Computing the distance between two finite element solutions defined on different 3D meshes on a GPU

Maxence Reberol, Bruno Lévy

Computing the distance between two finite element solutions defined on different 3D meshes on a GPU

Maxence Reberol* Bruno Lévy†

August 30, 2017

Abstract

This article introduces a new method to efficiently compute the distance (i.e., L^p norm of the difference) between two functions supported by two different meshes of the same 3D domain. The functions that we consider are typically finite element solutions discretized in different function spaces supported by meshes that are potentially completely unrelated. Our method computes an approximation of the distance by resampling both fields over a set of parallel 2D regular grids. By leveraging the parallel horse power of computer graphics hardware (GPU), our method can efficiently compute distances between meshes with multi-million elements in seconds. We demonstrate our method applied to different problems (distance between known functions, Poisson solutions, linear elasticity solutions) using different function spaces (Lagrange polynomials from order one to seven) and different meshes (tetrahedral, hexahedral, with linear or quadratic geometry).

Keywords: distance, field distance, finite element, error estimate, mesh comparison, approximation error, error analysis

1 Introduction

Mesh-based numerical methods approximate solutions of partial differential equations by combining simpler functions defined on a mesh. A natural question that arises is how to quantify the approximation error due to the numerical method, and how to compare different results obtained with different function spaces. Except for trivial problems, there is no analytical solution and one has to rely on error estimates. Our approach focuses on comparing solutions obtained with different methods. Comparing different solutions requires to measure the distance between functions that are not always defined in the same function space. The main difficulty is that the different function spaces may be supported by meshes that are potentially completely different/unrelated. We propose a practical tool to quickly measure the distance between two solutions in such a situation. We think that such a tool was missing in the "numerical toolbox". Our method has the potential of helping answering different questions, such as estimating the impact of the mesh on the solution, estimating the influence of the finite element space, comparing different codes, or estimating how to optimize the mesh size in order to reach a given accuracy.

*Inria Nancy and LORIA, (`maxence.reberol@inria.fr`)

†Inria Nancy and LORIA, (`bruno.levy@inria.fr` , `https://members.loria.fr/BLevy/`)

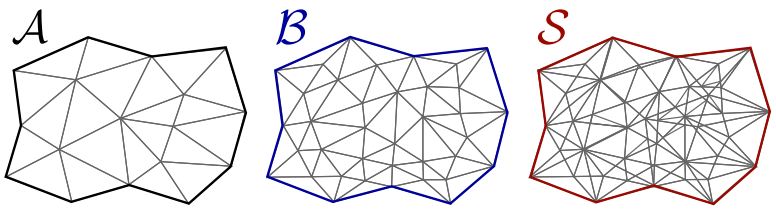


Figure 1: Two different meshes $\mathcal{A}$ and $\mathcal{B}$ of the same domain. $\mathcal{S}$ the supermesh of both $\mathcal{A}$ and $\mathcal{B}$. In 3D and with curved elements, computing such a $\mathcal{S}$ supermesh is very difficult, therefore we replace it with slicing and sampling.

Thus, our method computes an approximation of the distance between two fields, scalar or vector, which are piecewise-defined on 3D unstructured meshes. The cells of the two mesh can be arbitrary and completely different. In the results reported in this article, we used linear, quadratic and cubic tetrahedra, and trilinear, quadratic and cubic hexahedra. Our method can both compute the quadratic norm (L^2) and the maximum difference norm (L^∞). To compute the distance between the two functions, we resample them on a 3D regular grid, that we compute slice by slice. As discussed in the related work section below, there are alternative approaches based on mesh intersections that create a function space rich enough to exactly represent the two functions to be compared. However, such approaches are slower, and are not simply applicable to meshes that contain *curved* elements (e.g. trilinear hexahedra, quadratic tetrahedra). For this reason, we adopted a *brute-force* strategy (GPU-based) to sample both functions. It is fast and deals efficiently with large meshes: in the results section, we compute (Fig. 11) the L^p-distances between a piecewise-linear solution defined on 2.9M tetrahedra and a piecewise-quadratic solution defined on 2.3M tetrahedra within 2 seconds.

2 Related work

We first discuss standard finite element approaches in which the field difference is represented on a common functional space, where the distance can be computed by quadratures (§2.1). In the present work, the approach is very different and is closer to techniques that are used in high-order finite element visualization (§2.2).

2.1 Distance via quadratures

A possible approach to compute the distance between two fields is to project both functions onto the same function space that will be used to compute the distance. There are basically two possibilities: (i) project one field onto the other mesh, together with its associated functional space, or (ii) project both fields on a third mesh/function space, designed to accurately represent both input functions. In both cases, there is data transfer, or projection, from one mesh onto another.

The meshes we are interested in are unstructured, potentially completely unrelated / non-nested. In previous works, data transfer between such meshes have been developed as it appears in various frameworks such as Galerkin projections (i.e. interpolation in a weak sense) in the context of mixed finite elements [10], solution transfers after adaptative remeshing in time-dependant problems [4, 6] or multigrid solvers using non-nested meshes [18, 9].

In our general setting, projecting one field onto the other approximation space is not always applicable since the target space may not be suitable to represent the initial field with sufficient fidelity. Both meshes may have very different element sizes and shapes, polynomial orders or even discontinuity in the function spaces. In previous works, this concern has been addressed by producing an auxiliary mesh composed of element intersections of both input meshes, as illustrated in 2D in Fig. 1. Accurately and efficiently computing intersections between 3D meshes is difficult, and has been studied mainly for tetrahedral meshes, with different names: rendezvous mesh [20, 23], common-refinement mesh [12], supermesh [8, 15] or mesh intersection [4]. To keep memory requirements reasonable and to allow parallelizing the method, these intersection meshes are not built explicitly but locally. The advantage of this approach is that it defines a superspace [7] that includes both input approximation spaces. Then the field difference $f - g$ can be exactly represented, and used to compute the L^p-distance with adapted quadratures. They are used to compute conservative interpolations that preserve some field properties (e.g. mass or energy) during the projections. However, such methods are extremely difficult to implement for curved elements ($\mathbb{P}_2$, $\mathbb{P}_3$). The problem also appears when using trilinear hexahedra that have non-planar faces. The approach that we propose easily handles curved elements (but this cames at the price of a less elegant "brute force" approach).

2.2 Finite element visualization techniques

Our approach samples both fields on a regular 3D grid that is sliced, i.e. decomposed into a large number of parallel 2D grids. In a certain sense, evaluating the fields at regular samples of a 2D grid is very similar to rendering a finite element solution onto the screen (that is also considered as a 2d grid of pixels). For this reason, we discuss in this subsection several techniques to display finite element solutions that share some similarities with our approach. Interestingly, graphic hardware (GPU) and the associated standard rendering techniques can very efficiently process huge numbers of triangles and compute linear interpolations inside them. Note that in our context we are also interested in higher-order interpolations and possibly *curved* elements (more on this later).

Historically, GPUs were only optimized for linear interpolations. For this reason, a first approach to render the fields, or equivalently to evaluate them, is to find optimal refinements of the elements to build an accurate piecewise-linear approximation (i.e. tetrahedra) of the fields [13, 21], then to use the marching tetrahedra algorithm [3, 24] to evaluate the fields at pixel centers. This approach requires error estimates and leads to huge refined meshes, which are not practical, especially if the input meshes are already large.

