Fast computation of inertia through affinely extended Euler tensor

Antonio DiCarlo [a] and Alberto Paoluzzi [b]

[a] *SMFM@DiS, Università "Roma Tre" Via Vito Volterra, 62 I-00146 Roma, Italy*

[b] *Dipartimento Informatica e Automazione, Università "Roma Tre" Via della Vasca Navale, 79 I-00146 Roma, Italy*

Abstract

We introduce an affine extension of the Euler tensor which encompasses all of the inertia properties of interest in a convenient linear format, and we show how it transforms under affine maps. This result generalizes the standard theorems on the action of rigid transformations (translations and rotations) on inertia properties, allowing for stretch and shear components of the transformation. More importantly, it provides extremely simple and highly efficient computational tools. By these means, a very fast computation of the inertia properties of polyhedral bodies and surfaces may be obtained. The paper contains some mathematical background, a discussion of the state of the art, and a detailed algorithmic description of the new method, that computes and transforms the Euler tensor (strictly related to the inertia) under general affine maps, through addition and multiplication of 4×4 matrices. Evidence is given that the introduced transformation-based computational technique is much faster than conventional domain integration.

Key words: computation of inertia, geometric modeling, Euler tensor
PACS: 02.70, 03.40, 62.20

1 Introduction

In this paper an affine extension of the Euler tensor is introduced, which carries information on the moments of inertia of order 0 to 2 (mass, first moments, second moments and products). It makes it possible to compute the moments

Email addresses: `adicarlo@mac.com`, `paoluzzi@dia.uniroma3.it` (Alberto Paoluzzi).

of inertia transformed by the action of a *general affine map* by means of simple, straightforward matrix multiplications. Our approach provides an explicit, easily computed representation for the transformed inertia properties under mappings that include not only translations and rotations, but also stretches and shears—and even extreme, non-invertible “squashes”. Such a representation could be put to use in, say, automated optimum design of mechanisms, when constraints are imposed on inertia properties and/or the objective function that depends on them. The new method introduced here may be used when computing the inertia moments of a solid body by summing up the elementary inertia moments associated to the cells of a boundary representation of the body. Such operation might be easily parallelized by writing some SIMD [1] code, and using specialized hardware like the current programmable GPUs [2] and the soon coming Cell Processors.

The evaluation of area, volume, centroid and moments of inertia of rigid homogeneous solids frequently arises in a large number of engineering applications, both in Computer-Aided Design and in Robotics. Hence, quadrature formulae for multiple integrals have always been of great interest in computer applications. Many papers on integration methods were related to solid modeling, but—as Lee and Requicha pointed out in [1]—most computational studies in multiple integration deal with calculations over very simple domains, like a cube or a sphere, while the integrating function is very complicated. Conversely, in most of the engineering applications, the opposite problem usually arises. For an early approach to domain integration in solid modeling the reader is referred to Lee and Requicha's paper [1]. Some milestones in the rich engineering literature about volume integration on solid models are given below.

The methods for volume integration introduced by several authors may be considered as a specialization of the Timmer and Stern's general method [2,3], based in turn on change of coordinates and on the repeated application of the divergence theorem, and so reducing to the transformation of a volume integral into a surface integral and then into a line integral. A volume integration over a tetrahedral covering induced by the piecewise affine boundary of a polyhedral domain was given by Lien and Kajiya [4]. Cattani and Paoluzzi [5] proposed boundary integration methods applicable to polyhedra as well as to open 3D polyhedral surfaces and 2D polygons, and proved that the computation of inertia properties is $O(E)$, where E is the number of boundary edges. A generalization of their methods to dimension-independent polyhedra was given by Bernardini [6], who developed both boundary and decompositive algorithms, based on simplicial decompositions of the boundary or of the interior, respectively. Gonzalez-Ochoa, Scott McCammon and Jorg Peters [7]

[1] Single Instruction, Multiple Data.
[2] Graphical Processor Units.

extended the methods in [5] to solids bounded by polynomial surfaces. Their approach completely solves the problem of moments computation over solids bounded by piecewise polynomial surfaces, i.e. the largest class of solids used by PLM systems.

Our main theorem subsumes and generalizes the rotating- and translating-axis theorems of engineering mechanics (see, *e.g.*, [8]) to general affine mappings, which are not even required to be invertible. In its essence, it was formulated and proved about twenty years ago in component form, using standard techniques from computer graphics, *i.e.*, homogeneous coordinates, by one of us (A.P.), while he was visiting the Robotics group at Cornell [9]. Theorem and proof were never submitted to any journal, on the erroneous belief that they were already known and did not deserve further publication. The approach we discuss here has already been widely tested [10] and used to speed up geometric computing applications [11,12]. To the best of our knowledge, until today no similar treatment is documented in the literature or on the web. We further introduce a derived new method to compute the inertia of a surface or a solid, by transforming and summing several instances of the inertia of the standard 2- or 3-simplex.

In this paper, evidence is given that the presented transformational approach to the computation of inertia properties is much faster than direct domain-integration methods. Furthermore, it involves only additions and multiplications of 4×4 matrices, operations that may be easily and efficiently implemented as pixel shaders on the new programmable GPUs. This is done by attaching to each triangle—the actual unit datum of a graphics co-processor—the matrix representation of the corresponding extended Euler tensor, as a 4×4 texture [11]. Our approach may speed up tremendously modeling applications (Boolean operations included) that manage millions of triangles in real time. For computing inertia properties, in particular, our procedure performs either *one order of magnitude* (on RAM models of computation) or *three orders of magnitude* (if implemented on a programmable GPU) better than direct integration [5]. This result is truly impressive: the method we introduce may allow one to compute in real time the inertia properties of *millions* of triangles, instead than thousands.

With respect to the issues of input efficiency, precision of calculations, and robustness of computation, the approach here presented behaves very satisfactorily. (a) The simplest input may be a stream of (coherently oriented) boundary triangles. This kind of input may produce the real-time computation of the inertia of very complex models, if implemented on streaming hardware, e.g. on the graphics coprocessors nowadays available on commodity PCs. (b) The method is finite and exact for 3D polyhedra and polyhedral surfaces, whereas smooth boundaries should be approximated by a boundary triangulation. The precision of the computation is hence directly linked to the precision of the triangulation. (c) The method is not an iterative quadrature, but makes use of closed and very simple formulas, involving product and sum of 4×4 matrices of real numbers. Since the method is not iterative, the precision is not a real concern, and depends only on the precision of the used arithmetics, i.e. on the representation of floats.

The rest of the paper is organized as follows. Section 2 gives some algebraic background and defines the Euler tensor as well as the related inertia tensor in Euclidean three-dimensional space. Section 3 sets forth the notion of Euler tensor, affinely extended to four-dimensional space, shows how it transforms under the action of (extended) affine mappings, and introduces a simple procedure for its fast computation. The closing section discusses the benefits gained and outlines some future developments. The paper is completed by two appendices. Appendix A provides some elementary notions of affine geometry and a geometrical introduction to homogeneous coordinates. Appendix B illustrates a pseudocode implementation of the introduced approach and contrasts its performance with that of more conventional algorithms.

2 Background

2.1 Double tensors and tensor product of two vectors

Tensors An nth-order tensor is a multilinear scalar-valued operator. In a d-dimensional environment, it can be represented by d^n components, decorated with n indices and organized in a $d \times d \times \ldots \times d$ (n times) hypermatrix. In this paper we need to consider only tensors of order 2, also named *double* tensors, identified with linear applications of the underlying Euclidean vector space into itself. We employ here the term *tensor* as synonym of *linear application* of a vector space $\mathcal{V}$ into itself. In other words, a tensor $\mathbf{L} \in \mathrm{Lin}\,(\mathcal{V})$ is an application that maps *linearly* the vector $\mathbf{u} \in \mathcal{V}$ to the vector $\mathbf{Lu} \in \mathcal{V}$.

Tensor transposition Given $\mathbf{L} \in \mathrm{Lin}\,(\mathcal{V})$, its *transpose* is the (unique) tensor, denoted $\mathbf{L}^\top$, such that, for all $\mathbf{u}, \mathbf{v} \in \mathcal{V}$, $(\mathbf{L}\,\mathbf{u})\cdot\mathbf{v} = \mathbf{u}\cdot(\mathbf{L}^\top\mathbf{v})$, the Euclidean inner product in $\mathcal{V}$ being denoted as a dot product, as usual.

