FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Interdisciplinary Project in Informatics

Efficient Polyhedral Gravity Modeling in Modern C++

Jonas Schuhmacher

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Interdisciplinary Project in Informatics

Efficient Polyhedral Gravity Modeling in Modern C++

Effiziente Modellierung Polyhedraler Schwerkraft in Modernem C++

Author:	Jonas Schuhmacher
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc. and Dr.-Ing. Pablo Gómez
Date:	31.12.2022

I confirm that this interdisciplinary project is my own work and I have documented all sources and material used.

Munich, 31.12.2022 Jonas Schuhmacher

Acknowledgements

I want to thank my two advisors, Fabio and Pablo, for always giving valuable feedback, pushing me to try out new things, and for our biweekly meetings discussing the topic and other amusing stuff. It is always a pleasure.

Abstract

In exploring the solar system, not only the planets and moons are of interest, but also asteroids and comets. These bodies usually have a pretty irregular shape making the determination of their gravity field difficult. However, this knowledge is critical to the successful operation of spacecraft in close proximity to these bodies.

One of the available modeling techniques to solve the problem is the polyhedral gravity model. It is capable of analytically determining the full gravity tensor, including potential, acceleration, and the second derivatives for an arbitrary point P around a homogenous-density polyhedron. Consequently, the only required inputs are the polyhedral mesh and the constant density.

This work implements the analytical solution for the polyhedral gravity model via the line integral approach in an open-source project with extensive accompanying documentation on *readthedocs*. It relies on an efficient and parallelized backbone in `C++ 17`, vectorizing expensive computations. For example, the resulting performant implementation evaluated the full gravity tensor for thousands of points for the asteroid Eros with a mesh consisting of 24235 nodes and 14744 faces in less than a second on a typical system. Further, the implementation's interface is exposed via a simple `Python` module deployed and published on *conda*.

Finally, the implementation has undergone multiple tests in order to be successfully verified. Its results were compared to a cube-shaped body with closed-analytical solutions and publicly available implementations in `MATLAB` and `FORTRAN`.

Zusammenfassung

Typischerweise sind bei der Erforschung unseres Sonnensystems nicht nur Planeten und Monde von Interesse, sondern auch kleinere Körper wie Asteroiden und Kometen. Diese zeichnen sich meistens durch eine recht irreguläre Form aus, was die Bestimmung ihres Gravitationsfeldes nicht unbedingt einfach macht. Jedoch ist das Wissen um das Gravitationsfeld von kritischer Bedeutung für den Erfolg von Operationen in naher Distanz zu diesen Körpern.

Eines der verfügbaren Modelle ist das Polyhedrale Gravitationsmodel. Mit ihm ist es möglich, den vollständigen Schwerkrafttensor, bestehend aus Potential, Beschleunigung und zweiten Ableitungen, für einen beliebigen Punkt P rund um einen Polyeder homogener Dichte analytisch zu bestimmen. Als Eingaben werden dafür lediglich das Mesh des Polyeders sowie dessen konstante Dichte benötigt.

Diese Arbeit implementiert die analytische Lösung des Polyhedralen Gravitationsmodels mittels des Linienintegralen Ansatzes in einem Open-Source Projekt mit einer umfangreichen begleitenden Dokumentation auf *readthedocs*. Das Rückgrat dieser Implementierung besteht hierbei aus einer effizienten, parallelisierten und vektorisierten Implementierung in `C++17`. Als Beispiel für die Performanz zu nennen sei, dass die Berechnung von tausenden von Punkten für das Mesh von Eros, bestehend aus 24235 Knoten und 14744 Flächen, innerhalb von weniger als einer Sekunde finalisiert wurde auf einem typischen System. Zudem wurde das begleitende `Python` Interface der Implementierung als Teil dieser Arbeit auch auf *conda* veröffentlicht.

Abschließend sei zu erwähnen, dass die Implementierung mehreren Tests unterzogen worden ist. So sind ihre Ergebnisse mit existierenden Implementierungen in `FORTRAN` und `MATLAB` sowie der geschlossenen analytischen Lösung für einen würfelförmigen Körper erfolgreich verglichen worden.

Contents

Part I.

Introduction and Background

1. Introduction

Is it a bird? Is it a plane? No, it is a (space) potato. One can only assume what C.G. Witt thought on 13.08.1898 when he discovered Eros[1] depicted in Figure 1.1. However, one can see that Eros' shape is definitely not trivially described but has some commonalities with a potato.

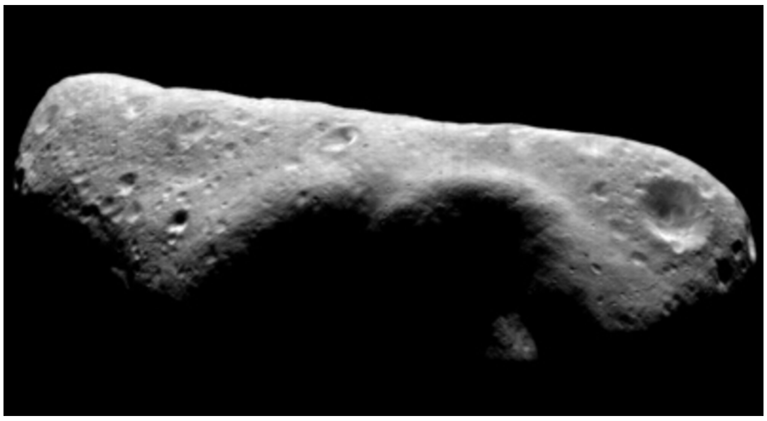


Figure 1.1.: Asteroid (433) Eros - “The space potato”, Source: `https://www.esa.int/ESA_Multimedia/Images/2005/09/33_km_lange_fliegende_Steinkartoffel_der_Asteroid_Eros`, lasted accessed: 12.09.2022

In the last decades, asteroids and comets have become a topic of increased interest, as more and more of them are discovered with the improved observation gear of today [1]. As of now, more than 1.2 million asteroids and more than three thousand comets have been discovered in our solar system.[2] They are worthwhile destinations for missions - like *Rosetta* in 2014 with its lander *Philae* [2] or the upcoming *Hera* mission involving deep-space cubesats for the first time [3] - to push the boundaries of what is technically possible and learn about the history of our solar system. Also, the prospect of asteroid mining is a rewarding goal to get rare and expensive materials that only rarely appear on Earth [4].

However, it is critical that vessels approaching asteroids or comets in close proximity correctly estimate the gravitational forces to complete their missions successfully. Tasks like that are non-trivial since most of these small bodies are not regularly shaped similarly and thus can neither be approximated as point sources nor spherical ones.

Hence, gravity models are needed. Historically, three approaches have been developed to represent a gravity field.

The *Spherical harmonics* approach uses the coefficients of the gravity field's Fourier series expansion in spherical (or similar) coordinates. However, the technique has several

[1]`https://en.wikipedia.org/wiki/433_Eros`, last accessed: 12.09.2022
[2]`https://ssd.jpl.nasa.gov`, last accessed: 12.09.2022

weaknesses: The spherical harmonics are only guaranteed to converge above the minimum Brillouin sphere but may be divergent inside. The minimum Brillouin sphere is the sphere centered around the origin of the body and still enveloping all of the body's mass. Thus, the model's use is problematic, especially in the surface area. [5]

Further, outside of the minimum Brillouin sphere, the model's convergence becomes slower if the examined body is of increasingly irregular shape [6].

Mascon models are the second approach. Here, the body is represented as a combination of multiple point masses, so-called mascons (short for "mass concentrations" [7]) filling up its volume. The mascon elements do not need to be of uniform size. Instead, various approaches exist, combining mascons of different weights [8]. Its simplicity and ability to model irregular shapes and density distributions like they appear for small bodies make the model appealing. Regardless, high accuracy of the gravity field can only be achieved with an extensive number of mascons. Even then, the field's accuracy near the body's surface remains challenging due to the discretized mass distribution. [8]

The *polyhedral gravity model* is the third of the triumvirate and particularly useful for the application on irregularly shaped bodies. It requires only two assumptions: homogeneous density and the knowledge of the polyhedron's mesh. The *polyhedral gravity model* even has closed analytical formula, which can be evaluated for an arbitrary point in space.

Further, a very recent fourth approach, *geodesyNets*, uses a fully connected neural network to map cartesian coordinates within a body to a corresponding density, finally making it possible to compute the gravitational field by integrating over the (neural) density field. The model is trained with any gravity signal as ground truth. In the first iteration, this was done using synthetic data generated by the *mascon model*, but the goal is to train the model onboard with real measured data. [6]

This work covers the *polyhedral gravity model*, presented in detail in the following Chapters, and implemented[3] in `C++17` with a strong parallelized backbone and a lightweight simple-to-use interface in `Python`, finally deployed on *conda*[4] to make it available to the scientific community.

[3]`https://github.com/esa/polyhedral-gravity-model` last accessed: 12.09.2022
[4]`https://anaconda.org/conda-forge/polyhedral-gravity-model` last accessed: 12.09.2022

2. Theoretical Background

2.1. Overview

Generally speaking, one can distinguish between numerical and analytical methods for determining the full gravity tensor of a polyhedron. On the one hand, numerical approaches cover techniques of numerical integration procedures like those presented by Talwani and Ewing [9] or Tsoulis et al. [10]. On the other hand, there have been multiple approaches to derive the analytical solution for the potential and attraction caused by a polyhedral source. Here one can mention Barnett [11], Pohánka [12], as well as, Werner and Scheeres [13, 14]. The list of available procedures to determine the analytical solution is long. For the following, we stick to the line integral approach presented by Petrović in 1996 [15], later refined in terms of handling singularities by Tsoulis and Petrović in 2001 [16]. Werner and Scheeres, as well as, Tsoulis' approaches, are later examined from an implementation-wise point of view in Chapter 3.

Equations 2.1–2.3 describe the gravitational potential V, the attraction V_{x_i} in the three (cartesian) directions, and the elements of the second derivative tensor $V_{x_i x_j}$ for an arbitrary point in space P around a polyhedral source with volume U, given a constant density ρ and the gravitational constant $G = 6.6743 \cdot 10^{-11}\ m^3 kg^{-1} s^{-2}$.

$$V = G\rho \iiint_U \frac{1}{l}\, dU \tag{2.1}$$

$$V_{x_i} = G\rho \iiint_U \frac{\partial}{\partial x_i}\left(\frac{1}{l}\right) dU \qquad \text{with } (i = 1, 2, 3) \tag{2.2}$$

$$V_{x_i x_j} = G\rho \iiint_U \frac{\partial^2}{\partial x_i \partial x_j}\left(\frac{1}{l}\right) dU \qquad \text{with } (i, j = 1, 2, 3) \tag{2.3}$$

$$\text{with } l = \sqrt{(x_1 - x_1')^2 + (x_2 - x_2')^2 + (x_3 - x_3')^2} \tag{2.4}$$

Further, Equation 2.4 is the distance function which refers to a coordinate system with the origin $x_1' = x_2' = x_3' = 0$ and a basis of orthonormal unit vectors $(\overrightarrow{e_1}, \overrightarrow{e_2}, \overrightarrow{e_3})$ located at P [17].

The Equations 2.1–2.4 mentioned above are then transformed by the repeated application of the divergence theorem of Gauss, as described by Petrović [15]. Simply expressed, one transforms the given volume integrals into surface integrals whose number equals the number of faces of the polyhedron. Secondly, the surface integrals are transformed into line integrals, each corresponding to a segment of the polyhedral faces. Further, specific terms expressing the analytical solution of the limiting values of the line integrals are added if singularities occur, later explained in detail in Subsection 2.2.6. As a result, the technique can be applied everywhere in mathematical space [16].