Still using early hardware, to simulate higher-order interpolation, an alternative [11] combines pre-computed images (textures) defined for each function of the polynomial basis. However, this requires significant pre-processing for all the basis functions and it is not well adapted to the case of a 3D mesh cut by a 2D plane.

With the introduction of programmable shaders in the standard OpenGL/DirectX rendering pipelines, it is more efficient to directly evaluate the finite element shape functions on the slicing plane at each pixel [5, 17]. The element containing the pixel is found via raycasting and the reference coordinates, required to evaluate the shape functions, are obtained via mapping inversion with the Newton-Raphson iterative method. Such an approach is expensive when considering dozens of millions of samples, especially for non-affine mappings. To avoid mapping inversions for curved cells, it has been proposed to approximate the shape functions along the rays in world space with L^2-projections [16]

on 1D polynomials. An alternative is proposed in [25].

In our context, since we sample the fields slice by slice (each slice independently), we just need to compute the intersection between the mesh and a plane parallel to the bounding box. Therefore we do no need the flexibility of the raycasting approach. We use instead the marching tetrahedra algorithm [3, 24] and the OpenGL rasterization to interpolate the reference coordinates at pixel centers. This has the advantages of making the algorithm simpler. More importantly, this makes use of the rasterization hardware, that is extremely efficient at determining the list of pixels contained in each triangle.

3 Our method: L^p-distance approximation via slicing and sampling

3.1 Overview

Our approach evaluates both input fields on a *3D regular grid of points*. Following the standard definition of finite elements, we assume that field values are defined by an *interpolation∘inverse mapping* composite function with the form $f_{|K}(\mathbf{x}) = \hat{f}_K \circ \mathbf{M}_K^{-1}(\mathbf{x})$, where f_K is the field interpolation function defined in a reference space and where $\mathbf{M}_K$ denotes the *forward mapping* of the element K from the reference space to the world space[1].

We avoid inverting the mappings by directly interpolating the reference coordinates ($\hat{\mathbf{x}} = \mathbf{M}_K^{-1}(\mathbf{x})$) in world space. Our implementation supports polynomial element mappings (e.g. trilinear hexahedra, quadratic tetrahedra) and various element shape functions. For non-affine mappings, we decompose each element into a set of small tetrahedra to approximate the geometry. This subdivision is executed directly on the GPU. This does not affect the performance too much since there is no additional required memory transfer (element subdivision is completely done on the GPU).

Sampling the fields directly over the 3D regular grid is not reasonable as it would consume too much memory. Thus we decompose distance computation (§3.2) into layers (§3.3), or *slices*, corresponding to parallel planes of the 3D regular grid. On each layer, both fields are evaluated at grid samples (§3.4, Fig. 2). Using this slicing algorithm, only one slice at a time needs to be stored. The OpenGL framework allows to efficiently process the layers by exploiting GPU hardware (Appendix A).

The distance approximation computed by our method converges to the distance computed with quadratures on an analytical problem (§4.1) and converges at different rates with the number of samples, and with the subdivision level for curved elements, depending of the mesh and field properties (§4.2). Performance is measured on test cases of various sizes and show that a good accuracy is obtained in less than one second for standard problems (§4.3). Fast distance computation allows to study the convergence of Lagrange finite elements on a realistic problem in a few minutes (§4.4). Alternatively, the grid sampling approach can be used to generate useful visualizations and allows to develop interactive tools to investigate localized features in the fields (§4.5). The computation converges in reasonable time (one minute) even with high-order polynomials (order 5 and 7) on multi-million element meshes (§B). However, there are limitations associated to

[1] thus it is inverted when evaluating $\hat{f}_K$ that takes its argument in the reference space.

regular sampling and to the usage of GPU hardware (§4.6).

The open-source program associated with this work is included in the supplementary material.

3.2 Problem setting

The input of our method is a pair of functions f and g (or fields), defined as finite-element solutions, supported by two different meshes $\mathcal{A}$ and $\mathcal{B}$ of the same domain Ω.

The goal of our method is to compute an approximation of the L^p-distance between the fields f and g, given by:

$$\begin{aligned} \|f-g\|_{L^p} &= \left(\int_{\mathcal{A}\cap\mathcal{B}} (f(\mathbf{x})-g(\mathbf{x}))^p d\mathbf{x} \right)^{\frac{1}{p}} && \text{if } p < \infty \\ &= \max_{x\in\mathcal{A}\cap\mathcal{B}} (f(\mathbf{x})-g(\mathbf{x})) && \text{if } p = \infty \end{aligned}$$

We approximate this norm by sampling it on a *3D regular grid of points*:

$$d_{L^p,h}(f,g) = \left(\sum_{i=1..N} h^3 (f(\mathbf{x}_i)-g(\mathbf{x}_i))^p \right)^{1/p}$$

where h is the space between two adjacent samples and N the number of samples inside the volumes of both $\mathcal{A}$ and $\mathcal{B}$. The samples $(x_i)_{i=1..N}$ can also be seen as the voxel centers of a *voxel grid*.

Clearly, for bounded piecewise continuous functions, we have:

$$d_{L^p,h}(f,g) \xrightarrow[h\to 0]{} \|f-g\|_{L^p}$$

Fields which are piecewise-defined on meshes, such as finite element solutions, are usually relatively smooth. Therefore convergence should be reached within a reasonable number of samples in practice. We verified this assumption by measuring the impact of the distance between two samples h (voxel size) on the distance computation (see experimental data in section 4.2). In the end, we have virtually built a voxelization of the field difference $f-g$. We can also interpret it as a piecewise-constant approximation of the field difference on a regular cubic mesh. Since there is a very large number of tiny voxels (potentially billions of them), this approximation is sufficiently accurate in practice.

3.3 Slicing

We decompose the 3D voxel grid into a list of 2D pixel grids, that we call *slices* and sample both input functions f and g on these slices. The distance computation can be rewritten as:

$$d_{L^p,h}(f,g)^p = h^3 \sum_{k=1..n_s} \sum_{i=1..N_k} (f(\mathbf{x}_i)-g(\mathbf{x}_i))^p$$

where n_s is the number of slices and $(\mathbf{x}_i)_{i=1..N_k}$ are the samples of the slice k which are inside both $\mathcal{A}$ and $\mathcal{B}$.

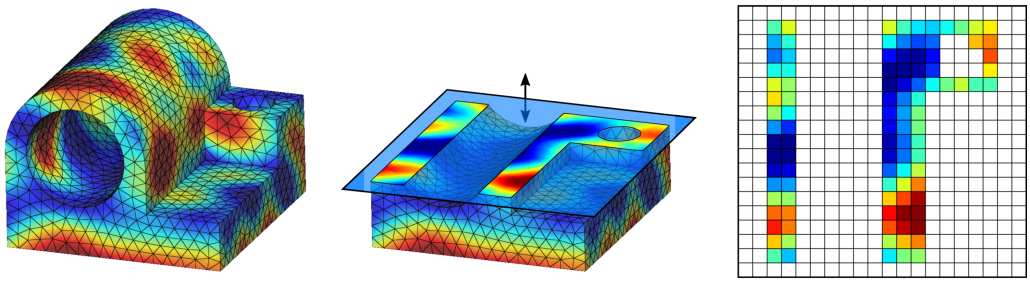


Figure 2: (left) scalar field on a tetrahedral mesh. (center) slicing plane. (right) field values at pixel centers.