Tensor product of vectors The *tensor product* of two vectors $\mathbf{a}, \mathbf{b} \in \mathcal{V}$ is the tensor $\mathbf{a} \otimes \mathbf{b} \in \mathrm{Lin}\,(\mathcal{V})$ that maps each vector $\mathbf{u} \in \mathcal{V}$ to the vector $(\mathbf{a} \cdot \mathbf{u})\mathbf{b}$. Transposition of a tensor product swaps the two vectors: $(\mathbf{a} \otimes \mathbf{b})^\top = \mathbf{b} \otimes \mathbf{a}$. We note for further use that [3]

$$\mathbf{a}\otimes(\mathbf{L}\,\mathbf{b}) = \mathbf{L}\,(\mathbf{a}\otimes\mathbf{b}) \quad \text{and} \quad (\mathbf{L}\,\mathbf{a})\otimes\mathbf{b} = (\mathbf{a}\otimes\mathbf{b})\mathbf{L}^{\top}. \tag{1}$$

Orthogonal decomposition of vectors Let $\mathbf{e}, \mathbf{u} \in \mathcal{V}$, with $\mathbf{e}$ a unit vector (*i.e.*, s.t. $\mathbf{e}\cdot\mathbf{e} = 1$). The tensors $\mathbf{e}\otimes\mathbf{e}$ and $\mathbf{I}-\mathbf{e}\otimes\mathbf{e}$ ($\mathbf{I}$ being the identity on $\mathcal{V}$) map $\mathbf{u}$ to its orthogonal projections into the direction spanned by $\mathbf{e}$ and its orthogonal complement, respectively: $(\mathbf{e}\otimes\mathbf{e})\mathbf{u} = (\mathbf{e}\cdot\mathbf{u})\mathbf{e}$, and $(\mathbf{I}-\mathbf{e}\otimes\mathbf{e})\mathbf{u} = \mathbf{u}-(\mathbf{e}\cdot\mathbf{u})\mathbf{e}$.

2.2 Euler and inertia tensors in three-space

Let $\mathcal{B}$ be a body in $\mathbb{E}^3$. In mechanics, the *Euler tensor* about $\mathbf{o}$ is defined as:

$$\mathbf{E} := \int_{\mathcal{B}} \mathbf{r}\otimes\mathbf{r}\, dM, \tag{2}$$

where $\mathbf{r} = \mathbf{x}-\mathbf{o}$ is the radius vector of the general point $\mathbf{x}\in\mathcal{B}$ with respect to the origin $\mathbf{o}\in\mathbb{E}^3$, and M is the *mass measure* supported by $\mathcal{B}$ (see, *e.g.*, [14]). The *inertia tensor* is then introduced as: [4]

$$\mathbf{J} := \int_{\mathcal{B}} \left(|\mathbf{r}|^2\,\mathbf{I} - \mathbf{r}\otimes\mathbf{r}\right) dM = (\operatorname{tr}\mathbf{E})\,\mathbf{I} - \mathbf{E}, \tag{3}$$

where $|\mathbf{r}|^2 := \mathbf{r}\cdot\mathbf{r}$. Notice that, while more familiar to many, the inertia tensor—contrary to the Euler tensor—only belongs in three-space, its very definition being motivated by the fact that planes (*i.e.*, two-dimensional subspaces) have codimension one. This is why $\mathbf{E}$, not $\mathbf{J}$, is to be extended to $\mathbb{E}^4$.

3 Extended Euler tensor

Let $\mathcal{B}$ be a regular, *i.e.*, uniformly d-dimensional subset of $\mathbb{E}^3$, which we now embed into a Euclidean 4-dimensional affine space $\mathbb{E}^4$, parametrized by a Cartesian frame $(\mathbf{o}_+, (\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3; \mathbf{k}))$, with $\mathbf{o}_+ = \mathbf{o}-\mathbf{k}$, so that $\mathbf{o}_+ \notin \mathbb{E}^3$, and $\mathbf{x}\in\mathbb{E}^3 \Leftrightarrow (\mathbf{x}-\mathbf{o}_+)\cdot\mathbf{k} = 1$. In other words, if the general point $\mathbf{x}\in\mathbb{E}^4$ has coordinates $(x_1, x_2, x_3; w)$, points in $\mathbb{E}^3$ are characterized by the property $w=1$.

[3] Use that $(\mathbf{a}\otimes(\mathbf{L}\,\mathbf{b}))\,\mathbf{u} = (\mathbf{a}\cdot\mathbf{u})(\mathbf{L}\,\mathbf{b}) = \mathbf{L}\,((\mathbf{a}\cdot\mathbf{u})\,\mathbf{b}) = \mathbf{L}\,((\mathbf{a}\otimes\mathbf{b})\,\mathbf{u}) = (\mathbf{L}\,(\mathbf{a}\otimes\mathbf{b}))\,\mathbf{u}$ for all $\mathbf{u}\in\mathcal{V}$ to prove the first identity. Follow the same lines for the second one.

[4] The *trace* tr is the (unique) linear map on $\operatorname{Lin}(\mathcal{V})$ s.t., $\forall\,\mathbf{a}, \mathbf{b}\in\mathcal{V}, \operatorname{tr}(\mathbf{a}\otimes\mathbf{b}) = \mathbf{a}\cdot\mathbf{b}$.

Definition 3-1. The *extended Euler tensor* $\mathbf{E}_+$ is defined with respect to $\mathbf{o}_+$ exactly as the Euler tensor $\mathbf{E}$ is defined with respect to $\mathbf{o}$ in Equation (2):

$$\mathbf{E}_+ := \int_{\mathcal{B}} \mathbf{r}_+ \otimes \mathbf{r}_+ \, dM, \tag{4}$$

where $\mathbf{r}_+ := \mathbf{x} - \mathbf{o}_+ = \mathbf{r} + \mathbf{k}$ is the radius vector w.r.t. $\mathbf{o}_+$. The Euler tensor is readily recovered from $\mathbf{E}_+$, since $\mathbf{E} = \mathbf{P}\,\mathbf{E}_+\mathbf{P}$, with $\mathbf{P}$ the orthogonal projection along $\mathbf{k}$, *i.e.*, $\mathbf{P} = \mathbf{I} - \mathbf{k} \otimes \mathbf{k}$ (see the coordinate representation below). The inertia tensor $\mathbf{J}$ may then be computed from $\mathbf{E}$ through Equation (3).

Definition 3-2 (Coordinate representation). Using the *structure products* [5]

$$I\,(i, j, h; k) := \int_{\mathcal{B}} x^i y^j z^h w^k \, dM,$$

where x, y, z stand for x_1, x_2, x_3, respectively, the following matrix representation of the extended Euler tensor is obtained:

$$[\,\mathbf{E}_+] = \int_{\mathcal{B}} \begin{pmatrix} x^2 & xy & xz & xw \\ xy & y^2 & yz & yw \\ xz & yz & z^2 & zw \\ xw & yw & zw & w^2 \end{pmatrix} dM,$$

or, performing the integration within the array, as:

$$[\,\mathbf{E}_+] = \begin{pmatrix} I\,(2,0,0;0) & I\,(1,1,0;0) & I\,(1,0,1;0) & I\,(1,0,0;1) \\ I\,(1,1,0;0) & I\,(0,2,0;0) & I\,(0,1,1;0) & I\,(0,1,0;1) \\ I\,(1,0,1;0) & I\,(0,1,1;0) & I\,(0,0,2;0) & I\,(0,0,1;1) \\ I\,(1,0,0;1) & I\,(0,1,0;1) & I\,(0,0,1;1) & I\,(0,0,0;2) \end{pmatrix},$$

where *second moments* appear on the main diagonal, while off-diagonal terms are *products of inertia*. Since $w = 1$ all over $\mathcal{B}$, the off-diagonal terms in the fourth row or column provide the *first moments* of inertia, and the fourth diagonal term the *mass* $M(\mathcal{B})$.

Our main result provides a straightforward way of computing the extended Euler tensor $\mathbf{E}_+^\star$ of the image $\mathcal{B}^\star := \mathbf{A}(\mathcal{B})$ of $\mathcal{B}$ under a *general* affine map $\mathbf{A}$, once the extended Euler tensor $\mathbf{E}_+$ of $\mathcal{B}$ itself is known. Hence we obtain a fast computation of $\mathbf{E}^\star$ (and of $\mathbf{J}^\star$, if desired).