2.2. Resulting Equations and Details

This application of the line integral approach finally yields the expressions given in Equations 2.5–2.7, calculating the full gravity tensor for point P. Notably, P is located on the origin. In order to calculate the gravity tensor for an arbitrary point P, one has to add an offset to the coordinates of the polyhedral source, as explained in Section 4.3.

$$V = \frac{G\rho}{2} \cdot \sum_{p=1}^{n} \sigma_p h_p \quad \cdot \left[\sum_{q=1}^{m} \sigma_{pq} h_{pq} LN_{pq} + h_p \sum_{q=1}^{m} \sigma_{pq} AN_{pq} + \text{sing}_{\mathcal{A}_p} \right] \tag{2.5}$$

$$V_{x_i} = G\rho \cdot \sum_{p=1}^{n} \cos(\overrightarrow{N_p}, \overrightarrow{e_i}) \quad \cdot \left[\sum_{q=1}^{m} \sigma_{pq} h_{pq} LN_{pq} + h_p \sum_{q=1}^{m} \sigma_{pq} AN_{pq} + \text{sing}_{\mathcal{A}_p} \right] \tag{2.6}$$

$$V_{x_i x_j} = G\rho \cdot \sum_{p=1}^{n} \cos(\overrightarrow{N_p}, \overrightarrow{e_i}) \quad \cdot \left[\sum_{q=1}^{m} \cos(\overrightarrow{n_{pq}}, \overrightarrow{e_j}) LN_{pq} + \sigma_p \cos(\overrightarrow{N_p}, \overrightarrow{e_j}) \sum_{q=1}^{m} \sigma_{pq} AN_{pq} + \text{sing}_{\mathcal{B}_{pj}} \right] \tag{2.7}$$

with $i, j = 1, 2, 3$

Generally, Equations 2.5–2.7 still contain the gravitational constant G and the density ρ untouched. In contrast, the triple integral has been replaced by a double summation. The outer summation runs over the faces of the polyhedron with running index p, whereas the inner sums iterate with q over the segments of each face. So if we use a polyhedron with a triangulated mesh, it always holds $q \in (1, 3)$ and $m = 3$. Equations 2.8–2.9 portray Equations 2.6–2.7 in a more compact notation, with the latter using the outer product $\otimes$.

$$\overrightarrow{V_x} = G\rho \cdot \sum_{p=1}^{n} \overrightarrow{N_p} \cdot \quad \left[\sum_{q=1}^{m} \sigma_{pq} h_{pq} LN_{pq} + h_p \sum_{q=1}^{m} \sigma_{pq} AN_{pq} + \text{sing}_{\mathcal{A}_p} \right] \tag{2.8}$$

$$\overline{\overline{V_{xx}}} = G\rho \cdot \sum_{p=1}^{n} \overrightarrow{N_p} \otimes \quad \left[\sum_{q=1}^{m} \overrightarrow{n_{pq}} LN_{pq} + \sigma_p \overrightarrow{N_p} \sum_{q=1}^{m} \sigma_{pq} AN_{pq} + \text{sing}_{\mathcal{B}_{pj}} \right] \tag{2.9}$$

2.2.1. Computation Point P and its Projection Points P' and P"

The different Equations 2.5–2.7 all rely on knowing computation point P and its projections comprising P' and P''. P' is the orthogonal projection of P onto the plane S_p of the face with index p. P'' is the orthogonal projection of P' onto each segment G_{pq} of the face with index p and its segments enumerated with q. The position of P' and P'' are also of interest for the applications of the singularity correction terms, as explained later in Subsection 2.2.6. Figure 2.1 gives an example of the location of P, P', and P'' for an irregularly shaped pentagon.

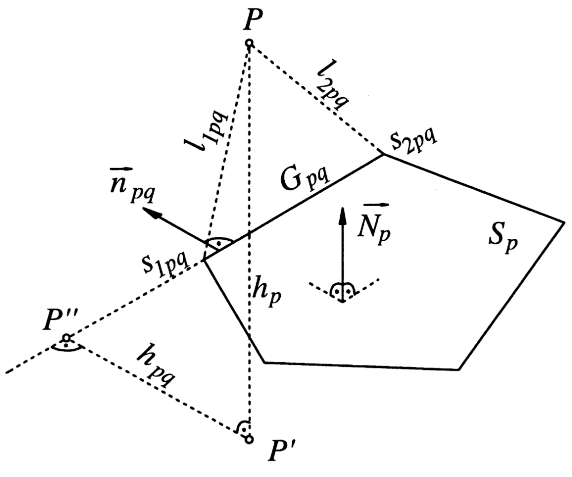


Figure 2.1.: Geometric properties of one polyhedral face in Equations 2.5–2.9 [15]

2.2.2. Vectors and Unit Normals

Further, one has to calculate the plane unit normals $\overrightarrow{N_p}$ and the segment unit normals $\overrightarrow{n_{pq}}$ respectively for each face p and every of its segments q. Equation 2.11 and Equation 2.12 show how this computation is performed by utilizing the vectors $\overrightarrow{G_{pq}}$ describing the line segments of the polyhedron.

$$\overrightarrow{G_{pq}} = (x_{p(q+1)} - x_{pq}) \cdot \overrightarrow{e_1} + (y_{p(q+1)} - y_{pq}) \cdot \overrightarrow{e_2} + (z_{p(q+1)} - z_{pq}) \cdot \overrightarrow{e_3} \tag{2.10}$$

$$\overrightarrow{N_p} = \frac{\overrightarrow{G_{p1}} \times \overrightarrow{G_{p2}}}{|\overrightarrow{G_{p1}} \times \overrightarrow{G_{p2}}|} \tag{2.11}$$

$$\overrightarrow{n_{pq}} = \frac{\overrightarrow{G_{pq}} \times \overrightarrow{N_p}}{|\overrightarrow{G_{pq}} \times \overrightarrow{N_p}|} \tag{2.12}$$

with $x_{pq}/y_{pq}/z_{pq}$ as the cartesian coordinates of the q-th vertex of face p

Here, it is essential to note that the points making up the faces of the polyhedral source must be in counterclockwise order so that the plane unit normals $\overrightarrow{N_p}$ point outwards the polyhedron [18]. If the vertices are sorted in a clockwise manner, i.e., the normals are inwards pointing, the sign of the result is going to be exchanged. If the vertices ordering is mixed, the results are going to be incorrect.

The constraint of outward pointing normals can be checked with, e.g., the Möller–Trumbore intersection algorithm [19]. One has to check for each calculated normal how often it intersects the polyhedron. If a normal intersects the polyhedron even times, it is pointing outwards. The algorithm accompanies this work's implementation.

2.2.3. Distances and Directions Related to Every Face

Next, one can determine the direction of the plane unit normal σ_p as described in Equation 2.13. This property represents the relative position of P with respect to the pointing

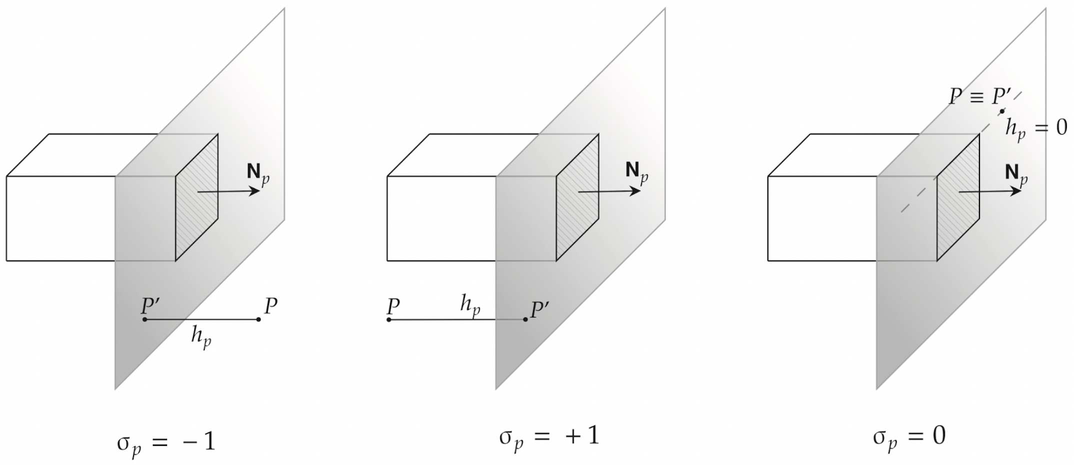


Figure 2.2.: Relative position of computation point P, P' and $\overrightarrow{N_p}$ with the resulting σ_p and h_p

direction of $\overrightarrow{N_p}$ [17]. If $\overrightarrow{N_p}$ points to the half-space that includes P then σ_p is negative, whereas the opposite holds if P is in the other half-space. Lastly, if $\sigma_p = 0$ then P lies on the face described by $\overrightarrow{N_p}$ as shown by Figure 2.2.

$$\overrightarrow{N_p} \cdot (-\overrightarrow{G_{p1}}) = \begin{cases} > 0, & \sigma_p = -1 \\ = 0, & \sigma_p = 0 \\ < 0, & \sigma_p = 1 \end{cases} \tag{2.13}$$

For the next steps, we need to calculate the Hessian formalism often written down in the vector equation of a plane $ax + by + cz = d$ for every face of the polyhedron. Using this formalism enables the calculation of the distance between P and P' abbreviated with h_p. This computation is shown in Equation 2.14.

$$h_p = \left| \frac{d}{\sqrt{a^2 + b^2 + c^2}} \right| \tag{2.14}$$

Consequently, it is possible to calculate the actual position of P' for every plane S_p according to Equation 2.15 with the before calculated quantities.

$$P'_{pi} = \begin{cases} |N_{pi} \cdot h_p| \text{ if } x < 0 \\ \begin{cases} -N_{p_i} \cdot h_p \text{ if } N_{pi} > 0 \\ N_{p_i} \cdot h_p \end{cases} \end{cases} \tag{2.15}$$
$$\text{with } (i, x) \in \left(x, \frac{d}{a}\right), \left(y, \frac{d}{b}\right), \left(z, \frac{d}{c}\right)$$

2.2.4. Distances and Directions Related to Every Segment

With the coordinates of P' known for every face, one can determine the orientations σ_{pq} of the segment normals $\overrightarrow{n_{pq}}$ by using Equation 2.16. The orientation σ_{pq} is negative if $\overrightarrow{n_{pq}}$

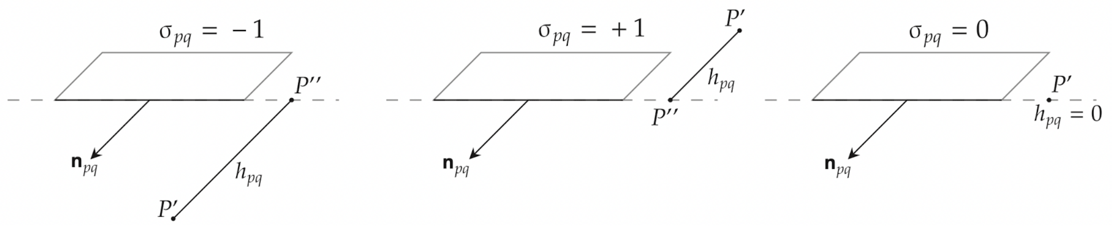


Figure 2.3.: Relative position of computation point P' and $\overrightarrow{n_{pq}}$ with the resulting σ_{pq} and h_{pq}

points to the half-plane containing the point P' and it is positive if the opposite holds. In case of $\sigma_{pq} = 0$, P' resides on the line segment represented by $\overrightarrow{G_{pq}}$. Figure 2.2 and Figure 2.3 illustrate this relation and respectively the three cases.

$$\overrightarrow{n_{pq}} \cdot (P'_p - V_{pq}) = \begin{cases} > 0, & \sigma_{pq} = -1 \\ = 0, & \sigma_{pq} = 0 \\ < 0, & \sigma_{pq} = 1 \end{cases} \tag{2.16}$$

with V_{pq} as the first vertex of the segment vector $\overrightarrow{G_{pq}}$

In order to find the position of P'' for every segment, one needs to solve a system of 3 linear equations. These equations are depicted in Equations 2.17–2.19. Equation 2.17 depicts the condition that the vector $\overrightarrow{P''_{pq}P'_p}$ is orthogonal to segment vector $\overrightarrow{G_{pq}}$. Equation 2.18 relates to the fact that $P'_p, P''_{pq}, \overrightarrow{G_{pq}}$ are coplanar and Equation 2.19 states that P''_{pq} and $\overrightarrow{G_{pq}}$ are colinear. Figure 2.1 exemplifies these relations in a more illustrative way. [17]

$$\overrightarrow{P'_p} \cdot \overrightarrow{G_{pq}} = \overrightarrow{P''_{pq}} \cdot \overrightarrow{G_{pq}} \tag{2.17}$$

$$\overrightarrow{P'_p} \cdot \left(\overrightarrow{G_{pq}} \times \overrightarrow{P'_pV}\right) = \overrightarrow{P''_{pq}} \cdot \left(\overrightarrow{G_{pq}} \times \overrightarrow{P'_pV}\right) \tag{2.18}$$

$$V_{pq} \cdot \left(\left(\overrightarrow{G_{pq}} \times \overrightarrow{P'_pV}\right) \times \overrightarrow{G_{pq}}\right) = \overrightarrow{P''_{pq}} \cdot \left(\left(\overrightarrow{G_{pq}} \times \overrightarrow{P'_pV}\right) \times \overrightarrow{G_{pq}}\right) \tag{2.19}$$

with V_{pq} as the first vertex of the segment vector $\overrightarrow{G_{pq}}$

The distances h_{pq} are then trivially computed knowing the quantities P''_{pq} and P'_p with the help of Equation 2.20.

$$h_{pq} = \left|\overrightarrow{P''_{pq} - P'_p}\right| \tag{2.20}$$

As Figure 2.3 shows, if $\sigma_{pq} = 0$ then a evaluation is unnecessary since in this case $h_{pq} = 0$.