Clearly, the inner sums can be computed independetly: after a slice is processed, the only value that we need to store is its contribution to the global sum. As a consequence, only one slice at a time needs to be stored. It makes it possible to compute distances using a 3D grid with billions samples while using a limited amount of memory (a 2D grid with millions samples).

For a given slice k, we can compute independently the values of f and g at all the pixel centers of the slice. That is, we first compute the lists $(f_i = f(\mathbf{x}_i))_{i=1..N_k}, (g_i = g(\mathbf{x}_i))_{i=1..N_k}$ independently, and then we compute the sum of the differences. We call this process *field sampling* (illustrated in Fig. 2).

The global algorithm of our approach is:

Algorithm 1 Distance computation

compute the voxel grid dimensions n_x, n_y, n_s from h
for each slice k in $[1, n_s]$ **do**
  compute the field samples $(f_i)_{i=1..N_k}$ ▷ subsection 3.4
  compute the field samples $(g_i)_{i=1..N_k}$ ▷ subsection 3.4
  compute the difference $c_k^p = \sum_{i=1..N_k} (f(\mathbf{x}_i) - g(\mathbf{x}_i))^p$
end for
compute the global distance $d_h^p(f, g) = (h^3 \sum_{k=1..n_s} c_k^p)^{1/p}$

3.4 Sampling the field on a 2D pixel grid

The ***input*** is a *field* f defined on a mesh $\mathcal{A}$ and a *regular grid of sampling points* $(\mathbf{x}_{ij})_{ij}$. The field values are piecewise-defined inside the elements K by a composite function with the form:

$$f_{|K}(\mathbf{x}) = \hat{f}_K \circ M_K^{-1}(\mathbf{x}) \tag{1}$$

where $\mathbf{M}_K$ denotes the element mapping such that $K = \mathbf{M}_K(\hat{K})$, and where $\hat{K}$ is the reference element (e.g. the unit cube or the reference tetrahedron). We denote by $\hat{\mathbf{x}} = \mathbf{M}_K^{-1}(\mathbf{x})$ the reference coordinates. The function $\hat{f}_K$ is the interpolation defined by the coefficients, associated to the cell K, of a function basis defined on the reference cell $\hat{K}$. The sampling points $(\mathbf{x}_{ij})_{ij}$ form a rectangular grid of equally spaced points. We denote by $P(\mathbf{x}) = 0$ the equation of the associated plane $\mathbf{P}$.

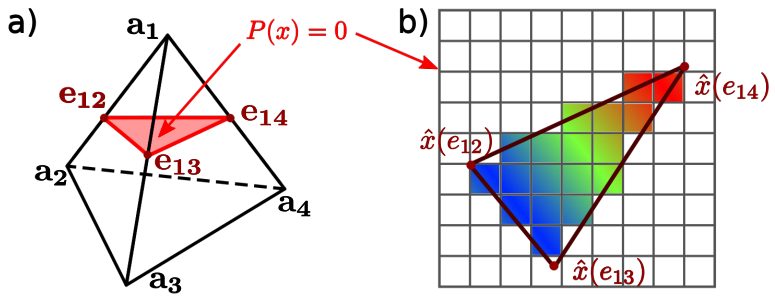


Figure 3: (a) marching tetrahedra: intersection between the slicing plane $\mathbf{P}$ and a tetrahedra. (b) triangle rasterization and interpolation of reference coordinates $\hat{\mathbf{x}}$ at pixel centers (symbolized by the pixel colors).

The ***output*** is the list of values $(f_{ij})_{ij}$ which are the values of f at the sampling points $(\mathbf{x}_{ij})$. For the samples which are not inside the mesh $\mathcal{A}$, we simply store a *no data* value.

When $\mathcal{A}$ is a tetrahedral mesh, we sample the field with Algorithm 2:

Algorithm 2 Field sampling on tetrahedral meshes

```
for each tetrahedron K such that K ∩ P ≠ ∅ do
    compute x̂ at K ∩ P vertices                    ▷ marching tetrahedra
    for each sample x_ij inside K ∩ P do
        interpolate linearly x̂ to get x̂_ij at x_ij   ▷ triangle rasterization
        compute f_ij = Σ_k f_k φ̂_k(x̂_ij)             ▷ field evaluation
    end for
end for
```

We will explain later in §3.5 how to extend algorithm 2 to deal with meshes composed of *curved* elements .

Marching tetrahedra algorithm

For each tetrahedra, we have to compute the intersection with the slicing plane. An simple and efficient algorithm is the standard marching tetrahedra [3].

Consider the tetrahedron of vertices $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3, \mathbf{a}_4$ and a plane of equation $P(\mathbf{x}) = 0$. By looking at the signs of $P(\mathbf{a}_1), P(\mathbf{a}_2), P(\mathbf{a}_3), P(\mathbf{a}_4)$, we determine the intersection configuration, which is either empty, a triangle or a square (i.e. two triangles). If it is one or two triangles, we compute the positions and the reference coordinates of the vertices at intersected edges. For example in figure 3.a., we have:

$$\mathbf{e}_{12} = (1-\alpha)\mathbf{a}_1 + \alpha\mathbf{a}_2$$
$$\hat{\mathbf{e}}_{12} = (1-\alpha)\hat{\mathbf{a}}_1 + \alpha\hat{\mathbf{a}}_2$$

with $\alpha = \frac{P(\mathbf{a}_1)}{P(\mathbf{a}_1)-P(\mathbf{a}_2)}$, $\hat{\mathbf{a}}_1 = (0,0,0)$, $\hat{\mathbf{a}}_2 = (0,1,0)$

Optimization: restriction to a set of candidates In a basic implementation, the marching tetrahedra algorithm is called for all tetrahedra of the mesh (at a given slicing plane). This can be highly inefficient for large meshes as most of the elements are not

intersected by the plane. We propose a simple optimization that consists in sorting the elements and maintaining a list of cell candidates.

For further implementation on the GPU, we want a range $[first, last]$ of *consecutive* element candidates. A possibility is to sort the tetrahedra by their position along the axis perpendicular to the slicing plane. For example, if we slice the mesh with planes perpendicular to the z-axis, we sort tetrahedra by their minimum z-coordinate. Then as the slicing plane altitude increases, the range $[first, last]$ is updated such that:

- $first$ is the tetrahedron below the slicing plane with the highest z-coordinate
- $last$ is the tetrahedron with the highest minimum z-coordinates, whose lowest vertex is below the slicing plane

This range still contains some tetrahedra which are not intersected by the slicing plane but we have found this simple optimization is sufficient for standard meshes, and results in a significant speed up in our GPU implementation, as it avoids lot of memory access and synchronization inefficiencies.

Triangle rasterization

The *input* of the rasterization is an intersection triangle produced by the marching tetrahedra step 3.4. The rasterization consists in determining the sampling points which are inside the input triangle, and interpolating the reference coordinates at these points. This process is illustrated in Fig. 3.b.

In our implementation discussed in section A, this step is executed automatically by the OpenGL pipeline.

Field evaluation at sample

At each sample $\mathbf{x}_{ij}$ inside the rasterized triangle, we evaluate the field value f_{ij} with the finite element formula:

$$f_{ij} = \sum_{k=1..N_K} f_k \hat{\phi}_k(\hat{x}_{ij})$$

For Lagrange finite elements, the f_k's are the coefficients associated to the N_K degrees of freedom of the element K, $(\hat{\phi}_k)_{k=1..N_K}$ is the Lagrange function basis associated to the reference element $\hat{K}$ and the reference coordinates $\hat{\mathbf{x}}_{ij}$ have been computed by the previous triangle rasterization step.