Theorem 3-3. *Under the hypothesis that the action of the affine map $\mathbf{A}$ on the mass measure M is such that the density of $M^\star := \mathbf{A}(M)$ w.r.t. M, namely*

$dM^\star/dM$, is constant *over $\mathcal{B}$, the extended Euler tensor transforms as follows:*

$$\mathbf{E}_+^\star = \tfrac{dM^\star}{dM}\,\mathbf{L}_+\,\mathbf{E}_+\,\mathbf{L}_+^\top\,, \tag{5}$$

where $\mathbf{L}_+$ is the linear part of the unique *affine extension of $\mathbf{A}$ to $\mathbb{E}_4$ that keeps $\mathbf{o}_+$ fixed: $\mathbf{A}_+(\mathbf{o}_+) = \mathbf{o}_+$.* [5]

Proof: Let $\mathbf{x}^\star := \mathbf{A}_+(\mathbf{x})$ and $\mathbf{r}_+^\star := \mathbf{x}^\star - \mathbf{o}_+$. Since $\mathbf{A}_+(\mathbf{o}_+) = \mathbf{o}_+$, $\mathbf{r}_+^\star = \mathbf{L}_+\mathbf{r}_+$ (see Equation (A.1)). Therefore,

$$\mathbf{E}_+^\star := \int_{\mathcal{B}} \mathbf{r}_+^\star \otimes \mathbf{r}_+^\star \, dM^\star = \int_{\mathcal{B}} \tfrac{dM^\star}{dM} (\mathbf{L}_+\mathbf{r}_+) \otimes (\mathbf{L}_+\mathbf{r}_+)\, dM.$$

Then, the two identities (1) plus the hypothesis that $dM^\star/dM$ is constant over $\mathcal{B}$ deliver the result:

$$\mathbf{E}_+^\star = \tfrac{dM^\star}{dM}\,\mathbf{L}_+ \left(\int_{\mathcal{B}} \mathbf{r}_+ \otimes \mathbf{r}_+\, dM \right) \mathbf{L}_+^\top = \tfrac{dM^\star}{dM}\,\mathbf{L}_+\,\mathbf{E}_+\,\mathbf{L}_+^\top\,. \qquad \square$$

Remark 3-4. Note that in Equation (5) it is the density of the transformed mass $M^\star$ w.r.t. the original mass M that matters, not the density of either of them w.r.t. the (signed [6]) *volume* measure V. Equation (5) works fine even if density of mass w.r.t. volume does not exist. However, in the very special case $M = M^\star = V$, one has $dM^\star/dM = \det(\mathbf{L}_+) = \det(\mathbf{L})$. This establishes a connection with the notion of *tensor densities* [15].

Remark 3-5. It should be emphasized that $\mathbf{A}$, and hence $\mathbf{L}_+$, is *not* required to have an inverse for Equation (5) to hold. In fact, think of a cube on which $M = V$, and let $\mathbf{A}$ be the affine map that squashes it into one of its square faces. Of course, its linear part has zero determinant and no inverse. If the mass density w.r.t. volume is assumed to be unaffected by $\mathbf{A}$, then $dM^\star/dM = 0$, and Equation (5) yields $\mathbf{E}_+^\star = \mathbf{0}$. If, on the contrary, $\mathbf{A}$ squeezes all of the mass of the cube uniformly into the target square, then $dM^\star/dM = 1$ and $\mathbf{E}_+^\star \neq \mathbf{0}$.

Example Let $\mathcal{B}$ be a surface embedded in $\mathbb{E}^3$. Equation (5) can be used to get a fast computation of $\mathbf{E}_+(\mathcal{B}) = \sum_{\boldsymbol{\tau}} \mathbf{E}_+(\boldsymbol{\tau})$, where $\boldsymbol{\tau} = \operatorname{conv}(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2)$ [7] is the general element of a triangulation of $\mathcal{B}$. Let us also assume that the mass measure supported by $\mathcal{B}$ has a piecewise constant density ρ with respect to the area measure A, namely constant within each triangle $\boldsymbol{\tau}$. Let us take

[5] Given $\mathbf{A}$, its distinguished extension $\mathbf{A}_+$ is explicitly constructed in Appendix A.
[6] You want to keep track of the orientation in order to be able to *substract* parts.
[7] We write $\operatorname{conv}(\dots)$, instead of $\operatorname{conv}\{\dots\}$, to stress that vertex *ordering* matters.

$\mathbf{\Delta}_2 := \mathrm{conv}\,(\mathbf{o}, \mathbf{o}+\mathbf{e}_1, \mathbf{o}+\mathbf{e}_2)$ as the prototype 2-simplex in $\mathbb{E}^3$ and assume that its mass and area measures coincide: $M = A$. Then, equation (5) reads

$$\mathbf{E}_+(\boldsymbol{\tau}) = \rho \, \det(\mathbf{L}_+)\, \mathbf{L}_+ \, \mathbf{E}_+(\mathbf{\Delta}_2)\, \mathbf{L}_+^\top ,$$

where $\mathbf{L}_+$ is characterized by the requirements that $\mathbf{A}(\mathbf{\Delta}_2) = \boldsymbol{\tau}$ and the unit normal to $\mathbf{\Delta}_2$ be mapped into the unit normal to $\boldsymbol{\tau}$, which imply:

$$\mathbf{L}_+\, \mathbf{e}_1 = \mathbf{x}_1 - \mathbf{x}_0 , \quad \mathbf{L}_+\, \mathbf{e}_2 = \mathbf{x}_2 - \mathbf{x}_0 , \quad \mathbf{L}_+^\top \mathbf{L}_+\, \mathbf{e}_3 = \mathbf{e}_3 , \quad \mathbf{L}_+\, \mathbf{k} = \mathbf{x}_0 - \mathbf{o}_+ .$$

The extended Euler tensor of the prototype $\mathbf{\Delta}_2$ is computed once and for all making use of the structure products given in [5], which provide the result:

$$[\,\mathbf{E}_+(\mathbf{\Delta}_2)\,] = \begin{pmatrix} I_2(2,0) & I_2(1,1) & 0 & I_2(1,0) \\ I_2(1,1) & I_2(0,2) & 0 & I_2(0,1) \\ 0 & 0 & 0 & 0 \\ I_2(1,0) & I_2(0,1) & 0 & I_2(0,0) \end{pmatrix} = \frac{1}{24} \begin{pmatrix} 2 & 1 & 0 & 3 \\ 1 & 2 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 3 & 3 & 0 & 12 \end{pmatrix},$$

where, from [5],

$$I_2(i,j) := \int_{\mathbf{\Delta}_2} x^i y^j dS = \frac{1}{i+1} \sum_{k=0}^{i+1} \binom{i+1}{k} \frac{(-1)^k}{k+j+1} . \tag{6}$$

4 Computational procedure

In this section we show how to compute the extended Euler tensor for a solid body using the method introduced in the previous section. The approach is implemented here using a mathematical pseudocode. The first prototype was developed in `PLaSM`, a functional design language [16,10] for rapid prototyping of geometric algorithms and applications [17]. An implementation of the same computation for surfaces is given in Appendix B.