2.2.5. Transcendental Expressions

Not yet calculated are the transcendental expressions AN_{pq} and LN_{pq} serving as abbreviations for the terms shown in Equation 2.21 and Equation 2.22. Both of them are a result of the line integral approach and the exact solutions for certain parts of the integration process. [15]

$$LN_{pq} = \ln(s_{2_{pq}} + l_{2_{pq}}) - \ln(s_{1_{pq}} + l_{1_{pq}}) = \ln\left(\frac{s_{2_{pq}} + s_{1_{pq}}}{l_{2_{pq}} + l_{1_{pq}}}\right) \tag{2.21}$$

$$AN_{pq} = \arctan\left(\frac{h_p s_{2_{pq}}}{h_{pq} l_{2_{pq}}}\right) - \arctan\left(\frac{h_p s_{1_{pq}}}{h_{pq} l_{1_{pq}}}\right) \tag{2.22}$$

$$\text{with} \quad l_{i_{pq}} = \left|\overrightarrow{PV_i}\right|, \quad s_{i_{pq}} = \left|\overrightarrow{P''_{pq}V_i}\right|$$

$$\text{and } V_i \text{ as segment vertex and } i \in (1,2)$$

Equations 2.21–2.22 contain the distances l_1, l_2 and s_1, s_2. The former pair reflects the 3D distances between computation point P and each segment endpoint, whereas the latter describes the 1D distances between the projection of computation point P on the corresponding segment P'' and each segment endpoint. These properties also are visualized in Figure 2.1. Depending on the relative position of P'' with respect to the two vertices, one has to control the signs as depicted in Table 2.1 [20].

Option	$\forall i : \lvert s_i \rvert < \lvert G_{pq} \rvert$	$\lvert s_2 \rvert < \lvert s_1 \rvert$	$\lvert s_1 \rvert < \lvert s_2 \rvert$	$\forall i : \lvert s_i \rvert = \lvert l_i \rvert$		
Further Distinguishing Case	-	-	-	P lies inside segment	P located on direction of the segment from its right	P located on direction of the segment from its left
Signs	$s_1 = -\lvert s_1 \rvert$ $s_2 = \lvert s_2 \rvert$ $l_1 = \lvert l_1 \rvert$ $l_2 = \lvert l_2 \rvert$	$s_1 = -\lvert s_1 \rvert$ $s_2 = -\lvert s_2 \rvert$ $l_1 = \lvert l_1 \rvert$ $l_2 = \lvert l_2 \rvert$	$s_1 = \lvert s_1 \rvert$ $s_2 = \lvert s_2 \rvert$ $l_1 = \lvert l_1 \rvert$ $l_2 = \lvert l_2 \rvert$	$s_1 = -\lvert s_1 \rvert$ $s_2 = \lvert s_2 \rvert$ $l_1 = -\lvert l_1 \rvert$ $l_2 = \lvert l_2 \rvert$	$s_1 = -\lvert s_1 \rvert$ $s_2 = -\lvert s_2 \rvert$ $l_1 = -\lvert l_1 \rvert$ $l_2 = -\lvert l_2 \rvert$	$s_1 = \lvert s_1 \rvert$ $s_2 = \lvert s_2 \rvert$ $l_1 = \lvert l_1 \rvert$ $l_2 = \lvert l_2 \rvert$

Table 2.1.: Coordinate Transformation for 3D distances l_1, l_2 and 1D distances s_1, s_2 [20]

2.2.6. Singularity Terms

Equation 2.23 and Equation 2.24 introduce the singularity terms and their calculation. Their evaluation is only necessary in the edge cases (a), (b), (c) and fulfills the purpose of overcoming the issues of appearing singularities due to the line integral approach. These singularities appear when the orthogonal projection of the computation point P'_p falls into the polygonal surface S_p or the onto the vectors $\overrightarrow{G_{pq}}$ describing the edges of face p. In such a case, the calculation contains a division by zero. The singularity terms overcome "these singularities and enable [...] the line integral formalism to be applied everywhere in space, regardless of the relative position of points P and P'" [17].

$$\text{sing}_{\mathcal{A}_p} = \begin{cases} -2\pi h_p & \text{if (a)} \\ -\pi & \text{if (b)} \\ -\theta h_p & \text{if (c)} \\ 0 & \text{otherwise} \end{cases} \tag{2.23}$$

$$\text{sing}_{\mathcal{B}_{pj}} = \begin{cases} -2\pi \cos(\overrightarrow{N_p}, e_j)\sigma_p & \text{if (a)} \\ -\pi \cos(\overrightarrow{N_p}, e_j)\sigma_p & \text{if (b)} \\ -\theta \cos(\overrightarrow{N_p}, e_j)\sigma_p & \text{if (c)} \\ 0 & \text{otherwise} \end{cases} \tag{2.24}$$

$$\text{with conditions} \begin{cases} (a)\ P'_p \text{ lies inside plane } S_p \\ (b)\ P'_p \text{ is located on } G_{pq}\text{, but not at any of it vertices} \\ (c)\ P'_p \text{ is located at one of } G_{pq}\text{'s vertices} \\ \text{otherwise } P'_p \text{ is located outside } S_p \end{cases}$$

3. Related Work

Chapter 2 already introduces the theoretical approach undertaken by Tsoulis et al. computing the full gravity tensor for an arbitrary point P around a polyhedral source. This chapter now provides additional insights into actual implementations of the previously described foundation.

3.1. Tsoulis' Work

Tsoulis implemented his analytical solution twice: in 2012 in `FORTRAN` and in 2021 in `MATLAB`. Both were the primary reference for the later presented implementation in `C++` with the accompanying interface in `Python`. Generally, both implementations of Tsoulis are similar in their structure and control flow, relying only on procedures but no special data classes and are documented in a comprehensible way.

3.1.1. FORTRAN

Tsoulis et al. were kind enough to make their `FORTRAN` implementation[1] [17] available online via the *Society of Exploration Geophysicists*. Aside from the difficulty in integrating `FORTRAN` code with modern programming languages and utilities such as `Python`, there are some improvements to the implementation of this work. E.g. the following details were changed to improve the structure, robustness and maintainability of the code: The `C++` implementation sticks to a modular partly class-based approach dividing the functionality across multiple files rather than being restricted to `FORTRAN`'s limitations. Further, this work reduces the number of operations by reusing computed values and simplifying calculations. The C++ implementation's I/O uses established mesh formats rather than sticking to legacy text files and does not require recompiling when the input changes.

To summarize, `C++` uses the same theoretical foundation as in Chapter 2 but utilizes a more structured approach with an optimized iteration scheme, reducing code size and eliminating unnecessary operations. The results of the `FORTRAN` implementation are numerically comparable to this work's results, with slight deviations as the `C++` implementation follows Tsoulis' recent revision in `MATLAB` [20].

3.1.2. MATLAB

The `MATLAB` implementation[2] [20] is accessible via *GitHub* with a slightly modified GPL3 license. It is a revision of the previously introduced `FORTRAN` implementation but no structural overhaul. However, the implementation conforms exactly to the described methods presented in Chapter 2 and produces numerically equivalent results as the `C++` implementations.

[1]`https://software.seg.org/2012/0001/index.html`, last accessed: 12.09.2022
[2]`https://github.com/Gavriilidou/GPolyhedron`, last accessed: 12.09.2022

3.2. Werner and Scheeres' Work

Other approaches include the analytical solution provided by Werner and Scheeres [14]. It has been implemented, e.g., as part of the *Small Bodies Geophysical Analysis Tool*[3] written in `C++`. In contrast to the techniques used by Tsoulis et al., their model relies mainly on the calculation of dyads for edges and faces. These combine the unit normals and connect two faces/edges with each other. This approach is advantageous as the results are "expressed intrinsically" [14] not related to any coordinate systems. However, the computational effort is comparable to Tsoulis' approach since it also contains expensive transcendental terms involving the evaluation of arctans and logarithms and the iteration over every segment in the end with one exception: The arctan only needs to be computed for every face, not segment as in Tsoulis' algorithm. Nonetheless, if we stick to triangles and assume the availability of sufficient large vector registers, this extra effort amortizes, as presented in Section 4.3.

[3]`https://github.com/bbercovici/SBGAT`, lasted accessed: 12.09.2022

Part II.

Implementation

4. Architecture

This chapter presents the components of the implementation and summarizes their functionality.

4.1. Overview of Components

The core `C++` implementation consists of five components (gray colored in Figure 4.1), with an additional sixth component defining the `Python Interface` (turquoise colored in Figure 4.1). Figure 4.1 depicts these components and their interrelations. Further, six external libraries are used to support these modules. These external libraries are all automatically set up via the `CMake` build system.

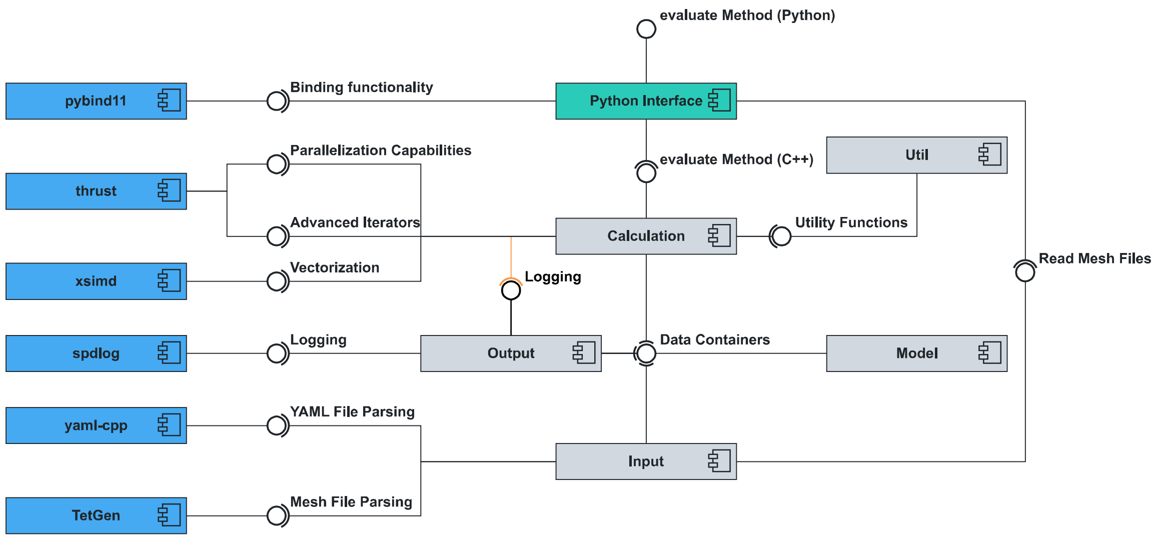


Figure 4.1.: The whole implementation and its dependencies as UML Component Diagram. The blue colored components are third party dependencies, whereas gray colored components are the implementation's core. The orange connection is only active when the log level is set to debugging.

The first part of the implementation's core is the `Model` component which provides the necessary constructs for the representation of a polyhedron and the resulting full gravity tensor. The second one is the `Calculation` component capable of computing the full gravity tensor with its `evaluate(...)` method. `Input` is responsible for reading in mesh files and configuration details like the constant density of the polyhedron or the desired computation points. In contrast, `Output` is more of a boundary component only in charge of producing `CSV` file output and interfacing to the logging library `spdlog`. The latter interface only exists when the executable is compiled with the log level set to at least debugging (orange colored in Figure 4.1). The `Util` component provides mathematical functions and utility for more elegant code in total. The sixth component is the `Python Interface` which defines the method overloads of `evaluate(...)` exposed to the `Python` environment.

The following section presents these components in more detail and lists essential design choices during the implementation. Figure 4.2 shows the data flow, the I/O behavior, and the connectivity between the modules in terms of data flow during the execution of the `C++` executable, and it is helpful to be considered by the reader in the subsequent sections.