This evaluation step can be generalised to more complex cases where one can compute f_{ij} from the reference coordinates and some coefficients:

$$f_{ij} = f_K(\hat{\mathbf{x}}_{ij})$$

This allows to implement the field interpolation for vector basis function such as the Nedelec or Raviart-Thomas finite elements.

3.5 Extension to curved elements

We now extend the algorithm to *curved* elements, i.e. elements which geometry is not defined by an affine mapping but by a polynomial one, such as $\mathbb{P}_2, \mathbb{P}_3$ elements and (trilinear) hexahedra. Before detailing the method, we motivate the particular approach that we are using: in the field evaluation equation (1), finding the element that contains a

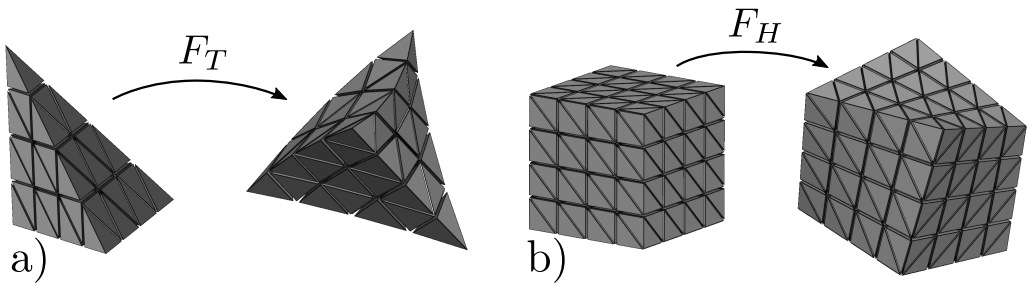


Figure 4: Piecewise linear approximation of mappings (decomposition in sub-tetrahedra) (a) quadratic tetrahedron mapping with two subdivision levels. (b) trilinear hexahedron mapping with two subdivision levels.

sample point and inversing the mapping (for each sample) are computationally expensive. When considering millions of samples and non-affine mappings, this becomes a prohibitive cost. In our approach, we never have to locate elements nor to inverse the mappings, the backward mechanism is replaced by a purely forward one, in the same spirit of the finite element assembly where one processes the cells independently and uses a change of variable to avoid inverse mapping computations. More importantly, all the steps of algorithms 2 and 3 can be efficiently implemented in a GPU pipeline which have dedicated hardware for triangle rasterization and linear interpolation.

The algorithm in the previous subsection computes exactly the reference coordinates at the sampling points as the tetrahedra mappings are affine functions. This argument holds for any element defined by an affine mapping, but in practice this only concerns tetrahedra or cubes in regular meshes. In this subsection, we are interested in extending our approach to meshes in which elements have non-planar faces such as hexahedra, pyramids, quadratic tetrahedra, etc.

Consider a curved element K defined by the polynomial mapping $\mathbf{M}_K$ of a reference element $\hat{K}$. To compute the field values, we need the reference coordinates at pixel centers, but this time it is no longer possible to get them by linear interpolation as the mappings are not affine. The exact approach would be to draw a piecewise linear bounding box of the element, then for each pixel of the rasterization of this bounding box, to inverse the mapping (with a Newton-type iterative method), then to check whether the pre-image of the sample is inside the reference element. This approach is computationally expensive and not straightforward to implement, so instead we propose to introduce a piecewise-linear approximation of the geometry:

The reference element $\hat{K}$ is decomposed into many sub-tetrahedra which are mapped using the mapping $\mathbf{M}_K$, as illustrated on figure 4. The field sampling algorithm becomes:

Algorithm 3 Field sampling on *curved* meshes

```
for each cell K do
    decompose K into sub-tetrahedra T's                    ▷ element decompositions
    for each sub-tetrahedron T such that T ∩ P ≠ ∅ do
        compute x̂ at T ∩ P vertices                        ▷ marching tetrahedra
        for each sample x_ij inside T ∩ P do
            interpolate linearly x̂ to get x̂_ij at x_ij     ▷ triangle rasterization
            compute f_ij = Σ_k f_k φ̂_k(x̂_ij)               ▷ field evaluation
        end for
    end for
end for
```

This algorithm is almost the same with the exception that there is a new outer loop. The reference coordinates are computed at sub-tetrahedra T vertices, then the marching tetrahedra step in executed on the sub-tetrahedra T and eventually the reference coordinates of the parent cell K are obtained at samples. This way we produce a piecewise linear approximation of the reference coordinates in the curved elements. Formally, the new field evaluation formula is:

$$f_{ij} = \sum_{k=1..N_K} f_k \ \hat{\phi}_k(\tilde{\mathbf{M}}_K^{-1}(\mathbf{x}_{ij}))$$

where $\tilde{\mathbf{M}}_K$ is the piecewise-linear approximation of the mapping $\mathbf{M}_K$.

Element decompositions

In our implementation, we use recursive subdivision levels *sl*. At each level, a basic subdivision is applied to all the current sub-elements.

For *hexahedra*, our basic subdivision consists in decomposing a cube into eight cubes. After all the subdivision levels have been applied, the sub-cubes are each decomposed into six tetrahedra. A level two subdivision (i.e. a hex into 384 sub-tetrahedra) is shown in Fig. 4. In the end, hexahedra are decomposed into $8^{sl} \times 6$ sub-tetrahedra. When using our decomposition of a hexahedron, there is a diagonal direction that appear on faces. Two adjacent hexahedra can be decomposed with two opposite diagonal directions. In this case, the partition of the space is not perfect and there may be small holes or overlaps at hexahedra interfaces. In practice, we observed that with a sufficiently large number of subtetrahedra, the introduced error is negligible. Note that it would be possible to completely avoid it by implementing compatible tesselations (at the price of combinatorial pre-preprocessing).

For *curved tetrahedra*, our basic subdivision consists in applying the 4-tetrahedra-1-octahedron decomposition and then decomposing the octahedron into four tetrahedra. So at each subdivision level, each tetrahedron becomes eight sub-tetrahedra. In the end, each *curved tetrahedron* is decomposed into 8^{sl} sub-tetrahedra. A level two subdivision of a *quadratic tetrahedron* (i.e. 64 sub-tetrahedra) is shown in Fig. 4.

The geometry approximation of curved cells introduces an error on the distance computation which decreases with the refinement level. The appropriate level of subdivision depends of the application: a level one subdivision may be sufficient for a rough estimate but one should use higher levels for high-accuracy distance computations (e.g. distance to reference solution in convergence analysis) or for highly curved elements.

4 Results and discussion

In this section, we validate the distance approximation on simple problems with analytical solutions (§4.1) and give insights on how to choose the approximation parameters (number of samples, subdivision level for curved cells) on practical problems (§4.2). The following part (§4.3) focuses on the performance. A example of application (convergence analysis) is shown for a linear elasticity problem (§4.4). Visualization and interactive usage of our approach is discussed (§4.5). Eventually, the limitations of our approach are summarised (§4.6).