4.1 Fast computation of inertia properties of solids

To focus on the essentials, let us assume that mass and volume measures coincide. With a view to computing the extended Euler tensor $\mathbf{E}_+(\mathcal{B})$ of a general solid body $\mathcal{B} \subset \mathbb{E}^3$, we compute first the extended Euler tensor $\mathbf{E}_+(\mathbf{\Delta}_3)$ of the prototype 3-simplex $\mathbf{\Delta}_3 := \mathrm{conv}\,(\mathbf{o}, \mathbf{o}+\mathbf{e}_1, \mathbf{o}+\mathbf{e}_2, \mathbf{o}+\mathbf{e}_3)$. Then, we use formula (5) repeatedly to transform $\mathbf{E}_+(\mathbf{\Delta}_3)$ into $\mathbf{E}_+(\boldsymbol{T}_{\boldsymbol{\tau}})$, where $\boldsymbol{T}_{\boldsymbol{\tau}} := \mathrm{conv}\,(\mathbf{o}, \boldsymbol{\tau})$ is the $\mathbf{o}$-based tetrahedron associated with the general element $\boldsymbol{\tau}$ of a triangulation of the boundary of $\mathcal{B}$, summing up all contributions:

$$\boxed{\mathbf{E}_+(\mathcal{B}) = \textstyle\sum_{\boldsymbol{\tau}\in\partial\mathcal{B}} \mathbf{E}_+(\boldsymbol{T}_{\boldsymbol{\tau}})} \tag{7}$$

Remark 4-1. The contributions from the intersections $\{\boldsymbol{T}_{\boldsymbol{\tau}} \backslash \mathcal{B}, \boldsymbol{\tau} \in \partial\mathcal{B}\}$, *i.e.*, the parts of the generated tetrahedra that fall *outside* $\mathcal{B}$, properly cancel out in the above summation. This fact is built into the integration procedure, since tetrahedral integrals get signed according to the visibility from $\mathbf{o}$ of the internal normal to the τ triangle.

```
Def  $Euler(\mathrm{conv}\,\{\mathbf{o}, \mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\} \subset \mathbb{E}^3) = \det(\mathbf{G})\ \mathbf{G}\ \mathbf{E}_+\ \mathbf{G}^T$
where
  $\mathbf{G} := \begin{pmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{pmatrix}, \mathbf{E}_+ := \frac{1}{120}\begin{pmatrix} 2 & 1 & 1 & 5 \\ 1 & 2 & 1 & 5 \\ 1 & 1 & 2 & 5 \\ 5 & 5 & 5 & 20 \end{pmatrix}$
end;
```

Fig. 1. Extended Euler tensor associated with a boundary triangle

The pseudocode in Figure 3 computes the tensor $\mathbf{J} = inertia(\Sigma)$, via equation (3), from a boundary triangulation $\Sigma := \{\boldsymbol{\tau}\}$ of $\partial\mathcal{B}$.

```
Def  $inertia(\Sigma) = \mathrm{tr}(\mathbf{E}_{3\times3})\ \mathbf{I} - \mathbf{E}_{3\times3}$
where
  $\mathbf{E}_+(\mathcal{B}) := \sum_{\boldsymbol{\tau}\in\Sigma} Euler(\boldsymbol{\tau}),$
  $\mathbf{E}_{3\times3}$ := upper-diagonal submatrix of $\mathbf{E}_+(\mathcal{B})$
end;
```

Fig. 2. Inertia of a solid computed from a triangulation of its boundary

4.2 Sample computations

A Cartesian frame is now taken for granted, and $\mathbb{E}^3$ briefly identified with $\mathbb{R}^3$.

Boundary triangulation of the standard cube An explicit triangulation of the boundary $\partial\boldsymbol{C}_3$ of the standard cube $\boldsymbol{C}_3 \subset \mathbb{R}^3$ is given as a pseudocode list of triangles. Each one of the 12 triangles is given as a triple of points, where the representation of a point is the list of its coordinates.

```
**Def** $cube\_boundary := (\ \boldsymbol{\tau}_1, \boldsymbol{\tau}_2, \ldots, \boldsymbol{\tau}_{12}\ )$
**where**
    $\boldsymbol{\tau}_1 := ((0,0,0),(1,1,0),(1,0,0)),\ \boldsymbol{\tau}_2 := ((0,0,0),(0,1,0),(1,1,0)),$
    $\boldsymbol{\tau}_3 := ((1,0,0),(1,0,1),(0,0,1)),\ \boldsymbol{\tau}_4 := ((1,0,0),(0,0,1),(0,0,0)),$
    $\boldsymbol{\tau}_5 := ((0,0,0),(0,0,1),(0,1,0)),\ \boldsymbol{\tau}_6 := ((0,1,0),(0,0,1),(0,1,1)),$
    $\boldsymbol{\tau}_7 := ((0,0,1),(1,0,1),(1,1,1)),\ \boldsymbol{\tau}_8 := ((1,1,1),(0,1,1),(0,0,1)),$
    $\boldsymbol{\tau}_9 := ((1,1,0),(0,1,1),(1,1,1)),\ \boldsymbol{\tau}_{10} := ((1,1,0),(0,1,0),(0,1,1)),$
    $\boldsymbol{\tau}_{11} := ((1,0,0),(1,1,0),(1,0,1)),\ \boldsymbol{\tau}_{12} := ((1,1,0),(1,1,1),(1,0,1))$
**end**;
```

Fig. 3. Triangulation of the boundary of the standard unit cube.

Extended Euler tensor of the standard cube The pseudocode expression on l.h.s. of Figure 4 returns the 4×4 matrix $[\mathbf{E}_+(\boldsymbol{C}_3)]$, shown on r.h.s.

$$Euler(\mathcal{B}) := \sum_{\boldsymbol{\tau} \in cube\_boundary} Euler(\mathrm{conv}\,(\mathbf{o}, \boldsymbol{\tau})) = \frac{1}{12}\begin{pmatrix} 4 & 3 & 3 & 6 \\ 3 & 4 & 3 & 6 \\ 3 & 3 & 4 & 6 \\ 6 & 6 & 6 & 12 \end{pmatrix}$$

Fig. 4. Computation of the extended Euler tensor of the standard unit cube.

Extended Euler tensor of a rotated unit cube Let the standard cube be rotated by $\pi/4$ about the axis $(1, -1, 0)$ through the origin. According to Theorem 3-3, the transformed (extended) Euler tensor may be obtained by evaluating the expression

$$Euler(\mathbf{Q}(\mathcal{B})) := \det(\mathbf{Q})\ \mathbf{Q}\ Euler(\mathcal{B})\ \mathbf{Q}^\top,$$

where $\mathbf{Q}$ is given by the pseudocode in Figure 6, and $\det(\mathbf{Q}) = 1$ since $\mathbf{Q}$ is a rotation. The matrix of the (extended) Euler tensor of rotated cube is computed in Figure 6 by summing the contributions of the (transformed) boundary triangles, by using the pseudocode given in Figure 5. The two results clearly coincide.

$$Euler(\mathbf{Q}(\mathcal{B})) := \sum_{\boldsymbol{\tau} \in cube\_boundary} Euler(\mathrm{conv}\,(\mathbf{o}, \mathbf{Q}(\boldsymbol{\tau})))$$

where

$$\mathbf{Q} := \mathbf{R}_z(-\tfrac{\pi}{4})\ \mathbf{R}_x(\tfrac{\pi}{4})\ \mathbf{R}_z(\tfrac{\pi}{4}),$$

$$\mathbf{R}_x(\alpha) := \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix},\ \mathbf{R}_z(\alpha) := \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

end;

Fig. 5. Computation of the extended Euler tensor of the rotated unit cube.

$$\begin{pmatrix} 0.09405663803669642 & 0.010723304703363204 & 0.08838834764831871 & 0.10355339059327409 \\ 0.010723304703363204 & 0.09405663803669642 & 0.08838834764831872 & 0.1035533905932741 \\ 0.08838834764831868 & 0.08838834764831871 & 0.8118867239266054 & 0.8535533905932726 \\ 0.10355339059327409 & 0.1035533905932741 & 0.8535533905932726 & 0.9999999999999991 \end{pmatrix}$$

Fig. 6. Extended Euler tensor of the rotated unit cube.

4.3 Cow example

The algorithm for fast computation of the Euler tensor discussed in this paper has been used in other projects. In particular, it constitutes the computational core of our parallel streaming design evaluation [12] with progressive generation of solid models starting from boundary triangulations [11]. In this case a *balanced* BSP-tree and cell decomposition are generated after an $O(n)$ preprocessing that computes the Euler tensor for each input triangle. In Figure 7 the solid cell decomposition for a cow model is shown.

The idea of algorithm [11] is very simple. A fast preprocessing, executed in linear time with the number of input triangles, computes for each triangle the 4x4 extended Euler matrix that codifies numerically its mechanical behavior. The sum of all such matrices gives a good representation of the spatial distribution of the surface points. Such a matrix may be used used to generate a best-fitting ellipsoid, centered in the center of mass of the surface, that may be considered mechanically equivalent to the triangulated surface.