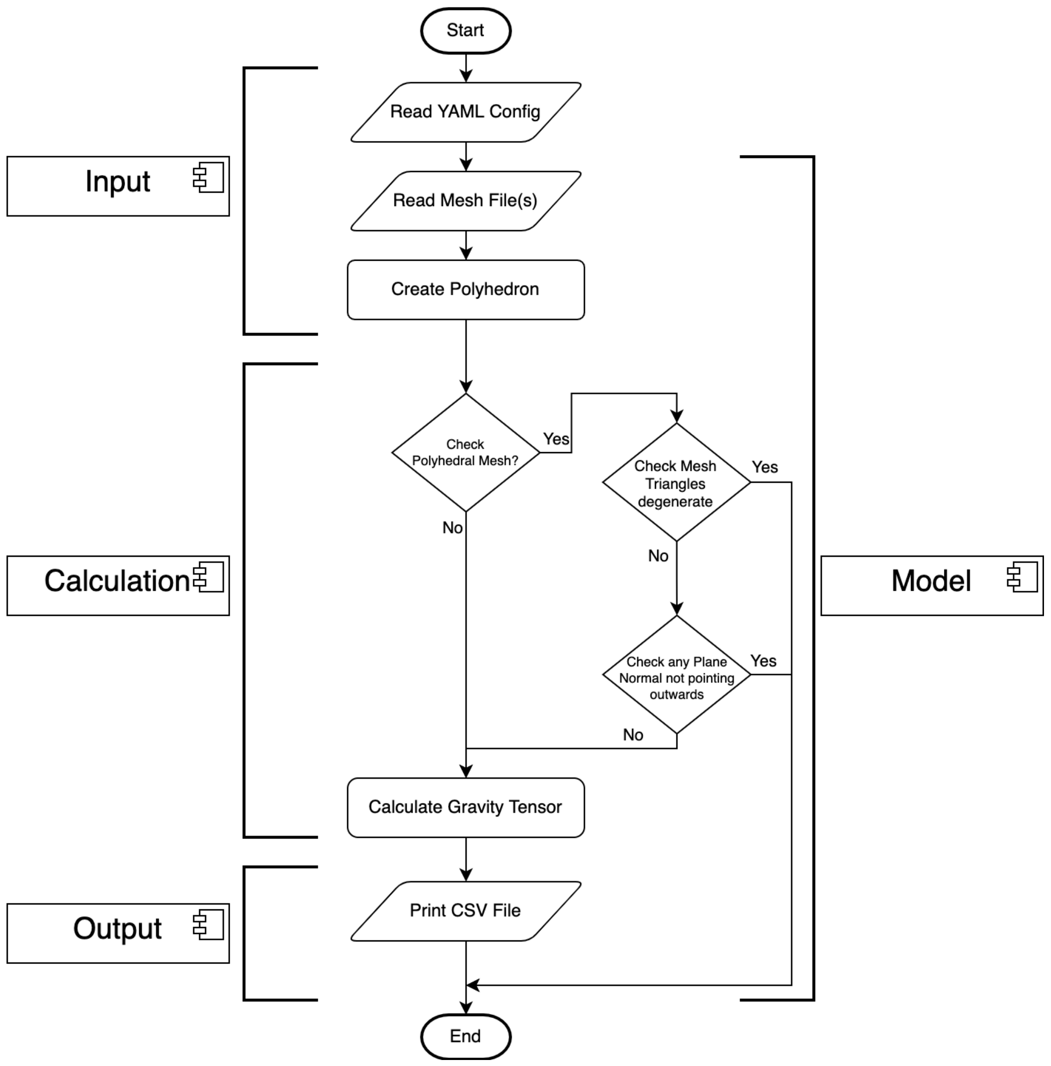


Figure 4.2.: The control flow of one execution run of the `C++` implementation as UML Flow Chart. The right and left sides show the involved components.

4.2. Model

The `Model` is the most significant component as it provides the data container `Polyhedron` used for storing the polyhedral mesh and data classes like `TranscendentalExpression`, `HessianPlane` or `Distance` which respectively group numerical values together and provide named access to them for better code readability. `GravityModelResult` also belongs to this group, but comes with one special functionality: Its method `eliminateRoundingErrors()` sets its members to exactly zero in case the floating point magnitudes are smaller than the implementation's defined epsilon $\epsilon = 1^{-14}$. This step is executed after the calculation and follows from the issue that even numerical neglectable rounding errors can accumulate over thousands of iterations.

The `Model` is the central component in Figure 4.1 since every communication between components relies on passing its data structures, but no further information. This makes the whole implementation more maintainable and simpler to extend as `Input` and `Output` and `Calculation` are not directly related neither logical nor functional.

4.3. Calculation

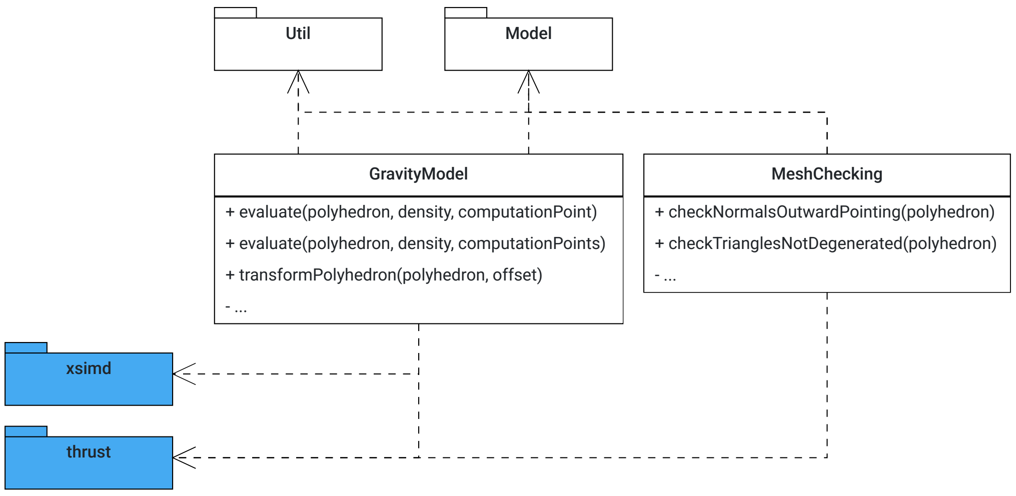

Util
Model
GravityModel
+ evaluate(polyhedron, density, computationPoint)
+ evaluate(polyhedron, density, computationPoints)
+ transformPolyhedron(polyhedron, offset)
- ...
MeshChecking
+ checkNormalsOutwardPointing(polyhedron)
+ checkTrianglesNotDegenerated(polyhedron)
- ...
xsimd
thrust

Figure 4.3.: UML Diagram of the `Input` component. Third-party libraries are colored in blue. Functions in the `detail` sub-namespace are not shown since they are not part of the public interface.

The module `Calculation` consists of the stateless namespace `GravityModel` that contains the actual procedures to evaluate the full gravity tensor at a given point in space and the namespace `MeshChecking` containing methods for checking the mesh input, as Figure 4.3 illustrates.

Starting with `GravityModel`, the main `evaluate(...)` method is made up of several sub-functions, each calculating one the properties presented in Chapter 2 on the plane-level. The array of appearing functions can be seen in Algorithm 1. If every property for a given plane p has been computed, the sum is calculated according to Equations 2.5–2.7

and aggregated in running variables. The here presented approach has the advantage that intermediate values are only present in memory for one plane at a certain point in time, and only the accumulating variables need to be stored persistently for the duration of the whole algorithm—an invaluable benefit when working with bigger polyhedral meshes.

Two external libraries are used in the `Calculation` module: *thrust*[1] and *xsimd*[2]. The first one provides the implementation with algorithms similar to the `C++` standard library but with easy-to-activate parallelization capabilities. Further, *thrust* comes with a set of useful tools like `zip_iterator` or `transform_iterator`. For example, the latter is used to dynamically apply an offset on the polyhedron's coordinates depending on the given computation point P. This method keeps the actual implementation clean from a repeating addition of the offset while at the same time not modifying anything persistently in memory. The second one is used to vectorize expensive computations like the arctan. Chapter 5 is going to present both tools with respect to their performance impacts.

Algorithm 1: The Polyhedral Gravity Calculation. Values with a subscript q imply an inner loop running over the segments of each face p.

```
  Input: Polyhedron polyhedron, Density $\rho$, Computation Point $P$
  Output: Potential, Attraction and 2. Derivative Tensor at $P$
1 forall faces $p \in polyhedron$ do
      1. Calculate Segment Vectors $\overrightarrow{G_{pq}}$ (2.10)
      2. Compute Plane Unit Normal $\overrightarrow{N_p}$ (2.11)
      3. Compute Plane Normal Orientation $\sigma_p$ (2.13)
      4. Compute Hessian Normal Form
      5. Compute Distance $h_p$ between $P$ and $P'$ (2.14)
      6. Compute Position of $P'$ (2.15)
      7. Compute Segment Normals Orientations $\sigma_{pq}$ (2.16)
      8. Compute Position of $P''$ (2.17) (2.18) (2.19)
      9. Compute Segment Normals $\overrightarrow{n_{pq}}$ (2.12)
     10. Compute Segment Distances $h_{pq}$ between $P$ and $P''$ (2.20)
     11. Compute 3D distances $l_1, l_2$ between $P$ and vertices and 1D distances $s_1, s_2$ between
         $P''$ and vertices
     12. Compute Transcendental Expressions $LN_{pq}$ (2.21) and $AN_{pq}$ (2.22)
     13. Compute Singularites $\text{sing}_{\mathcal{A}_p}$ (2.23) and $\text{sing}_{\mathcal{B}_{pj}}$ (2.24)
    Sum up according to Equations 2.5–2.7 onto running variables
```

As Subsection 2.2.2 mentions, the plane unit normals used for the calculation need to point outwards of the polyhedron. Typically, the vertices of a polyhedral source file are

[1]`https://github.com/NVIDIA/thrust`, last accessed: 02.10.2022
[2]`https://github.com/xtensor-stack/xsimd`, last accessed: 02.10.2022

sorted either counterclockwise so that all normals are pointing outwards or clockwise so that all normals are inwards pointing. In order to check for a counterclockwise sorting, one can use the Möller–Trumbore intersection algorithm. The calculation component implements this algorithm in a parallelized way to check if all plane normals of a polyhedron are pointing outwards. It further provides a method to check the input mesh for degenerate triangles. Here, one checks if every triangle's surface is greater than zero. Since the Möller–Trumbore algorithm has quadratic complexity and is not required if one already knows the properties of the mesh file, it is only an option and no mandatory element of the routine, as Figure 4.2 shows.

4.4. Input

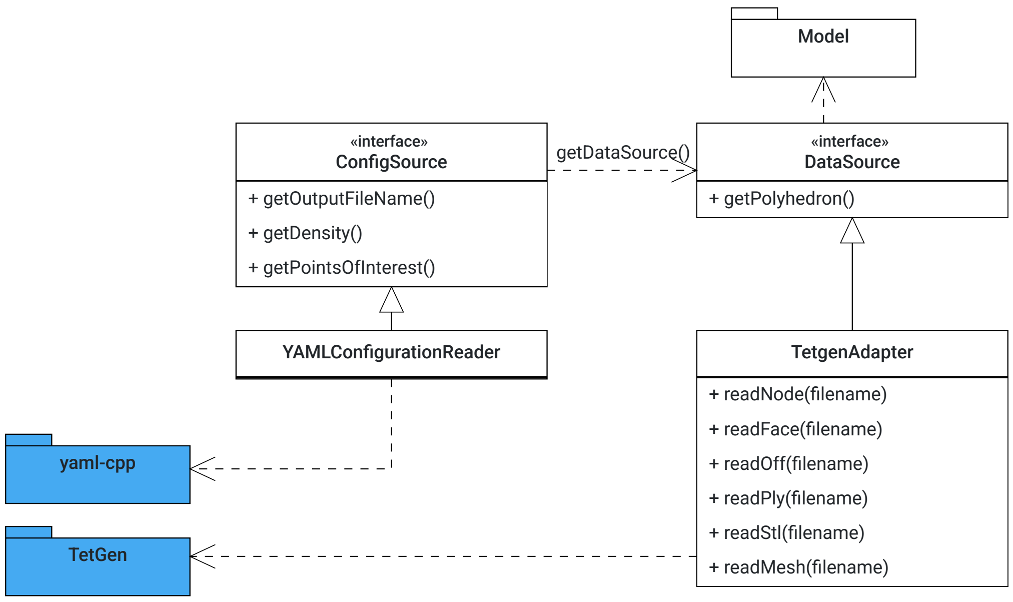


Figure 4.4.: UML Class Diagram of the `Input` component. Third party libraries are colored in blue.