4.1 Validation on the sinus bump problems

When the analytical solution of a problem is known, we can compute the finite element solution error by mapping quadrature points (defined on the reference cell) to world-space and evaluating both the approximated and the exact solution at these mapped points. This procedure is sufficiently fast as it does not involve inversing the mappings. In the following examples, we use this error (computed with high-order quadratures) as the ground truth and we compare our approximated distance to this one. With the first example, we show that our distance computation convergences to the right values for linear and trilinear finite elements. In the second example, we compute the distances to the reference solution of a linear elasticity problem for various finite element basis and show they are the same as the finite element errors computed from quadratures.

First, we consider the *sinus bump problem* which is a Poisson problem of the form $-\Delta u = f$ where $f = 3\pi^2 \sin(\pi x)\sin(\pi y)\sin(\pi z)$, with homogeneous boundary Dirichlet conditions, defined on the unit cube $[0,1]^3$. The known closed-form solution is given by $u = \sin(\pi x)\sin(\pi y)\sin(\pi z)$. We solve this problem with the finite element basis $\mathbb{P}_1$ (linear) and $\mathbb{Q}_1$ (trilinear), the solutions computed on meshes made of respectively 7836 tetrahedra and 3375 hexahedra are shown on the left part of the figure 5. To evaluate the impact of non-linear element mappings, we built the meshes in such a way such that the interior edges and faces do not align with the cube boundaries, so the hexahedron faces are not planar. The reference errors (dotted lines in plot of figure 5) are computed using high-order quadratures available in [2]. It is worth mentioning that to get an accurate L^∞ error, we used quadratures associated to polynomials of order more than 20 (thousands of points per element) as the analytical solution is not polynomial.

For the tetrahedral mesh, the errors plotted in Fig. 5 (top) show that our distance approximations converge quickly to the right values with the number of samples. As expected, the L^∞-distance converges more slowly, since the L^∞-distance measures the field difference where both fields differ the most, and obviously an uniform sampling is not a very efficient way to find this point. Another aspect to consider is that the fields are very smooth, so the number of samples required to converge in this specific case may not be relevant for more difficult problems with more irregular solutions.

In the hexahedral mesh, the interior faces are not planar as the hexahedra geometries are determined by trilinear mappings with bilinear faces. In our distance approximation, we decompose such *curved* cells into sub-tetrahedra. The figure 5 shows the impact of the subdivision level on the computed L^∞-distance. We only plot the L^∞-distance because this is the worst case of the three distances (the same behavior is observed for L^1 and L^2, in a lesser extent). From this example, we observe that a simple decomposition of hexahedra into six tetrahedra is definitively not enough as it leads to significant errors on the distance approximations. Otherwise, we see that two levels of subdivision (that decompose each hexahedron into 384 sub-tetrahedra) is sufficient to approach the exact

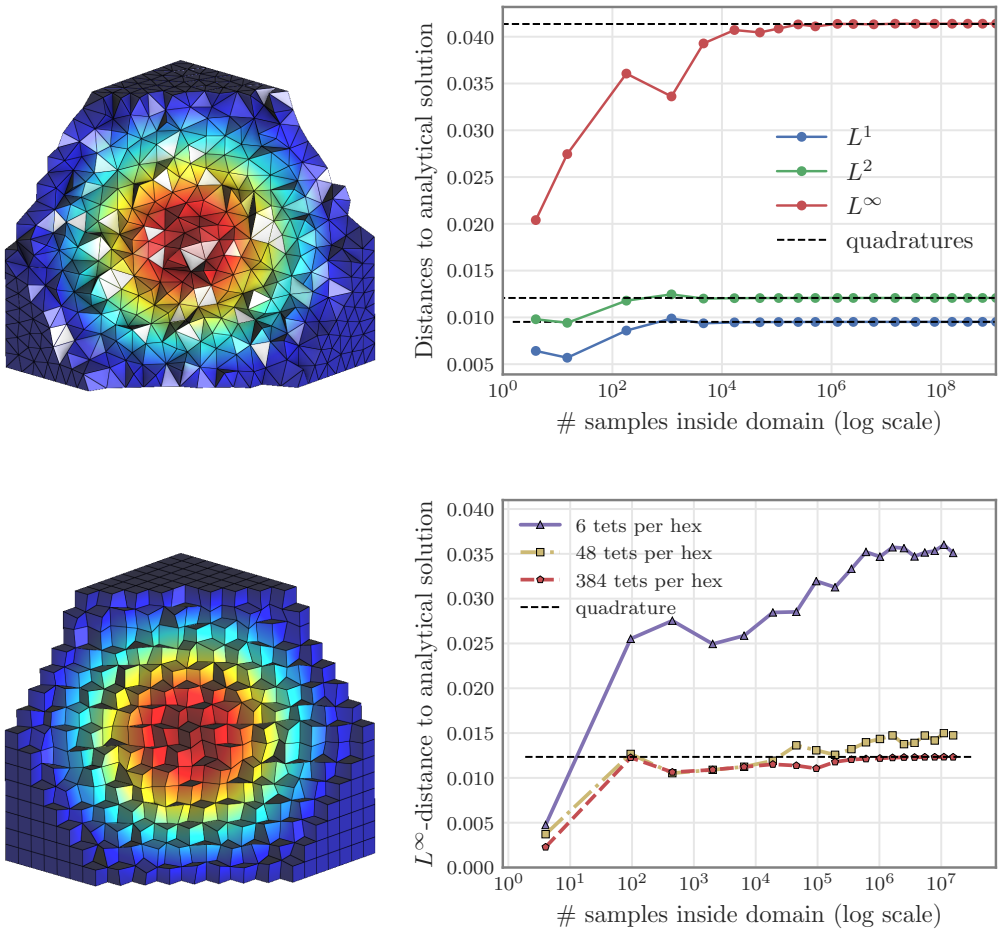


Figure 5: Distances to the analytical solution. (left) $\mathbb{P}_1$ and $\mathbb{Q}_1$ solutions defined on a tetrahedral and a hexahedral mesh of the unit cube. (top-right) Convergence of the distance computations with the number of samples inside the domain, for the $\mathbb{P}_1$ solution of the sine bump problem. (bottom-right) Impact of the subdivision level applied to hexahedra on the L^∞-distance to the reference solution for a $\mathbb{Q}_1$ solution of the sine bump problem. The reference distances (dotted lines) are computed with high-order quadratures.

