Vol. 111, No. 1, 2016. https://doi.org/10.3970/cmes.2016.111.001, URL https://doi.org/10.3970/cmes.2016.111.001.

[11] Woollands, R., Taheri, E., and Junkins, J. L., "Efficient Computation of Optimal Low Thrust Gravity Perturbed Orbit Transfers," *Journal of the Astronautical Sciences*, Vol. 67, No. 2, 2020, pp. 458–484. https://doi.org/10.1007/s40295-019-00152-9, URL https://doi.org/10.1007/s40295-019-00152-9.

[12] Woollands, R. M., Bani Younes, A., and Junkins, J. L., "New Solutions for the Perturbed Lambert Problem Using Regularization and Picard Iteration," *Journal of Guidance, Control, and Dynamics*, Vol. 38, No. 9, 2015, pp. 1548–1562. https://doi.org/10.2514/1.G001028, URL https://doi.org/10.2514/1.G001028.

[13] Woollands, R. M., Read, J. L., Probe, A. B., and Junkins, J. L., "Multiple Revolution Solutions for the Perturbed Lambert Problem using the Method of Particular Solutions and Picard Iteration," *Journal of the Astronautical Sciences*, Vol. 64, No. 4, 2017, pp. 361–378. https://doi.org/10.1007/s40295-017-0116-6, URL https://doi.org/10.1007/s40295-017-0116-6.

[14] Swenson, T., Woollands, R., Junkins, J., and Lo, M., "Application of Modified Chebyshev Picard Iteration to Differential Correction for Improved Robustness and Computation Time," *Journal of the Astronautical Sciences*, Vol. 64, No. 3, 2017, pp. 267–284. https://doi.org/10.1007/s40295-016-0110-4.

[15] Hairer, E., Wanner, G., and Nørsett, S. P., *Solving Ordinary Differential Equations I*, 2nd ed., Springer Berlin, 1993. https://doi.org/10.1007/978-3-540-78862-1.

[16] Bai, X., and Junkins, J. L., "Solving initial value problems by the Picard-Chebyshev method with NVIDIA GPUs," *Advances in the Astronautical Sciences*, 2010.

[17] Rivlin, T. J., "The Chebyshev Polynomials," *Mathematics of Computation*, Vol. 30, 1976. https://doi.org/10.2307/2005983.

[18] Fukushima, T., "Vector Integration of Dynamical Motions by the Picard-Chebyshev Method," *The Astronomical Journal*, Vol. 113, 1997, p. 2325. https://doi.org/10.1086/118443.

[19] Koblick, D., Poole, M., and Shankar, P., "Parallel high-precision orbit propagation using the Modified Picard-Chebyshev Method," *ASME International Mechanical Engineering Congress and Exposition, Proceedings (IMECE)*, 2012. https://doi.org/10.1115/IMECE2012-87878.

[20] Acton, C. H., "Ancillary data services of NASA's Navigation and Ancillary Information Facility," *Planetary and Space Science*, Vol. 44, No. 1, 1996, pp. 65–70. https://doi.org/10.1016/0032-0633(95)00107-7, URL https://www.sciencedirect.com/science/article/pii/0032063395001077.

[21] NVIDIA corporation, "CUDA C++ Programming Guide," , Nov 2021. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[22] Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., and McDonald, J., *Parallel programming in OpenMP*, Morgan kaufmann, 2001.

[23] "Xianyi/OpenBLAS," , Last access: November 2021. URL https://github.com/xianyi/OpenBLAS#supported-cpus-and-operating-systems.

[24] Wang, Q., Zhang, X., Zhang, Y., and Yi, Q., "AUGEM: Automatically generate high performance Dense Linear Algebra kernels on x86 CPUs," *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12. https://doi.org/10.1145/2503210.2503219.

[25] Xianyi, Z., Qian, W., and Yunquan, Z., "Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor," *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, 2012, pp. 684–691. https://doi.org/10.1109/ICPADS.2012.97.