The third component is the `Input`. It consists of two interfaces: `ConfigSource` and `DataSource`, and two implementing classes: `YAMLConfigurationReader` relying on *yaml-cpp*[3] and `TetgenAdapter` depending on *TetGen*[4], as shown by Figure 4.4. The first duo collects the configuration data from a `yaml` file including density, computation point(s) and the output `csv` filename. The latter two handle the data input and read a polyhedral mesh from various file formats according to the use cases presented in Figure 4.5.

[3]`https://github.com/jbeder/yaml-cpp`, last accessed: 22.09.2022
[4]`https://github.com/libigl/tetgen`, last accessed: 22.09.2022

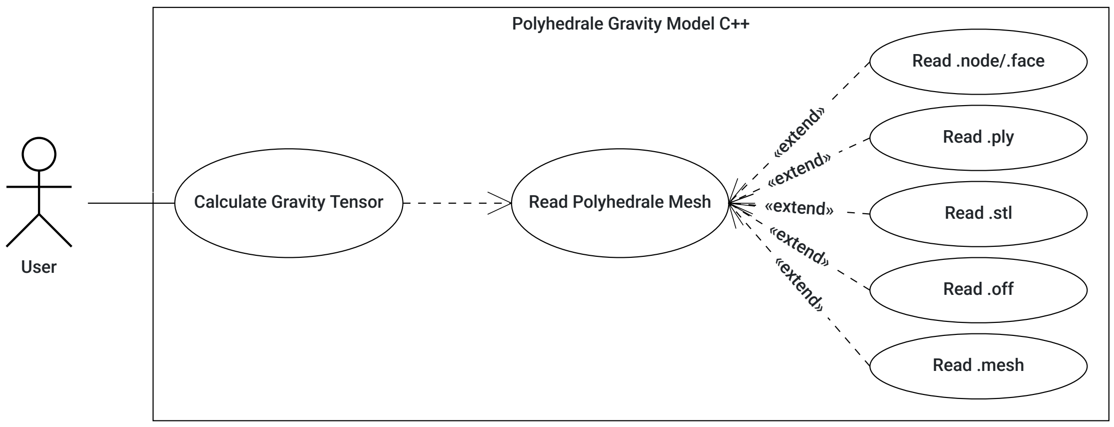


Figure 4.5.: UML Use Case Diagram of the `Input` component.

4.5. Output

The `Output` fulfills two purposes. First, it can print the calculated full gravity tensors for all computation points, which can be arbitrary many to a `CSV` file. The decision to include a `CSVWriter` was made at the end of the project when it was undeniable that thousands of points' gravity written on `stdout` would not be very usable. The second purpose lies behind the orange connection in Figure 4.1. The component initializes a static `DEFAULT_LOGGER` hiding the actual logger from *spdlog*[5] in order to configure logging without the necessity of a `main` function. The relation in Figure 4.1 is marked in orange since it is only active if the executable is compiled with `LOG_LEVEL` set to `DEBUG` level or lower. Otherwise, the preprocessor removes logging statements.

4.6. Python Interface

Figure 4.2 showed the data flow within the `C++` executable. In contrast, Figure 4.6 depicts the whole situation using the polyhedral gravity library from within `Python`. Configuration details are missing since the `Python` implementation can pass them as arguments to the `evaluate(...)` method defined by the sixth component `Python Interface`. This component depends on *pybind11*[6] providing the key functionality for creating the interface to the *Python Interpreter*.

In general, the module does not expose internal data structures like `Polyhedron` or `GravityModelResult`, but rather uses the base types of `C++` standard library. This enables automatic conversion between `Python` array-like structures (e.g. list or numpy arrays) and the `C++` types by *pybind11*. Further, it allows the utilization of all the syntactic sugar like tuple unpacking shown in Chapter 6. Using the standard mechanisms of *pybind11* implies that the data is copied[7] from the `Python` types to the `C++` types due to different memory layouts. One could change this behavior, but we decided against it for three reasons.

[5]`https://github.com/gabime/spdlog`, last accessed: 20.09.2022

[6]`https://github.com/pybind/pybind11`, last accessed: 22.09.2022

[7]`https://pybind11.readthedocs.io/en/stable/advanced/cast/stl.html`, last accessed: 01.10.2022

First, the separation keeps a clean split between `Python` and `C++`, and the user can rely on immutability. Second, there were some issues when using the opaque types and mixing `numpy` and `Python` lists. Lastly, the `Python` interface comes with file input capabilities. So, meshes of enormous size can be directly read from the source without any preprocessing in `Python`.

Additionally, the `Python Interface` exposes the Mesh Checks of the `Calculation` component and the capabilities of `Input` in a submodule called `utility`.

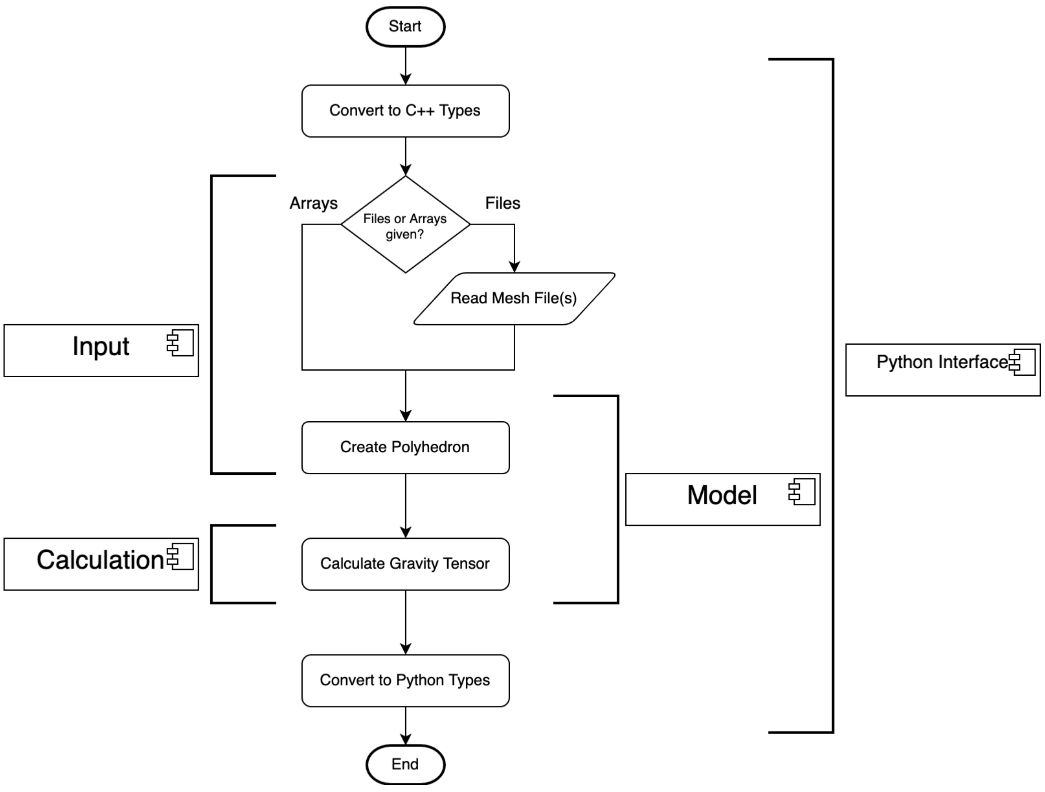


Figure 4.6.: The control flow of one execution run of the `Python` implementation as UML Flow Chart illustrating the two-way interface with either mesh file(s) or array input.

5. Runtime Measurements and Optimizations

This chapter introduces the undertaken measures with the ultimate goal of improving overall runtime.

5.1. Optimization and Parallelization

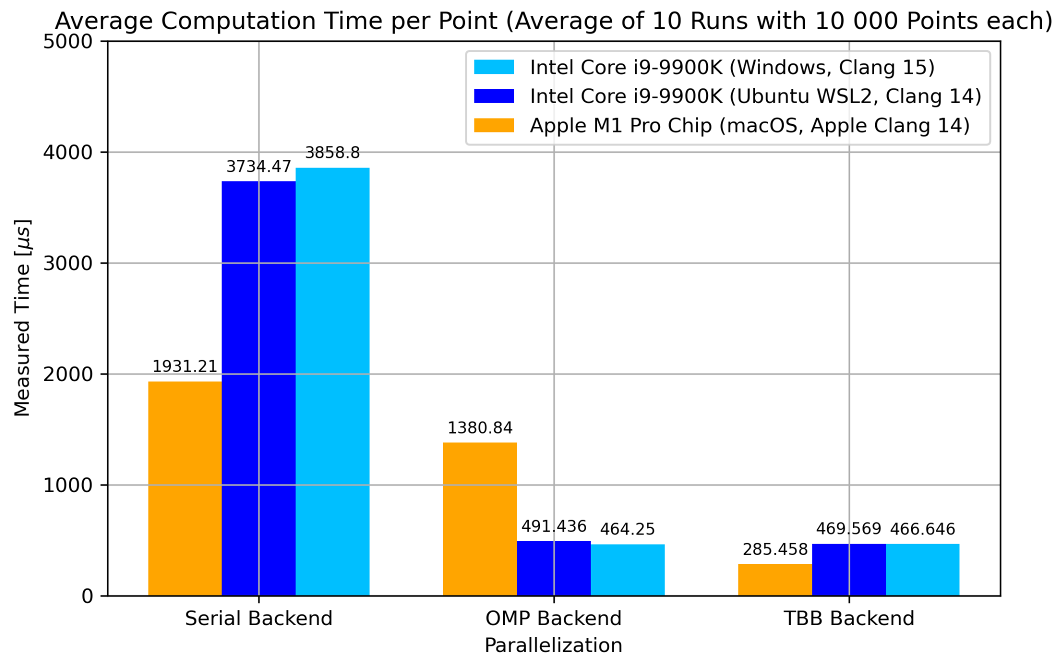


Figure 5.1.: Final time measurements of the implementation for different software and hardware setups. The program was compiled on *macOS* with *Apple Clang 14.0.0*, on *Linux (WSL2)* with *Clang 14.0.0*, and on *Windows* with *Clang 15.0.1*. The mesh of Eros consists of 24235 vertices and 14744 faces (displayed in Figure 7.3a). The average time measurements for a single point are given in microseconds [μs].

The first prototype did not calculate values plane-wise but rather property-wise. This decision led to an enormous memory footprint since everything needed to be stored until the final summation happened. When writing this first prototype, the thought was that one could reuse, e.g., the computed plane unit normals, which do not differ if one changes the computation point P. However, the implications on memory footprint made the approach unfavorable for bigger mesh sizes.

As a result, this approach was discarded and replaced by the plane-wise ansatz, which also enabled the utilization of the more efficient fused operation: `thrust::transform_reduce` requiring fewer memory reads and writes.

The library *thrust* comes with a variety of parallelization backends. These include the *serial C++ standard* (`CPP`), *OpenMP* (`OMP`), *Intel's Threading Building Blocks*, and *Nvidia's CUDA* (`CUDA`). The former three use the host's CPU, whereas the latter requires a GPU. Generally, the implementation of the polyhedral gravity model supports only the CPU parallelization backends. *CUDA* is not supported as it would require turning away from the current functional decomposition, a more CPU-related paradigm. Before compile time, the aforementioned three backends can be conveniently chosen and exchanged via the `CMake` variable `POLYHEDRAL_GRAVITY_PARALLELIZATION`. Figure 5.1 shows the comparison between these backends for two different processor architectures. Overall, the speed gain due to parallelization is significant for both processors.

Intel's Threading Building Blocks performs equally well compared to the *OpenMP* framework on `x86_64` architecture using both Clang as compiler. A comparison of the core utilization by VTune yielded similar results. However, on `aarch64` the `TBB` backend performs much better. *XCode Instruments'* inverted call tree shows that `OMP` waits and synchronizes the threads more often than `TBB`. Moreover, `TBB` uses task stealing more frequently. As a result, the core utilization is around five times lower for `OMP`. This finding fits well with the observed 4.8x speedup `TBB` provides compared to `OMP` on `aarch64`.

5.2. Vectorization

After completion of the parallelization efforts, profiling enabled the potential for further improvements. The program was profiled with *Intel VTune* for `x86_64` and *XCode Instruments* for `aarch64`. In both cases, one operation had a noticeable performance impact: arctan.