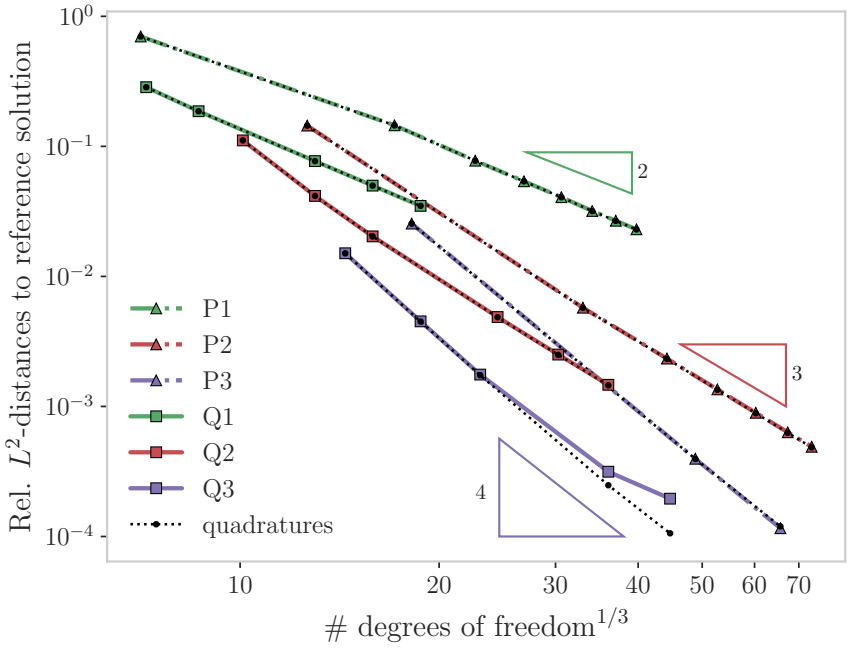


Figure 6: Linear elasticity sine bump problem. Convergence of the Lagrange finite element solutions to the reference solution with the number of degrees of freedom for the L^2-distance. The reference solution is P_4 and defined on a fine mesh. The Rel. L^2-distance is the L^2-distance divided by the L^2-norm of the reference solution. The dotted lines are the L^2 finite element errors computed with high-order quadratures. The loss of convergence in $Q3$ data points is due to insufficient element decompositions in the distance computations.

distance on this problem. A single level of subdivision (that decomposes each hex into 48 tets) can be enough if one is interested in a approximated distance.

The second problem is a linear elasticity problem solved with the standard displacement formulation. It is similar to the first one: unit cube domain, homogeneous Dirichlet boundary conditions and a displacement solution whose components are:

$$\forall i \in [0, 2],\ u_i = \sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$$

The problem is solved with tetrahedral and hexahedral meshes of various size, equipped with Lagrange finite elements of order one to three. We also compute a reference solution with a fine mesh equipped of order four finite elements. For each approximation space, we compute the L^2 distance to the reference solution with our approach and the finite element approximation error with high-order quadratures, as the analytical solution is known. For the distance parameters, we use 500^3 samples and a subdivision level of three for the hexahedron mappings. We show the results as finite element convergence plots (Fig. 6). As the domain is a cube and the meshes are isotropic, the cubic root of the number of degrees of freedom is approximatively proportional to the maximum element size h. For linear elasticity, the L^2 error estimate associated to finite elements of order k is bounded by $\|u - u_h\|_{L^2} < Ch^{k+1}$.

We observe that the distance to a reference solution computed with our approach is

equal to the error computed with quadratures in most cases. However, the computed distance is no longer exact for the refined hexahedral $Q3$ meshes. This error is due to the piecewise-linear approximation of the trilinear mappings. As the finite element approximation is becoming very accurate (inferior to 10^{-3}), the approximation error made on the mapping is becoming dominant. This effect can be mitigated by increasing the subdivision level to push away the issue.

From these validation cases, we conclude that our distance approximation has a good behavior as it converges to the right values when increasing the number of samples and as curved cells can be sufficiently approximated with sub-tetrahedra. Still, it should be noted that high-accuracy measures require fine element decompositions for curved elements. If the decomposition is insufficient, the computed distances can be significantly wrong.

4.2 Parameter sensitivity on non-trivial meshes and fields

In practice, one has to choose a number of samples, and a subdivision level in the case of curved elements. We present two examples that give insights on how to choose them.

Our first case study is a linear elasticity problem defined on a mechanical piece that we referred to as *hanger*. The 3D model, the result of the finite element simulation and examples of meshes are shown in Fig. 12. A detailed study of the convergence of the finite element basis on this problem will be presented in section 4.4. For now we focus on the convergence of the distance computation in function of our approximation parameters. In Fig. 7.a., we study the distance between a $\mathbb{P}_3$ solution obtained on a coarse tetrahedral mesh (86k tetrahedra) and another field which is a $\mathbb{P}_2$ solution defined on a fine mesh (960k tetrahedra). We observe that the distance approximation has converged for less than ten millions of samples. This small number of samples required can be explained by the smoothness of the displacement field. In Fig. 7.b., we compute the distance between a $\mathbb{P}_2$ solution defined on a tetrahedral mesh (209k cells) and a $\mathbb{Q}_2$ solution defined on a hexahedral mesh (36k hexahedra), for which we apply different subdivision level during the L^∞-distance computation. We observe that a single level of subdivision suffices, which corresponds to approximate the geometry of hexahedra with 48 sub-tetrahedra. It is important to notice that a simpler decomposition into six tetrahedra leads to *significantly wrong results*.

The previous example only considers smooth fields (mechanical displacement from linear elasticity), in the next example we will look at the behavior of the distance computation with more irregular fields. To generate random fields with a controlled spatial correlation, we use the Perlin noise algorithm [19]. The experimental results in Fig. 8 show that the convergence of the distance computation is slower for a high frequency field, especially for the L^∞ norm. This behavior is expected, however the convergence for high frequency fields is only slower by a reasonable factor even if the frequency is much higher. This could be explained by the fact that the computed distances (except the L^∞ distances) are global and measure the differences in average, so the global behavior is captured relatively quickly. In our experimental results, we observed that it is very rare to need more than one billion samples inside the domain to converge, except maybe for pathological cases but which are not typical results of finite element simulations. In practice, to ensure that the sampling is sufficient, we compute the distance several times while increase the number of samples until the computed distance is stable.

Our algorithm uses a regular grid of samples to evaluate the distance. We now study the influence of the orientation of the axes in function of the sampling density in Fig. 9 for the same simulations (sinus bump and linear elasticity), by estimating the relative

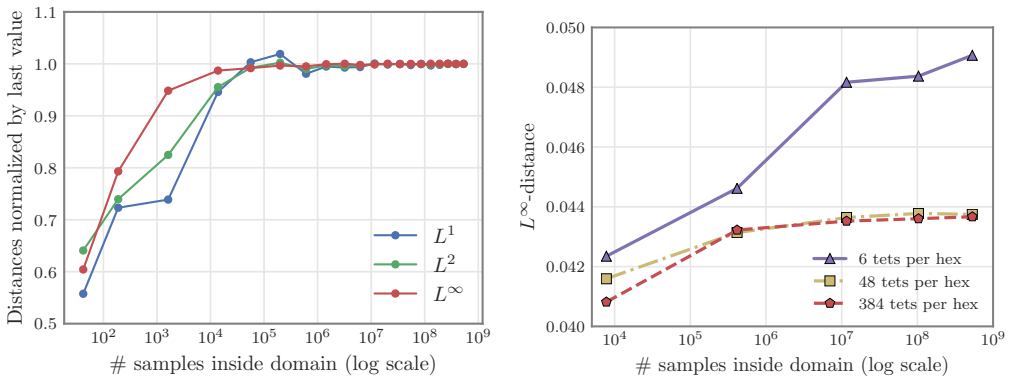


Figure 7: *hanger* linear elasticity problem, displacement solution shown in Fig. 12. *(left)* convergence of the L^1, L^2, L^∞ distances, the distances are divided by their last values so the three distances can be displayed with same scale. *(right)* influence of the subdivision level on the L^∞-distance computation for a hexahedral mesh of the *hanger* model.