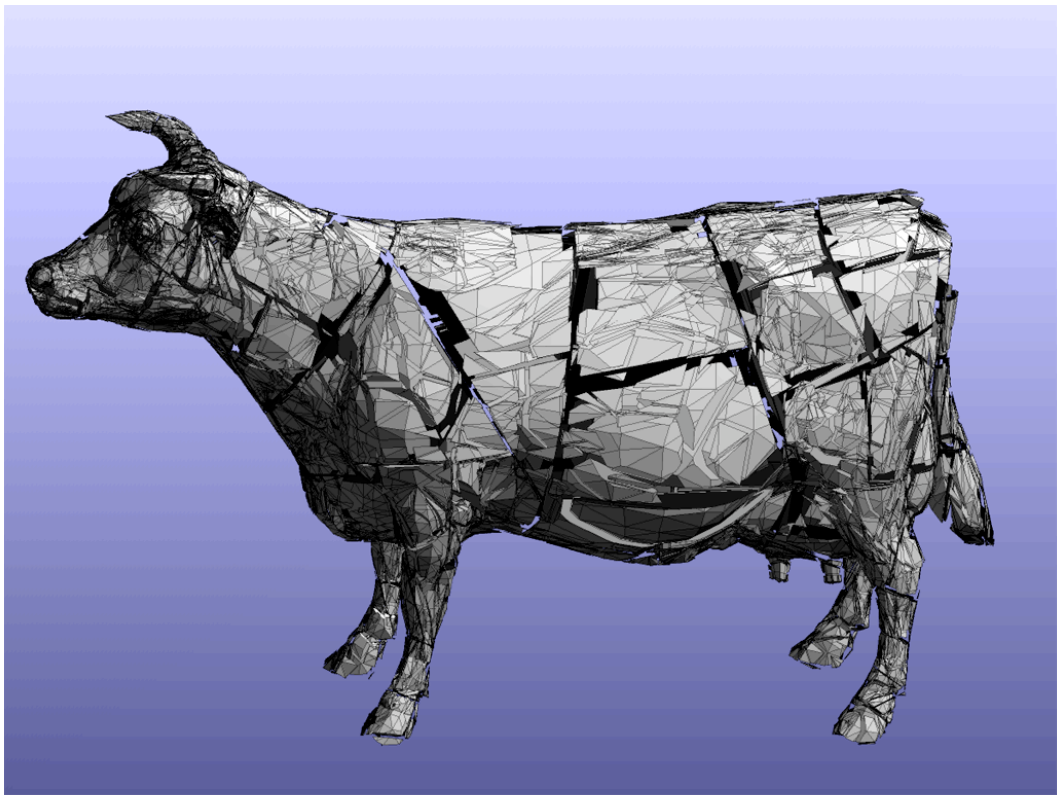


Fig. 7. Exploded view of the balanced BSP solid model generated from a boundary triangulation, and using our fast computation of the Euler tensor.

The first solid approximation to the surface is given by the smallest best-fitting parallelepiped, which has edges parallel to the principal axes of the ellipsoid and contains the surface. Such a solid is represented as a standard BSP-tree, as the intersection of six linear subspaces tangent to the boundary surface. This enclosing solid is then split by a plane through the center of mass and normal to the direction of maximal elongation. The set of input triangles is in turn split into two subsets, according of whether they are above or below this dividing plane, and the two subsets are associated to the below and above sub-trees of the BSP root. The confinement of each surface subset into a narrower and properly rotated best-fitting parallelepiped, and its splitting into the principal direction are recursively repeated for each sub-tree.

The Euler tensor of the model in Figure 7 was computed in 0.17 seconds, on a triangulation with 5804 triangles, using an Intel Centrino 2.00Ghz with 1.047.784 KB of RAM and MS Visual C++ 6.0. The sustained average obtained on several, even much bigger, models is slightly greater than 35,000 triangles/sec, for a total number of more than 350,000 integrals/sec of a low-degree scalar field over a triangular domain.

5 Performance evaluation

In this section we compare the relative performance of the computation of the (matrix representation of the) Euler tensor for a single triangle in $\mathbb{E}^3$, since the inertia properties of either a surface or a body in 3D are obtained by summing up such matrices over a boundary triangulation. We show that computations based on Equation (5) are more than *one order of magnitude* faster than those based on formulæ for direct integration on triangles in $\mathbb{E}^3$, as given in [5].

From the implementation documented in Listing B.2 given in Appendix B, one can see that the direct computation of the Euler tensor for a triangle in $\mathbb{E}^3$ requires 55 tuples of values for the variables used in the integration formula. Without code optimizations, 9 powers with integer exponent, 14 products, 6 binomial coefficients and 11 additions or differences are to be computed for each tuple. An optimistic estimate of the computation size provides therefore a lower bound of $55 \times 40 = 2200$ floating-point operations per triangle. Several such operations are not actually elementary, but they can be approximately considered to be so after some code optimization.

To compare, consider that the fast computation based on Equation (5), as implemented in Figure B.3 given in Appendix B, requires: (i) 2 multiplications of 4×4 matrices, for a total of $2 \times 16 \times 7 = 224$ real operations; (ii) the computation of the matrix representation of the affine map $\mathbf{G}$, that requires 2×3 subtractions and 3×3 multiplications and subtractions to be assembled, for a total of 15 algebraic operations; (iii) $\det(\mathbf{G})$ actually consists in a 3×3 determinant, for another 17 operations. But, considering (a) the special structure of the term *Euler2D* (one row and one column zero), (b) the structure of $\mathbf{G}$ (one unit row) and (c) the fact that the result is a symmetric matrix (only $(4 \times 5)/2 = 10$ strict components have to be computed), the double product of matrices globally requires 96 algebraic operations, instead than 224. The whole computation hence requires $96 + 15 + 17 = 128$ floating-point serial operations per triangle. This proves our claim that the new method performs *one order of magnitude* better than direct integration [5], using the RAM model of computation.

If the computation of the extended Euler tensor is implemented as a pixel shader on the new programmable GPUs, then the new method is globally faster by *three orders of magnitude*, due to the specialized hardware supporting parallel product and parallel sum of 4×4 matrices and 4-vectors. As a matter of fact, one order of magnitude is gained by parallel data paths for product of matrices. The matrix ($\mathbf{G}$ *Euler2D*) has one zero column, so that a vectorized implementation may use 12 independent data paths, each one for a cost of 3 products and 2 sums. Ten data paths may be used for the second matrix product, which computes 10 strict components, again for a cost of 3 products and 2 sums. Serializing the vectorized operations, we have at most 12 parallel data paths, each one for 10 floating point operations per triangle. Another order of magnitude is gained in the summation of Euler matrices of the triangles, due to the separate data paths of the $4 \times 4 = 16$ components (better: 10 strict components) of the resulting matrix.

6 Conclusion and further developments

We have given a precise definition of the extended Euler tensor, and provided simple formulæ to (a) *fast compute* and (b) *easy transform* the moments of inertia under (c) *general affine maps* through (d) *simple operations* (multiplication and addition) with 4×4 matrices. This technique is *much faster* than conventional volume integration and can be implemented efficiently on programmable GPUs. A pseudocode implementation of both the conventional and the new procedure has been presented and analyzed to compare their computational cost. The method introduced here is beeing used to speed up streaming applications [11,12] in high-performance geometric computing.

A "textured inertia" is also an important component of the new descriptive language *Bioplasm* for multiscale modeling and simulation of deformable living tissues and organs, we are currently designing. In fact, we are strongly interested in developing innovative geometric modeling tools for biophysics simulations of soft, growing bodies, on widely scattered time and space scales. The computation of inertia properties via the extended Euler tensor may be intended as a paradigmatic plan of action for geometric computing with biophysics-based textures.

Acknowledgements

We are pleased to acknowledge valuable comments and advice from Professor Imre Horváth and two anonymous reviewers, that made this paper more readable and complete. The first idea of the extended Euler tensor and its transformations under general affine maps come to A.P. during his sabbatical leave at Cornell in 1986, while tackling a problem posed by Professor John E. Hopcroft. We would also like to thank Giorgio Scorzelli for his help with the computation of the cow example. The research of the last author was supported in part by IBM Corp. through a SUR (Shared University Resources) Award and PLM-Lab donation to Roma Tre, and by the grant 2001-2004 from Spike Consortium, on MIUR Italian National Project SPI-09.

References

[1] Y. Lee, A. Requicha, Algorithms for computing the volume and other integral properties of solids i: known methods and open issues, Communications of the ACM 25 (9) (1982) 635–641.

[2] G. Timmer, J. Stern, Computation of global geometric properties of solid objects, Computer-Aided Design 12 (6) (1980) 301–304.