In order to improve the performance, *xsimd* was incorporated into the project to explicitly vectorize the arctan operations used during the calculation of the Transcendental Expressions, as shown in Equation 2.22. Since the computation of AN_{pq} requires two evaluations of arctan, one can effectively pack both in one vector register and "half" the number of operations. A simple change that reduced the average runtime by $8 - 24\%$ using *Intel's SSE* or *ARM's NEON* instruction sets, as Figure 5.2 shows.

The explicit vectorization of other operations, e.g., euclidean norms and logarithms, was also considered. However, there are two obstacles. First, the current implementation does not contain any pairwise applications of these operations where one could unify two unpacked operations into one. Second, the vectorization of less expensive operations like additions and multiplications being part of, e.g., the euclidean norm, did not positively impact the speed. Presumably, if one would rewrite the algorithm from a plane-wise implementation to property-wise computation, one could explicitly vectorize much more. However, this contradicts the finding at the opening of this Chapter 5, as the property-based implementation would consume too much memory - an inevitable trade-off.

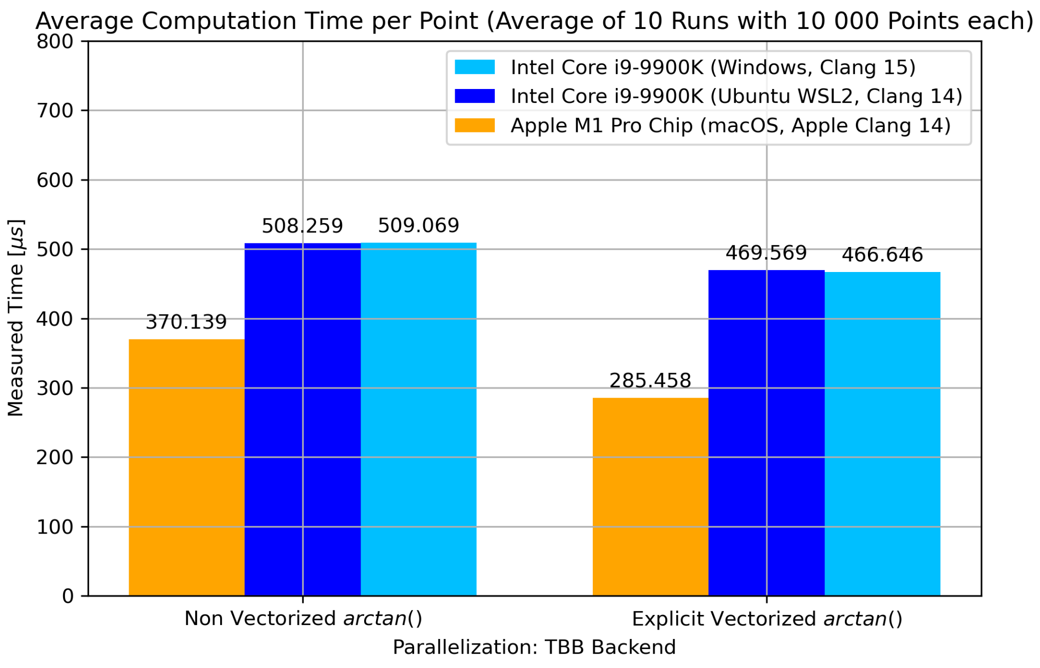


Figure 5.2.: The same scenario as in Figure 5.1, but one time with no explicit vectorization of the arctan with *xsimd*. The average time measurements for a single point are given in microseconds [μs].

6. Usage

This chapter briefly shows the actual utilization of the provided interfaces in `C++` and `Python` with the provided `evaluate(...)` method(s), which bring everything presented in Chapter 4 together.

Listing 1 shows the minimal example of running the gravity model from within `C++`. One must define a `Polyhedron`, the density, and a vector of computation points/ a single point.

```cpp
//Get the input values
std::unique_ptr<ConfigSource> config =
  std::make_unique<YAMLConfigReader>("config_file.yaml");
Polyhedron poly = config->getDataSource()->getPolyhedron();
double density = config->getDensity();
std::vector<std::array<double, 3>> computationPoints =
  config->getPointsOfInterest();
std::string outputFileName = config->getOutputFileName();

//Evaluate the full gravity tensor at every computation point
std::vector<GravityModelResult> result =
  GravityModel::evaluate(polyhedron, density, computationPoints);
```

Listing 1: Minimal usage example with the usage of a configuration file of the `C++` implementation

Listing 2 and Listing 3 present the application of the python interface. The `evaluate(...)` methods is more flexible since it also accepts directly file input (available formats were presented in Figure 4.5) in contrast to the version of the `C++` implementation. Of course, these methods support not only `numpy` arrays but also the basic `Python` list type. All decisions were made with the goal of keeping the interface as simple as possible.

The installation of the interface is simple since the `Python` package is available via *conda*[1]. Alternatively, one can clone the repository and install the package via *pip* from the source. Both methods work for all three operating systems Windows, macOS, and Linux. While the package on *conda* only supports the `x86_64` architecture, `aarch64` is also an option when building from source.

Further reference to the usage is accessible on the official documentation on *readthedocs*[2]. It contains extensive information about the available functions and supported input file types (and how to convert to them).

[1]`https://anaconda.org/conda-forge/polyhedral-gravity-model` last accessed: 12.09.2022
[2]`https://polyhedral-gravity-model-cpp.readthedocs.io/en/latest/` last accessed: 22.09.2022

```python
import numpy as np
import polyhedral_gravity as gravity_model
import polyhedral_gravity.utility as mesh_sanity

# Define input
cube_vertices = np.array([[-1, -1, -1], ...])
cube_faces = np.array([[1, 3, 2],...])
computation_point = [1, 0, 0]
density = 1.0

# Additional guard statement checking the mesh
# for degenerate traingles & normals outwards pointing
if mesh_sanity.check_mesh(cube_vertices, cube_faces):
  # For one point (alternative: computation_points could also be a list of points)
  potential, acceleration, tensor =
    gravity_model.evaluate(cube_vertices, cube_faces, density, computation_point)
```

Listing 2: Usage example of the Python with a polyhedron defined during runtime and mesh checking

```python
    import numpy as np
    import polyhedral_gravity as gravity_model

    # The list contains as many files as required to define the polyhedron
    potential, acceleration, tensor =
      gravity_model.evaluate(["file.node", "file.face"], density, computation_point)
    potential, acceleration, tensor =
      gravity_model.evaluate(["file.ply"], density, computation_point)
```

Listing 3: Usage example of the Python with file input

Part III.

Results and Conclusion

7. Verification and Results

The implementation has been tested by using the framework *GoogleTest*[1]. Generally, one can divide tests cases into two classes:

- Functionality tests
- Correctness tests

The first category checks the correct functioning of I/O components so that, e.g., the same polyhedron saved in different mesh formats produces the same results. The second category covers the tests verifying that the calculation outputs correct results. These were the key focus of this work. They can be split into the test cases, comparing the results to existing implementations (see Section 7.1) and comparing the results to the analytical solution (see Section 7.2).

7.1. Comparison with existing Implementations

The first introduced test cases compare the produced gravity tensors to the expected output generated by Tsoulis implementations in `FORTRAN` and `MATLAB`. Since numerical equality of only single results would not be enough, additional test cases handling every single property have been implemented. These tests check every single value, like plane unit normals, plane unit normal orientations, etc., ever produced during the total computation for a small cube example, as well as the extensive Eros mesh. These tests ensure that this work's results are consistent with those from the reference implementations. Further, they likewise show that the calculation's intermediate values are correct.

7.2. Comparison with Analytical Solution

The second class of test cases actually only contains one extensive test case, comparing the potential and attraction vectors produced by the polyhedral model to the actual closed analytical solution existing for a cube-shaped body [21]. As evaluating the integral by hand for thousands of points would be cumbersome, the integrals were evaluated using `sympy` and compared to the polyhedral model's output.

Figure 7.1 shows these results produced by the polyhedral model. Generally, one can see slight distortions at the corners of the cube in Figure 7.1c. This result is the expected output as the corners provoke slightly bigger gravitational forces on the surroundings than somewhere in the edge's center, also illustrated by Figure 7.2a. Figure 7.1b depicts the equipotential regions around the cube. If one would place hypothetically water (or any fluid) on the cube's edge, the forming lake would have a spherical shape according to these

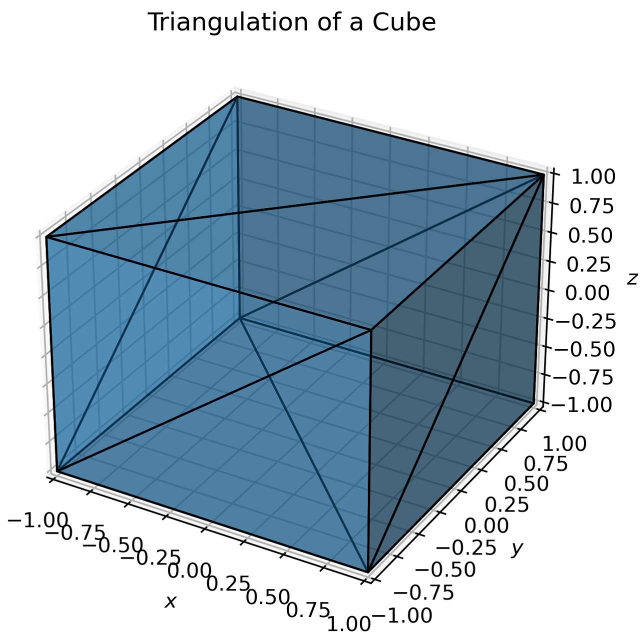
[1] `https://github.com/google/googletest`, last accessed: 02.10.2022

equipotential regions. Figure 7.2b exemplifies this circumstance and shows a second property as well. The lake would be slightly pulled toward the cube's corners due to the additional located mass.

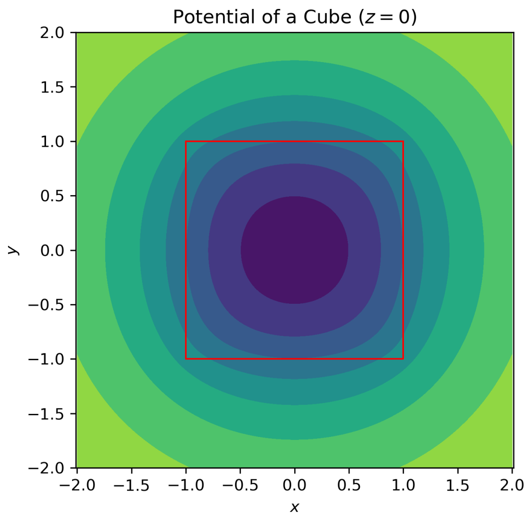
7.3. Other Results: Asteroid Eros and Hypothetical Torus

Figure 7.3 the in Chapter 1 introduced asteroid Eros. It shows its gravitational potential next to its gravitational field of attraction for the XY, XZ, and YZ planes - respectively, a cut-through with one coordinate set to zero. One can observe that, especially near the surface, the shape of Eros matters more than when far away, where the equipotential lines tend to get more circular. The more distant away, the less influence the surface geometry has, and it becomes feasible to approximate the body with a point mass.

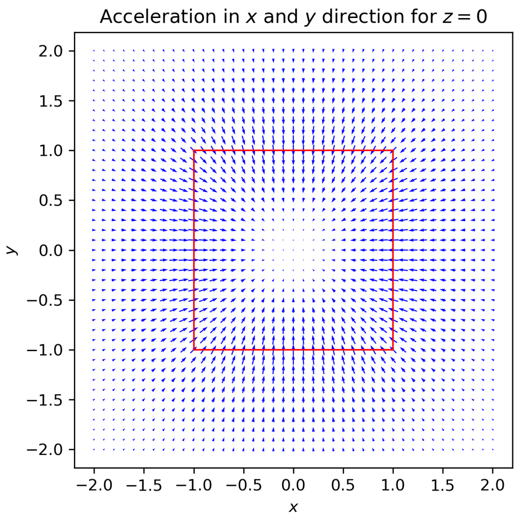
Figure 7.4 depicts a hypothetical Torus-shaped asteroid. This body has a hole in the mid, and as Figure 7.4c shows, a spacecraft would be accelerated towards the ring. In contrast, a spacecraft perfectly placed in the center would not be accelerated in any direction. Further, one can immediately notice the symmetry between the XZ and the XY planes' diagrams.