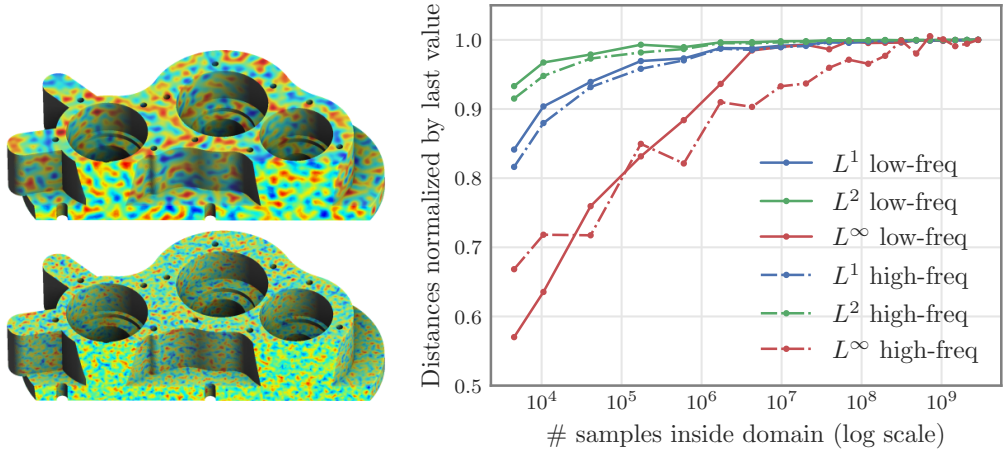


Figure 8: *(left)* cut *carter* model with Perlin random noise scalar fields, one low frequency and one high frequency. *(right)* for both fields, the distances between two approximation basis (one P_1 on a fine mesh and one P_2 on a coarse mesh)

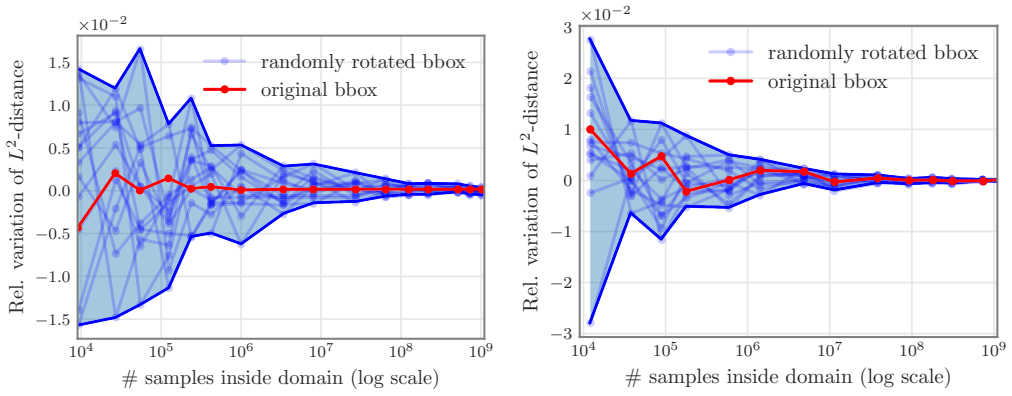


Figure 9: Influence of the axes orientation in function of sampling density. *(left)* Sine bump on the cube. *(right)* Linear elasticity problem on the hanger model.

variation of the computed L^2 distance for randomly rotated axes. As can be seen, with one billion samples, the influence of the orientation becomes negligible.

4.3 Performance

As our approach exploits fast GPU operations, distance computations usually take a few seconds with a mid-range desktop graphics card and less than one second with a high-range one. In this subsection, we show the timings associated to the processing of various meshes and fields presented in this article. The timings are affected by many factors: the number of samples inside the domain, the number of cells in the meshes, the basis used for interpolation inside the cells, the subdivision level used to approximate curved elements, the dimension of the field (scalar or vectorial). So the required time to get a good distance will vary with each model, but the important point is that for standard meshes (up to a few millions of cells), it is always possible to get an accurate distance in less than a few seconds.

The timing reported in Fig. 11 are obtained with a *Nvidia Geforce GTX 1080*. Each pair of fields is described in the table 11. For the hexahedral meshes, we always used two levels of subdivision (each hex is decomposed into 384 tetrahedra), which is the conservative choice according to our empirical results. The reported timings do not include file transfer (from the hard-drive to the system memory) but they do include the transfers from system memory to the GPU memory. As we are mainly interested in the minimal time required to get an accurate distance, we plot the convergence of the L^2, L^∞-distance against the timings on the right part of Fig. 11.

In general, it takes less than one second to process up to one billion of samples inside the domain for medium-sized meshes (between 100k and 1M cells). The hexahedral meshes are processed more slowly, which is expected as each cell is decomposed into hundreds of sub-tetrehadra and this operation takes time, even if it is done directly on the GPU. Looking at the distance convergence (plots at the bottom of the figure), we observe that we often reach a very accurate L^2 computation in less than one second even for large meshes. An accurate L^∞ computation requires more samples for large

[3] Inria Gamma 3D meshes: `https://www-roc.inria.fr/gamma/gamma/download/download.php`

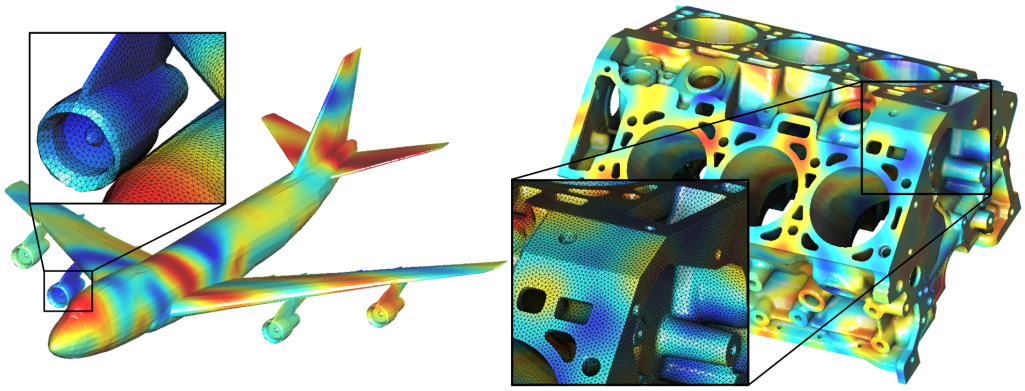


Figure 10: *(left)* Perlin random field on model *747* [3]. *(right)* Perlin random noise on model *40 heads*. The fine tetrahedral meshes are displayed in the zoomed parts.

meshes such as models #5, #6 (displayed in Fig. 10), but it remains in the range of a few seconds. A more complex example involving polynomials of order 5 and 7 on large meshes is presented in Appendix B.

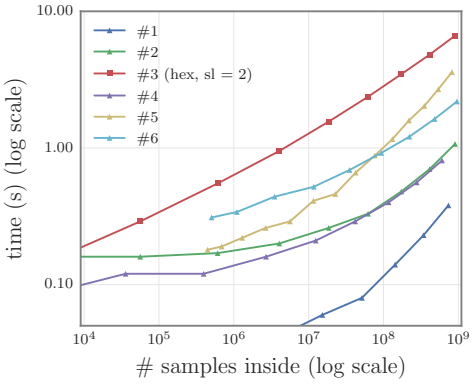
These fast computations allows to compute dozens of distances in minutes, which is very useful for convergence analysis, such the one which is presented in subsection 4.4. For larger meshes (dozens of millions of cells), our approach can scale up with basic mesh decomposition. In this case, the computing time can go up to minutes to get accurate values, but this is still several orders of magnitudes faster as compared with the time required by the corresponding finite element simulations.