[3] M. Morteson, Geometric Modeling, John Wiley & Sons, New York, 1985.

[4] S. L. Lien, J. Kajiya, Symbolic method for calculating the integral properties of arbitrary nonconvex polyhedra, IEEE Computer Graphics & Applications 4 (10) (1984) 35–41.

[5] C. Cattani, A. Paoluzzi, Boundary integration over linear polyhedra, Computer-Aided Design 22 (2) (1990) 130–135.

[6] F. Bernardini, Integration of polynomials over n-dimensional polyhedra, Computer Aided Design 23 (1) (1991) 51–58, special issue 'Beyond Solid Modeling'.

[7] C. Gonzalez-Ochoa, S. McCammon, J. Peters, Computing moments of objects enclosed by piecewise polynomial surfaces, ACM Trans. Graph. 17 (3) (1998) 143–157.

[8] T. C. Huang, Engineering Mechanics, Addison-Wesley, Reading, Massachusetts, 1969, second edition.

[9] A. Paoluzzi, Integration constraints in parametric design of physical objects, Techn. Rep. 87-804, Dept. of Computer Science, Cornell University, Ithaca, NY (January 1987).

[10] A. Paoluzzi, Geometric Programming for Computer Aided Design, John Wiley & Sons, Chichester, UK, 2003.

[11] C. Bajaj, A. Paoluzzi, G. Scorzelli, Progressive conversion from B-rep to BSP for streaming geometric modeling, Computer-Aided Design and Applications 3 (5–6) (2006).

[12] G. Scorzelli, A. Paoluzzi, V. Pascucci, Parallel solid modeling using BSP dataflow, International Journal of Computational Geometry & Applications 16 (2006), to appear.

[13] E. Clayberg, D. Rubel, Eclipse: Building Commercial-Quality Plug-ins, Eclipse Series, Addison-Wesley Professional, Boston, MA, 2004.

[14] M. Gurtin, An Introduction to Continuum Mechanics, Mathematics in Science and Engineering, Academic Press, San Diego, CA, 1981.

[15] J. H. Heinbocke, Introduction to Tensor Calculus and Continuum Mechanic, Trafford Publishing, Canada, 2001.

[16] A. Paoluzzi, V. Pascucci, M. Vicentino, Geometric programming: a programming approach to geometric design, ACM Transactions on Graphics 14 (3) (1995) 266–306.

[17] A. Paoluzzi, `PLaSM` language web site, `http://www.plasm.net`, 2005.

[18] The Eclipse Consortium, Eclipse web site, `http://www.eclipse.org`, 2005.

A APPENDIX: Linear representation of affine maps

A.1 Affine maps

An affine map $\mathbf{A} : \mathbb{E} \to \mathbb{E}$ is defined as

$$\mathbf{A} : \mathbf{x} \mapsto (\mathbf{o} + \mathbf{t_o}) + \mathbf{L}\,(\mathbf{x} - \mathbf{o})\,, \tag{A.1}$$

where $\mathbf{L} \in \mathrm{Lin}\,(\mathcal{V})$ is the *linear part* of $\mathbf{A}$ and $\mathbf{t_o} = \mathbf{A}(\mathbf{o}) - \mathbf{o} \in \mathcal{V}$ is the displacement of the origin due to $\mathbf{A}$. Obviously, $\mathbf{A}$ leaves the origin fixed if and only if $\mathbf{t_o} = \mathbf{0}$; it is a translation if and only if its linear part is the identity: $\mathbf{L} = \mathbf{I}$.

A.2 Radius representations of an affine map

The action A.1 of an affine map $\mathbf{A}$ on the general point $\mathbf{x}$ of an affine space $\mathbb{E}$ may be rephrased in terms of the corresponding radius vector $\mathbf{r} = \mathbf{x} - \mathbf{o}$, by introducing the application $\mathbf{R_o} : \mathcal{V} \to \mathcal{V}$ which maps the radius vector of $\mathbf{x}$ into the radius vector of $\mathbf{A}(\mathbf{x})$:

$$\mathbf{R_o}(\mathbf{r}) := \mathbf{A}(\mathbf{o} + \mathbf{r}) - \mathbf{o} \;=\; \mathbf{t_o} + \mathbf{L}\,\mathbf{r}\,. \tag{A.2}$$

Hence, the *radius representation* $\mathbf{R_o}$ is *linear* if and only if $\mathbf{t_o} = \mathbf{A}(\mathbf{o}) - \mathbf{o} = \mathbf{0}$. Therefore, it is plainly impossible to obtain linear representations of *all* affine maps of $\mathbb{E}$ staying *within* $\mathrm{Lin}\,(\mathcal{V})$. However, a simple one-dimensional extension of $\mathcal{V}$ does the trick.

A.3 Linear representation through one-dimensional extension

Let us define the *extended* vector space

$$\mathcal{V}^+ := \{\, (\mathbf{u}, \alpha) \mid \mathbf{u} \in \mathcal{V},\ \alpha \in \mathbb{R}\,\}\,, \tag{A.3}$$

endowed with the obvious extension of the basic linear operations:

$$(\mathbf{u},\alpha) + (\mathbf{v},\beta) := (\mathbf{u}+\mathbf{v}, \alpha+\beta)\,, \qquad \lambda\,(\mathbf{u},\alpha) := (\lambda\,\mathbf{u}, \lambda\,\alpha)\,.$$

It is useful to identify $\mathcal{V}$ with a 1-codimensional subspace of $\mathcal{V}^+$, getting rid of the cumbersome pair notation through the following positions:

$$\mathbf{u} \simeq (\mathbf{u},0) \quad \& \quad \mathbf{k} := (\mathbf{0},1) \qquad \Rightarrow \qquad (\mathbf{u},\alpha) \simeq \mathbf{u}+\alpha\,\mathbf{k}\,. \tag{A.4}$$

The vector $\mathbf{k} \notin \mathcal{V}$ is made into a unit vector orthogonal to $\mathcal{V}$ by the following extension ot the Euclidean inner product:

$$(\mathbf{u}+\alpha\,\mathbf{k})\cdot(\mathbf{v}+\beta\,\mathbf{k}) := \mathbf{u}\cdot\mathbf{v} + \alpha\,\beta\,.$$

Extending $\mathcal{V}$ into $\mathcal{V}^+$ entails the extension of the affine point-space $\mathbb{E} = \mathbf{o} + \mathcal{V}$ into $\mathbb{E}^+ = \mathbf{o} + \mathcal{V}^+$. In order to extend the affine map $\mathbf{A}$ to $\mathbb{E}^+$, its linear part $\mathbf{L}$ has to be extended to a tensor $\mathbf{L}_+ \in \mathrm{Lin}\,(\mathcal{V}^+)$. This gives us some freedom, since $\mathbf{L}_+\mathbf{k}$, the image of $\mathbf{k}$ under the extension of $\mathbf{L}$, can be chosen at will. In fact, the general linear extension of $\mathbf{L}$ has the form:

$$\mathbf{L}_+ = \mathbf{L} + \mathbf{k}\otimes(\mathbf{a}+\alpha\,\mathbf{k}), \tag{A.5}$$

where $\mathbf{L}$ is tacitly identified with its *trivial* extension into $\mathrm{Lin}\,(\mathcal{V}_+)$, such that $\mathbf{L}\,\mathbf{k} = \mathbf{0}$.[8] We now shift the origin of $\mathbb{E}_+$ away from $\mathbb{E}$ (identified with the hyperplane through $\mathbf{o}$ normal to $\mathbf{k}$), assuming as origin the point

$$\mathbf{o}_+ = \mathbf{o} - \mathbf{k} \qquad \Rightarrow \qquad \mathbf{r}_+ := \mathbf{x} - \mathbf{o}_+ = \mathbf{r} + \mathbf{k}\,. \tag{A.6}$$

The radius representation $\mathbf{R}_+$ of $\mathbf{A}_+$ with respect to $\mathbf{o}_+$ reads therefore:

$$\begin{aligned} \mathbf{R}_+(\mathbf{r}_+) := \mathbf{A}_+(\mathbf{o}_+ + \mathbf{r}_+) - \mathbf{o}_+ &= \mathbf{A}_+(\mathbf{o} + \mathbf{r}) - \mathbf{o}_+ \\ &= \mathbf{t}_\mathbf{o} + \mathbf{k} + \mathbf{L}_+\,\mathbf{r} \\ &= \mathbf{t}_\mathbf{o} + \mathbf{k} + \mathbf{L}_+(\mathbf{r}_+ - \mathbf{k}) \\ &= (\mathbf{t}_\mathbf{o} - \mathbf{a}) + (1-\alpha)\,\mathbf{k} + \mathbf{L}_+\,\mathbf{r}_+\,. \end{aligned} \tag{A.7}$$