(a) Mesh of the cube

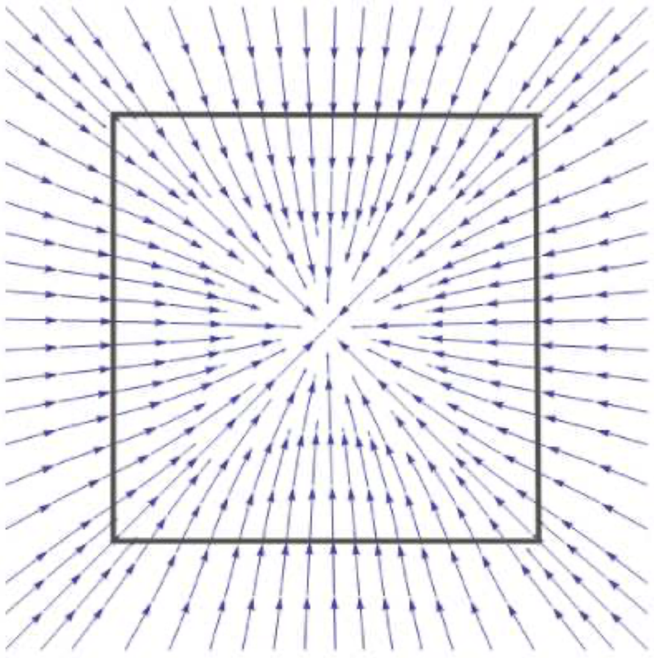


(b) Potential around the cube

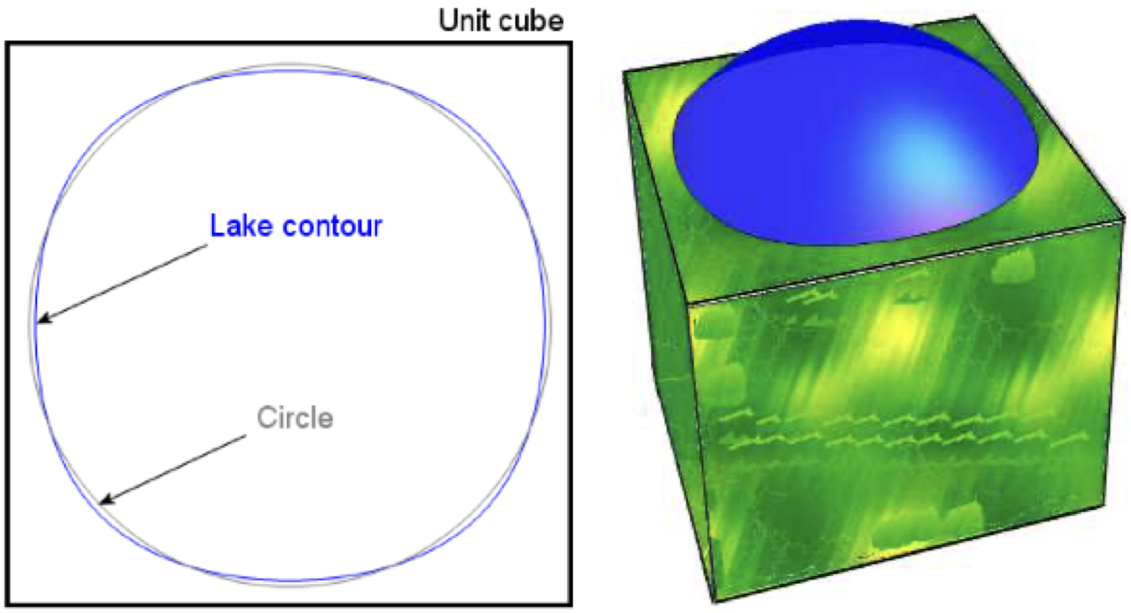


(c) Attraction field for the cube

Figure 7.1.: The model's results for a cube with edge length 2 centered around the origin (0|0|0).

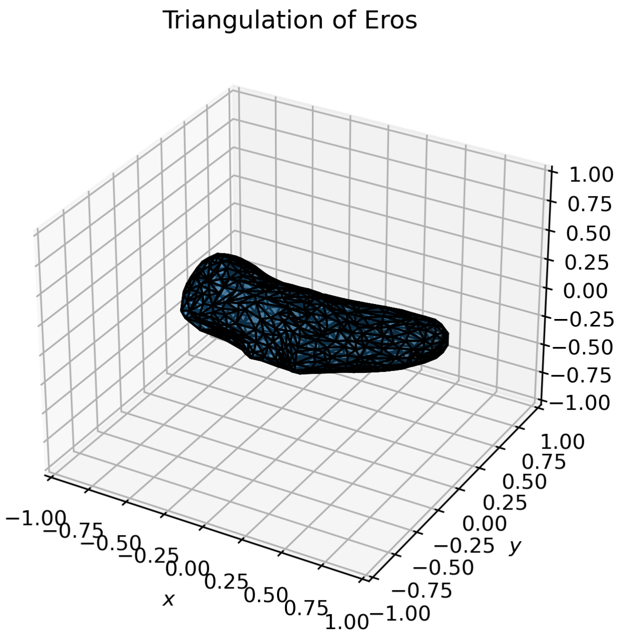


(a) Expected attraction field of the cube

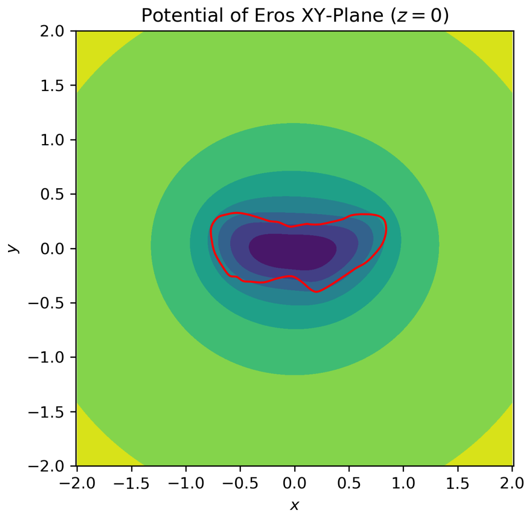


(b) Expected Lake Formation in equipotential regions on the surface

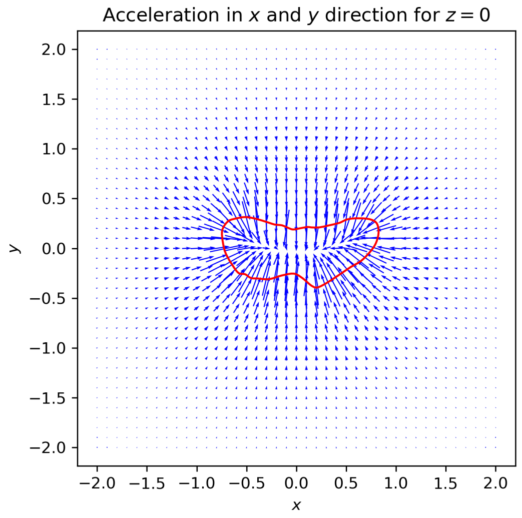
Figure 7.2.: The expected results calculated using the closed analytical formula [21]



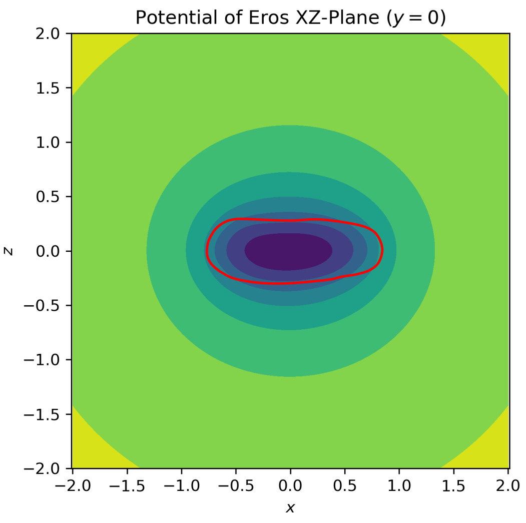
(a) To improve readability, this diagram only shows a lower resolution triangulation of Eros with only 10% of the nodes and vertices compared to the actually used one.



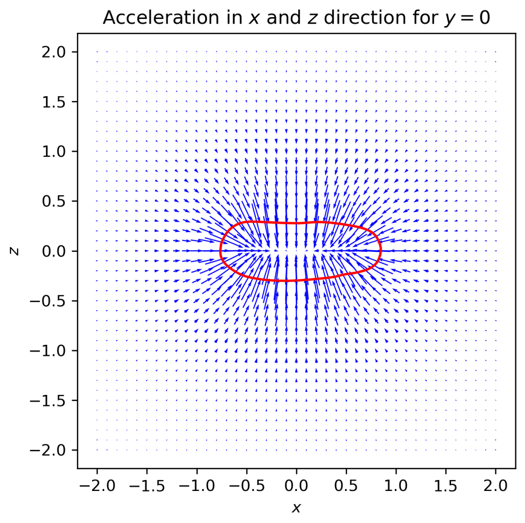
(b) Potential in the XY-Plane



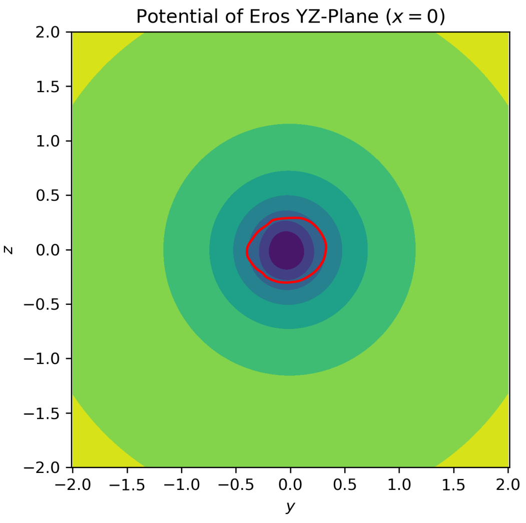
(c) Attraction field in the XY-Plane



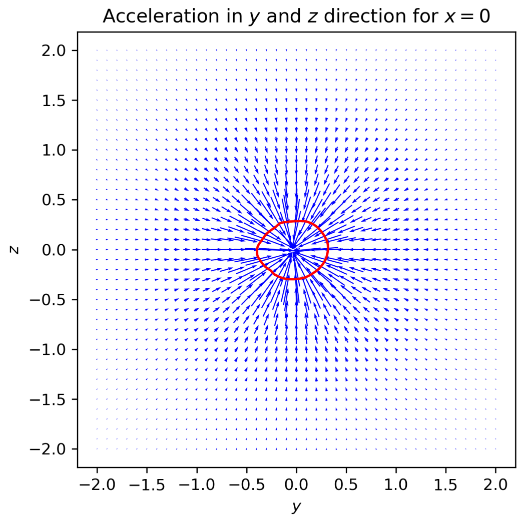
(d) Potential in the XZ-Plane



(e) Attraction field in the XZ-Plane

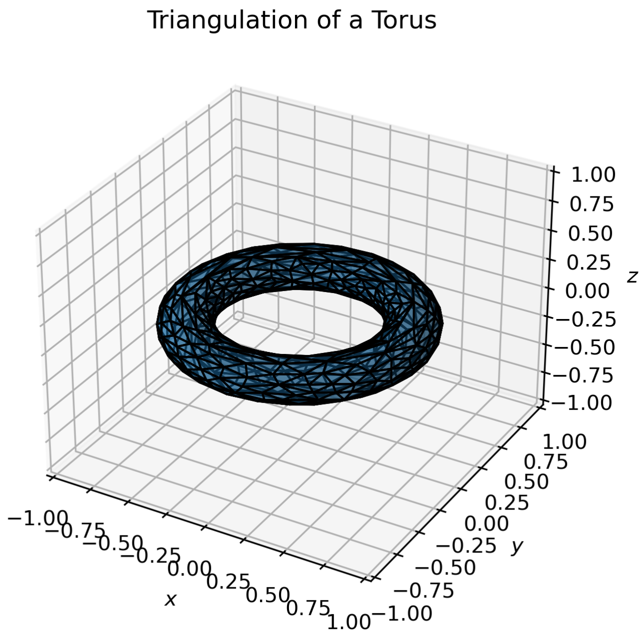


(f) Potential in the YZ-Plane

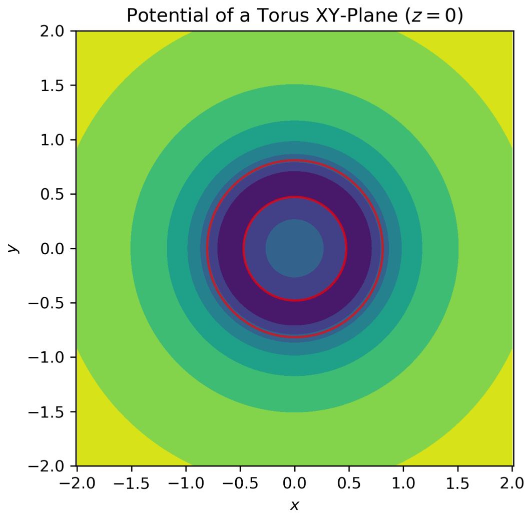


(g) Attraction field in the YZ-Plane

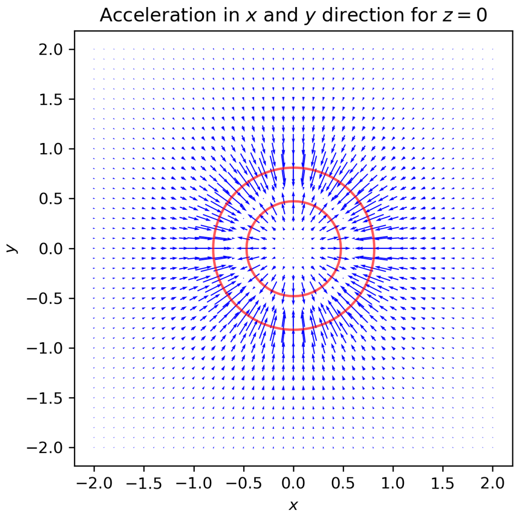
Figure 7.3.: The model's unit-less results for a normed model of the Eros asteroid.