Hence, the distinguished extension characterized by $\mathbf{a} = \mathbf{t}_\mathbf{o}$ and $\alpha = 1$, *i.e.*,

$$\mathbf{L}_+ = \mathbf{L} + \mathbf{k}\otimes(\mathbf{t}_\mathbf{o} + \mathbf{k}), \tag{A.8}$$

renders $\mathbf{R}_+$ *linear* for any affine map of $\mathbb{E}$, by embedding the general *translation* $\mathbf{r} \mapsto \mathbf{r} + \mathbf{t}_\mathbf{o}$ of $\mathbb{E}$ into the *shear* $\mathbf{r}_+ \mapsto (\mathbf{k}\otimes\mathbf{t}_\mathbf{o})\,\mathbf{r}_+ + (\mathbf{k}\otimes\mathbf{k})\,\mathbf{r}_+$ of the extended space $\mathbb{E}^+$ (see Figure A.1).

[8] This corresponds to the identification of an $n\times n$ matrix with the $(n+1)\times(n+1)$ matrix obtained by adding to the previous one a null row and a null column.

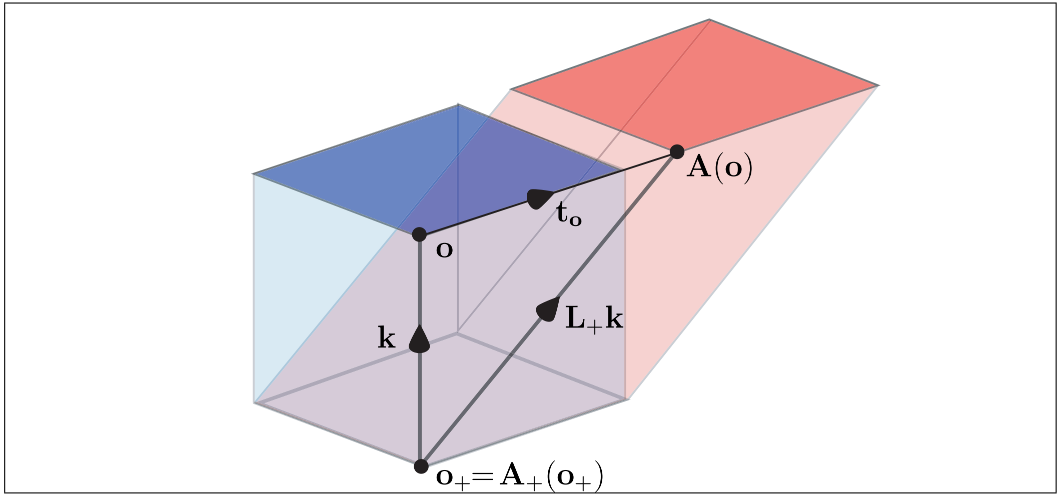


Fig. A.1. A *translation* of $\mathbb{E}$, *i.e.*, $\mathbf{A} : \mathbf{x} \mapsto \mathbf{x} + \mathbf{t_o}$, is extended to a *shear* of $\mathbb{E}^+$: $\mathbf{A}_+ : \mathbf{x} \mapsto \mathbf{o}_+ + \mathbf{L}_+(\mathbf{x} - \mathbf{o}_+) = \mathbf{o}_+ + (\mathbf{k} \otimes (\mathbf{t_o} + \mathbf{k}))\,\mathbf{r}_+ = \mathbf{o}_+ + (\mathbf{k} \cdot \mathbf{r}_+)(\mathbf{t_o} + \mathbf{k})$.

Homogeneous coordinates are provided by any Cartesian frame $(\mathbf{o}_+, (\boldsymbol{\beta}, \mathbf{k}))$, with $\boldsymbol{\beta}$ an arbitrary orthonormal basis of $\mathcal{V}$.

B APPENDIX: Fast computation of inertia properties of surfaces

In this Appendix we provide a pseudocode implementation of the conventional and the fast computation of the extended Euler tensor for a surface. Both algorithms were actually tested using the RAD language `PLaSM` for the development of geometric algorithms [16,10,17]. The `PLaSM` IDE is available as an Eclipse plugin [18,13].

For the same reasons why mass was identified with volume in Section 4, devoted to solid bodies, we assume here that mass and area measures coincide. As in Section 4.2, a Cartesian frame is taken for granted and the ambient space $\mathbb{E}^3$ is identified with $\mathbb{R}^3$.

B.1 Structure products on the standard 2-simplex in $\mathbb{R}^2$

Figure B.1 displays a pseudocode implementation of formula (6) which provides the structure products $I_2(\alpha, \beta)$ for the standard triangle $\boldsymbol{\Delta}_2 \in \mathbb{R}^2$. This function is used as a component in Sections B.2 and B.3, where results are transferred—in different ways—to a general triangle in $\mathbb{R}^3$.

$$\boxed{\textbf{Def}\ \textit{term2D}(\alpha,\beta\in\mathbb{N}) := \frac{1}{\alpha+1}\sum_{h=0}^{\alpha+1}\binom{\alpha+1}{h}\frac{-1^h}{h+\beta+1}}$$

Fig. B.1. Implementation of structure products $I_2(\alpha,\beta)$.

B.2 Fast computation of the extended Euler tensor for a general triangle in $\mathbb{R}^3$

Figure B.3 provides a pseudocode implementation of the fast computation method presented in this paper. The linear representation of an affine transformation that maps the prototype triangle $\operatorname{conv}(\mathbf{0},\mathbf{e}_1,\mathbf{e}_2) \simeq \boldsymbol{\Delta}_2$ into the general triangle $\boldsymbol{\tau} = \operatorname{conv}(\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2)$ is denoted as $\mathbf{G}$. The points $\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2$ are supposed affinely independent.

B.3 Direct integration of the structure products on a general 2*-simplex in* $\mathbb{R}^3$

In order to compare the performance of the approach presented in this paper and documented above with the conventional integration method for structure products [5], we give a functional implementation of the latter in Figure B.2. The functional pseudocode is a straightforward translation of Formula B.1 for direct integration over the triangle $\boldsymbol{\tau} = \operatorname{conv}(\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2) \subset \mathbb{R}^3$, whose first and second sides through $\mathbf{v}_0$ are denoted $(a_1,a_2,a_3) \equiv \mathbf{a} := \mathbf{v}_1 - \mathbf{v}_0$ and $(b_1,b_2,b_3) \equiv \mathbf{b} := \mathbf{v}_2 - \mathbf{v}_0$. Formula B.1 is proved in [5], where an imperative coding in *Pascal* is also given.

$$\begin{aligned} I_{\boldsymbol{\tau}}(\alpha,\beta,\gamma) &= \int_{\boldsymbol{\tau}} x^\alpha y^\beta z^\gamma \, dA \\ &= 2\,A(\boldsymbol{\tau}) \sum_{h=0}^{\alpha}\binom{\alpha}{h} x_o^{\alpha-h} \sum_{k=0}^{\beta}\binom{\beta}{k} y_o^{\beta-k} \sum_{m=0}^{\gamma}\binom{\gamma}{m} z_o^{\gamma-m} \cdot \\ &\quad \cdot \sum_{i=0}^{h}\binom{h}{i} a_1^{h-i} b_1^{i} \sum_{j=0}^{k}\binom{k}{j} a_2^{k-j} b_2^{j} \sum_{l=0}^{m}\binom{m}{l} a_3^{m-l} b_3^{l}\ I_2(\mu,\nu)\,, \end{aligned} \tag{B.1}$$

where $\mu = (h{+}k{+}m) - (i{+}j{+}l)$, $\nu = (i{+}j{+}l)$, and $I_2(\mu,\nu)$ is the general structure product for the standard triangle $\boldsymbol{\Delta}_2 \subset \mathbb{R}^2$, as given by formula (6) (see also Section B.1). Since our reference implementation [12] is data-flow-oriented, and data-flow does not allow for iteration, formula (B.1) is implemented by generating all the tuples (i,j,l,h,k,m,a,b,c), and streaming them all through the local function `formula`.

```
**Def** $\textit{EulerTensor}\ (\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2 \subset \mathbb{E}^3) :=$
$\begin{pmatrix} int(2,0,0) & int(1,1,0) & int(1,0,1) & int(1,0,0) \\ int(1,1,0) & int(0,2,0) & int(0,1,1) & int(0,1,0) \\ int(1,0,1) & int(0,1,1) & int(0,0,2) & int(0,0,1) \\ int(1,0,0) & int(0,1,0) & int(0,0,1) & int(0,0,0) \end{pmatrix}$
**where**
  $\mathbf{a} := \mathbf{v}_1 - \mathbf{v}_0,\ \mathbf{b} := \mathbf{v}_2 - \mathbf{v}_0,\ (x_0, x_1, x_2) := \mathbf{v}_0,$
  $jacobian := |\mathbf{a} \times \mathbf{b}|,$
  $int\ (\alpha, \beta, \gamma \in \mathbb{N}) :=$
    $jacobian * \sum_{(h,k,m) \in S_1} \sum_{(i,j,l) \in S_2} formula(i, j, l, h, k, m, \alpha, \beta, \gamma)$
  **where**
    $S_1 := S_h \times S_k \times S_m,\ S_2 := S_i \times S_j \times S_l,$
    $S_h := \{0, 1, \ldots, \alpha\},\ S_k := \{0, 1, \ldots, \beta\},\ S_m := \{0, 1, \ldots, \gamma\},$
    $S_i := \{0, 1, \ldots, h\},\ S_j := \{0, 1, \ldots, k\},\ S_l := \{0, 1, \ldots, m\}$
    $formula(i, j, l, h, k, m, \alpha, \beta, \gamma \in \mathbb{N}) :=$
      $\binom{\alpha}{h} * \binom{\beta}{k} * \binom{\gamma}{m} * \binom{h}{i} * \binom{k}{j} * \binom{m}{l} *$
      $(x_0 \,\hat{}\,(\alpha - h)) * (y_0 \,\hat{}\,(\beta - k)) * (z_0 \,\hat{}\,(\gamma - m)) *$
      $(a_x \,\hat{}\,(h - i)) * (b_x \,\hat{}\,i) * (a_y \,\hat{}\,(k - j)) * (b_y \,\hat{}\,j) *$
      $(a_z \,\hat{}\,(m - l)) * (b_z \,\hat{}\,l) * term2D(\mu, \nu),$
      $\mu := h + k + m - \nu,$
      $\nu := i + j + l$
    **end**
  **end**;
```

Fig. B.2. Computation of the Euler tensor for a triangle with vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2 \in \mathbb{E}^3$ using direct integration formulæ.

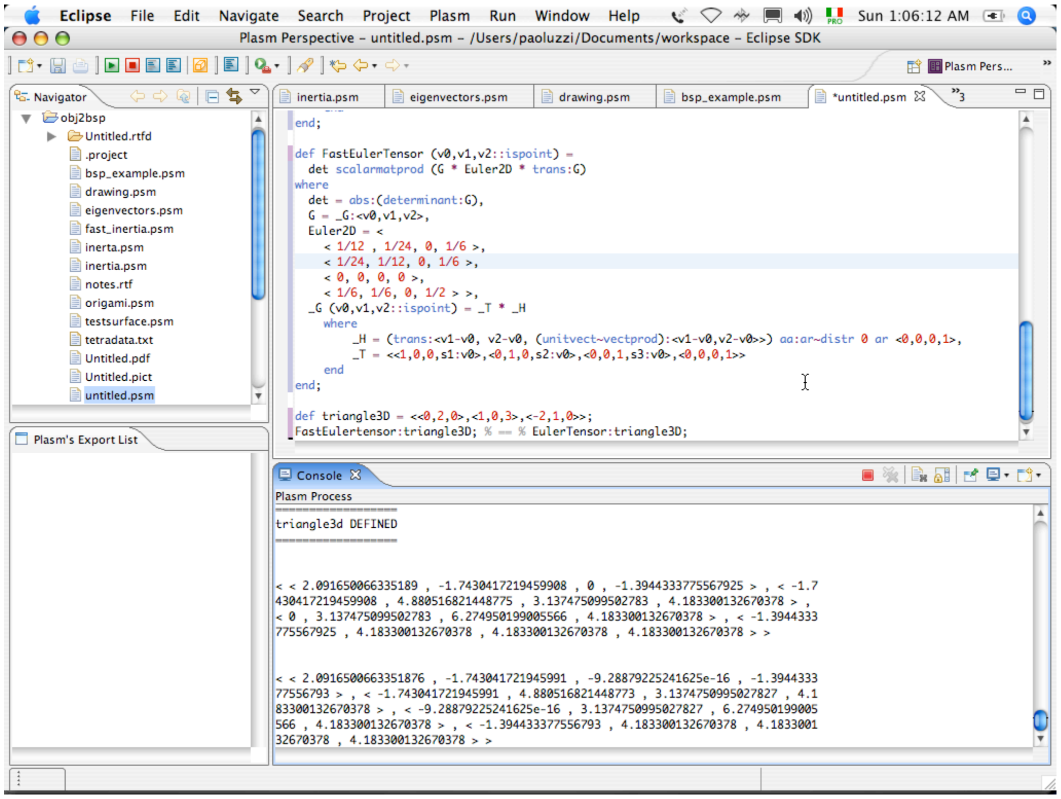
Def *FastEulerTensor* $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2 \subset \mathbb{E}^3) := \det(\mathbf{G})\ \mathbf{G}\ \mathit{Euler2D}\ \mathbf{G}^T$
where

$$\mathit{Euler2D} := \frac{1}{24}\begin{pmatrix} 2 & 1 & 0 & 4 \\ 1 & 2 & 0 & 4 \\ 0 & 0 & 0 & 0 \\ 4 & 4 & 0 & 12 \end{pmatrix},$$

$$\mathbf{G} := \begin{pmatrix} \mathbf{v}_1 - \mathbf{v}_0 & \mathbf{v}_2 - \mathbf{v}_0 & (\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0) & \mathbf{v}_0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

end;

Fig. B.3. Fast computation of the Euler tensor for the triangle with vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2 \in \mathbb{E}^3$.



```
end;

def FastEulerTensor (v0,v1,v2::ispoint) =
  det scalarmatprod (G * Euler2D * trans:G)
where
  det = abs:(determinant:G),
  G = _G:<v0,v1,v2>,
  Euler2D = <
    < 1/12 , 1/24, 0, 1/6 >,
    < 1/24, 1/12, 0, 1/6 >,
    < 0, 0, 0, 0 >,
    < 1/6, 1/6, 0, 1/2 > >,
  _G (v0,v1,v2::ispoint) = _T * _H
    where
        _H = (trans:<v1-v0, v2-v0, (unitvect~vectprod):<v1-v0,v2-v0>>) aa:ar~distr 0 ar <0,0,0,1>,
        _T = <<1,0,0,s1:v0>,<0,1,0,s2:v0>,<0,0,1,s3:v0>,<0,0,0,1>>
    end
end;

def triangle3D = <<0,2,0>,<1,0,3>,<-2,1,0>>;
FastEulertensor:triangle3D; % == % EulerTensor:triangle3D;
```

```
==================
triangle3d DEFINED
==================


< < 2.091650066335189 , -1.7430417219459908 , 0 , -1.3944333775567925 > , < -1.7
430417219459908 , 4.880516821448775 , 3.137475099502783 , 4.183300132670378 > ,
< 0 , 3.137475099502783 , 6.274950199005566 , 4.183300132670378 > , < -1.3944333
775567925 , 4.183300132670378 , 4.183300132670378 , 4.183300132670378 > >


< < 2.0916500663351876 , -1.743041721945991 , -9.28879225241625e-16 , -1.3944333
77556793 > , < -1.743041721945991 , 4.880516821448773 , 3.1374750995027827 , 4.1
83300132670378 > , < -9.28879225241625e-16 , 3.1374750995027827 , 6.27495019900
566 , 4.183300132670378 > , < -1.394433377556793 , 4.183300132670378 , 4.1833001
32670378 , 4.183300132670378 > >
```

Fig. B.4. The Eclipse IDE with the `PLaSM` plugin in action.