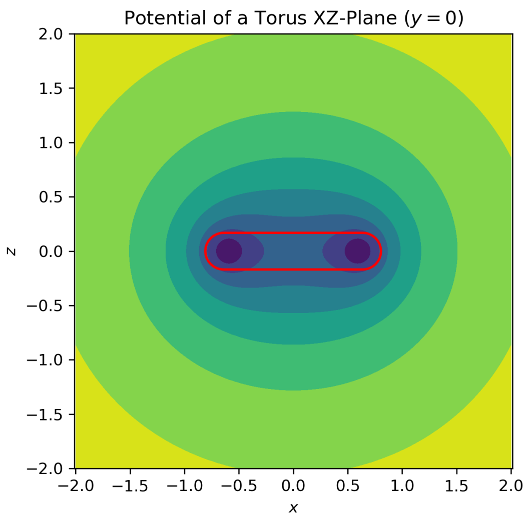
(a) To improve readability, this diagram only shows a lower resolution triangulation of the Torus with only 10% of the nodes and vertices compared to the actually used one.



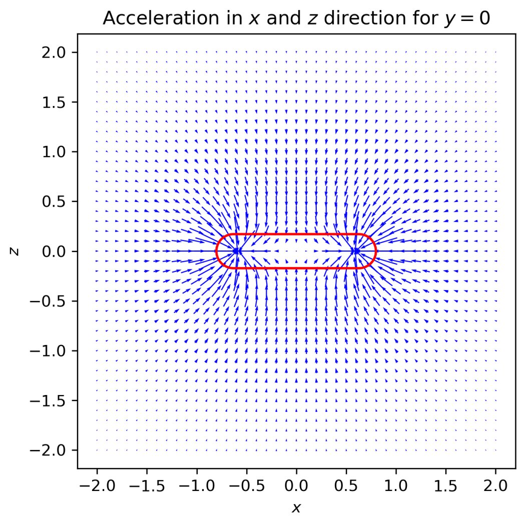
(b) Potential in the XY-Plane



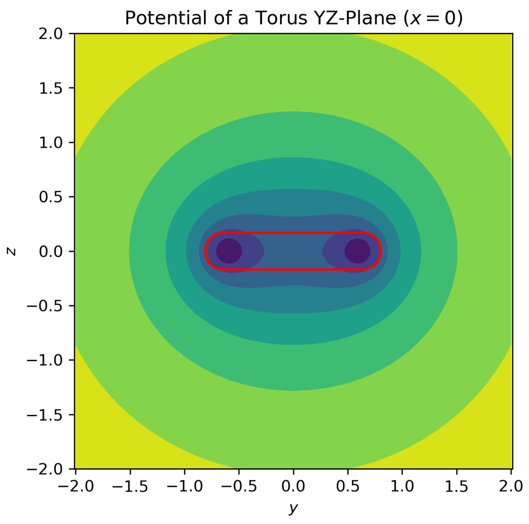
(c) Attraction field in the XY-Plane



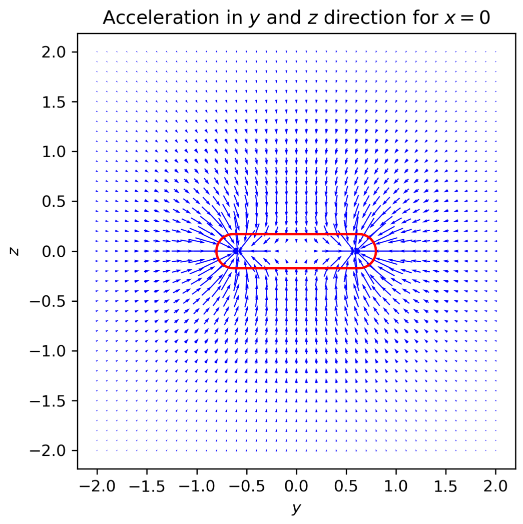
(d) Potential the XZ-Plane



(e) Attraction field in the XZ-Plane



(f) Potential in the YZ-Plane equals the XY-Plane due to symmetry



(g) Attraction field in the YZ-Plane equals the XY-Plane due to symmetry

Figure 7.4.: The model's unit-less results for a normed model of a Torus shaped asteroid.

8. Conclusion

Last, this chapter is going to summarize and give a brief outlook on potential future research.

8.1. Summary

In conclusion, the significant achievement of this work is the creation of a fast parallelized implementation of the polyhedral gravity model. We have chosen to implement the line-integral approach developed by Tsoulis et al., which provides the analytical solution of the full gravity tensor for an arbitrary point in space around a given constant-density polyhedron. The implementation can use a wide variety of mesh file input, which sets it apart from established implementations. Also, the parallelization backend is not fixed but is choosable depending on the user's hardware. Further, the `C++` implementation comes with a slim but powerful `Python` interface deployed on *conda*, making installation a no-brainer. It is worth emphasizing that the implementation compiles and runs on `aarch64`, as well as on `x86_64`, in combination with the most common operating systems: Windows, Linux, and macOS.

It is extensively tested, as presented in Chapter 7. In addition to standard tests, the algorithm has been verified by comparison with Tsoulis' reference implementations and the analytical solution where closed-form solutions exist. The development was executed following software engineering best practices using continuous testing and adding extensive documentation published on *readthedocs*. These properties make this implementation of the polyhedral gravity model more accessible than previous expert programs and ready to be used by the scientific community in future research projects.

8.2. Outlook

The model is successfully deployed on *conda* and ready to be used by anyone interested. So what could be up next? Generally, as the introduction showed, there are millions of potential small bodies where one could apply the model. So, applying the model to different shapes is always an exciting option. Improving the performance of the implementation could also be a valuable objective depending on the use case. The dependencies support *CUDA*. However, the current structure would need a substantial revision in its functional decomposition and utilized data structures. Such change would not necessarily improve code readability but could strongly boost performance.

Another appealing option would be the direct integration of heterogeneous-density polyhedrons via the composition of multiple homogeneous-density polyhedrons. Currently, users would need to do this composition on a higher level on their own. This feature could provide even more flexibility in calculating the gravity tensor, making the utilization more straightforward.

Chapter 1 introduced *geodesyNets* as a new approach using a fully connected neural network to map cartesian coordinates within a body to a corresponding neural density field. In order to generate this field, training data is required, which can be either synthetic or real measured gravity signal. Izzo and Gómez [6] used the *Mascon models* as initial ground truth. The *polyhedral gravity model* could be an interesting alternative to use as ground truth.

Part IV.

Appendix

List of Figures

List of Tables

Bibliography

[1] E. S. Agency, “Asteroiden und Kometen – Millionen Objekte.” `https://www.esa.int/Space_in_Member_States/Germany/Asteroiden_und_Kometen_Millionen_Objekte`, 2010. Accessed: 12.09.2022.

[2] E. S. Agency, “rosetta.” `https://www.esa.int/Science_Exploration/Space_Science/Rosetta`, 2022. Accessed: 12.09.2022.

[3] E. S. Agency, “When CubeSats meet asteroid.” `https://www.esa.int/Space_Safety/Hera/When_CubeSats_meet_asteroid`, 2019. Accessed: 12.09.2022.

[4] S. D. Ross, “Near-earth asteroid mining,” *Space*, pp. 1–24, 2001.

[5] M. Šprlák and S.-C. Han, “On the use of spherical harmonic series inside the minimum brillouin sphere: Theoretical review and evaluation by grail and lola satellite data,” *Earth-Science Reviews*, vol. 222, p. 103739, 2021.

[6] D. Izzo and P. Gómez, “Geodesy of irregular small bodies via neural density fields: geodesynets,” *arXiv preprint arXiv:2105.13031*, 2021.

[7] A. Konopliv, S. Asmar, E. Carranza, W. Sjogren, and D. Yuan, “Recent gravity models as a result of the lunar prospector mission,” *Icarus*, vol. 150, no. 1, pp. 1–18, 2001.

[8] P. T. Wittick and R. P. Russell, “Mascon models for small body gravity fields,” in *AAS/AIAA Astrodynamics Specialist Conference*, vol. 162, pp. 17–162, 2017.

[9] M. Talwani and M. Ewing, “Rapid computation of gravitational attraction of three-dimensional bodies of arbitrary shape,” *Geophysics*, vol. 25, no. 1, pp. 203–225, 1960.

[10] D. Tsoulis, O. Jamet, J. Verdun, and N. Gonindard, “Recursive algorithms for the computation of the potential harmonic coefficients of a constant density polyhedron,” *Journal of Geodesy*, vol. 83, no. 10, pp. 925–942, 2009.

[11] C. Barnett, “Theoretical modeling of the magnetic and gravitational fields of an arbitrarily shaped three-dimensional body,” *Geophysics*, vol. 41, no. 6, pp. 1353–1364, 1976.

[12] V. Pohánka, “Optimum expression for computation of the gravity field of a homogeneous polyhedral body1,” *Geophysical Prospecting*, vol. 36, no. 7, pp. 733–751, 1988.

[13] R. A. Werner, “The gravitational potential of a homogeneous polyhedron or don’t cut corners,” *Celestial Mechanics and Dynamical Astronomy*, vol. 59, no. 3, pp. 253–278, 1994.

[14] R. A. Werner and D. J. Scheeres, “Exterior gravitation of a polyhedron derived and compared with harmonic and mascon gravitation representations of asteroid 4769 castalia,” *Celestial Mechanics and Dynamical Astronomy*, vol. 65, no. 3, pp. 313–344, 1996.

[15] S. Petrović, “Determination of the potential of homogeneous polyhedral bodies using line integrals,” *Journal of Geodesy*, vol. 71, no. 1, pp. 44–52, 1996.

[16] D. Tsoulis and S. Petrović, “On the singularities of the gravity field of a homogeneous polyhedral body,” *Geophysics*, vol. 66, no. 2, pp. 535–539, 2001.

[17] D. Tsoulis, “Analytical computation of the full gravity tensor of a homogeneous arbitrarily shaped polyhedral source using line integrals,” *Geophysics*, vol. 77, no. 2, pp. F1–F11, 2012.

[18] P. J. SCHNEIDER and D. H. EBERLY, *CHAPTER 13 - COMPUTATIONAL GEOMETRY TOPICS*. The Morgan Kaufmann Series in Computer Graphics, San Francisco: Morgan Kaufmann, 2003.

[19] T. Möller and B. Trumbore, “Fast, minimum storage ray/triangle intersection,” in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH ’05, (New York, NY, USA), p. 7–es, Association for Computing Machinery, 2005.

[20] D. Tsoulis and G. Gavriilidou, “A computational review of the line integral analytical formulation of the polyhedral gravity signal,” *Geophysical Prospecting*, vol. 69, no. 8-9, pp. 1745–1760, 2021.

[21] J. M. Chappell, M. J. Chappell, A. Iqbal, and D. Abbott, “The gravitational field of a cube,” *arXiv preprint arXiv:1206.3857*, 2012.