4.4 Example of application: accuracy of low-order Lagrange finite element basis on a linear elasticity problem

Analyzing convergence error plots is a standard way to compare different finite element approaches applied to the same problem. The errors can be efficiently computed when an analytical solution is known, but this analytical solution exists only for simple cases: regular domains in 3D (cube, sphere, L-shape) and relatively simple boundary conditions, sources terms and coefficients. Our fast distance computation allows to plot similar error convergences for arbitrary domains and problems, the only drawback is that we have to choose a reference solution that we consider as the ground truth. In the following example we compute distances between solutions that have millions of degrees of freedom in a short amount of time.

We consider a linear elasticity problem defined on the *hanger* model which is shown on Fig. 12. We used the hexahedral mesh available in the dataset of [14]. The inner cylinder on the left is fixed and a uniform force is applied on the inner cylinder on the right. The resulting deformed mesh is also shown in Fig. 12. Tetrahedral meshes of the model with various element sizes are generated by remeshing the boundary with *vorpaline* [1] and meshing the interior with *TetGen* [22]. As hexahedral meshing is a difficult problem, we use successive regular refinements (each hex becomes 8 hexs) of the input model from [14]. We solve the problem using Lagrange finite elements of order one, two and three, available in the *mfem* library [2].

model	dim	# cells	basis
#1: joint (fig. 2)	1	10 k 65 k	P_2 (tet) P_1 (tet)
#2: hanger (fig. 12)	3	1 739 k 343 k	P_1 (tet) P_2 (tet)
#3: hanger (fig. 12)	3	36 k 290 k	Q_2 (hex) Q_1 (hex)
#4: carter (fig. 8)	1	1 210 k 373 k	P_1 (tet) P_2 (tet)
#5: 747 (fig. 10)	1	1 385 k 577 k	P_1 (tet) P_2 (tet)
#6: 40heads (fig. 10)	1	2 905 k 2 350 k	P_1 (tet) P_2 (tet)



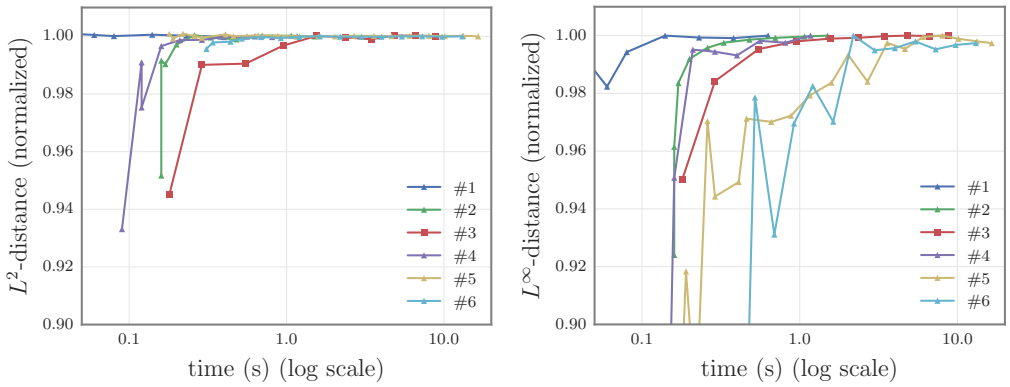


Figure 11: (top-left) Description of the pair of fields. (top-right) Evolution of the computation time with the number of samples inside the domain. (bottom) The L^2, L^∞ computed distances versus the computation time (which is linked to the number of samples in top-right plot). To display the various test cases in the same plots, the L^2 distance is divided by the last value obtained for each model and the L^∞ distance is divided by the maximum of the computed L^∞-distances for each model.

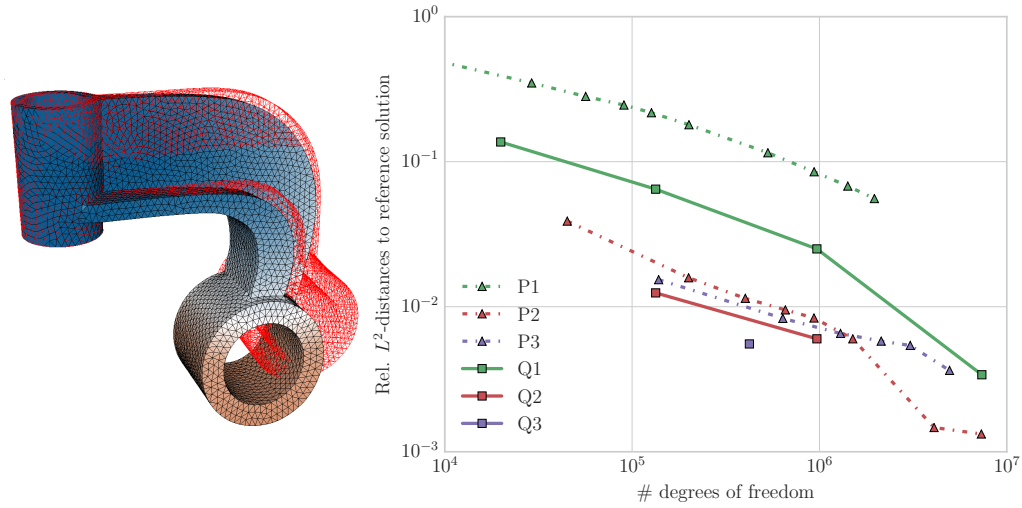


Figure 12: The *hanger* linear elasticity problem. *(left)* Finite element solution, non-deformed mesh in red, the color is the displacement magnitude. *(right)* Convergence of the Lagrange finite element solutions to the reference solution with the number of degrees of freedom for the L^2-distance. The reference solution is P_2 and defined on a fine mesh. The Rel. L^2-distance is the L^2-distance divided by the L^2-norm of the reference solution.

For the error computation, the reference solution is a $\mathbb{P}_3$ solution defined on a fine mesh (960k tetrahedra, 13.6M degrees of freedom), we used 129M of samples inside the domain and two subdivision levels in the hexahedral meshes (each hexahedra is decomposed into 384 tetrahedra to approximate the geometry). In this case, computing the 29 distances shown in Fig. 12 took a total time of 30 seconds, whereas computing the finite element solutions took a dozen of hours.

The computed distances (to the reference solution) for each mesh and basis are shown in the log-log plot of the figure 12. As expected, we observe that (a) for a given basis, refining the mesh (increasing the number of degrees of freedoms in our plots) will decrease the error, (b) increasing the polynomials of the Lagrange elements decreases the error (for a given number of degrees of freedom) and (c) hexahedral elements perform better than tetrahedral elements. Especially, it is interesting to observe that on our problem, trilinear hexahedra are approximatively four times more accurate than linear tetrahedra, whose errors are very large (a well known fact for linear elasticity).

Contrary to the *sine bump* elasticity problem (Fig. 6), we do not observe the convergence rates predicted by the theory: second and third order finite elements converge at the same rate than order one. Our hypothesis is that as there is no body force in the *hanger* problem, the solution accuracies are not sufficient to observe the right convergence rates, which would require very fine meshes.

When performing finite element convergence analysis, one should be aware that the distances should not be interpreted blindly as finite element approximation errors as the reference solution is not exact. What is measured is the convergence to the reference solution and not the convergence to the real one. In practice, a safe approach for analysis is to only observe the solutions that are sufficiently far away from the reference field (e.g. at least three times less accurate).

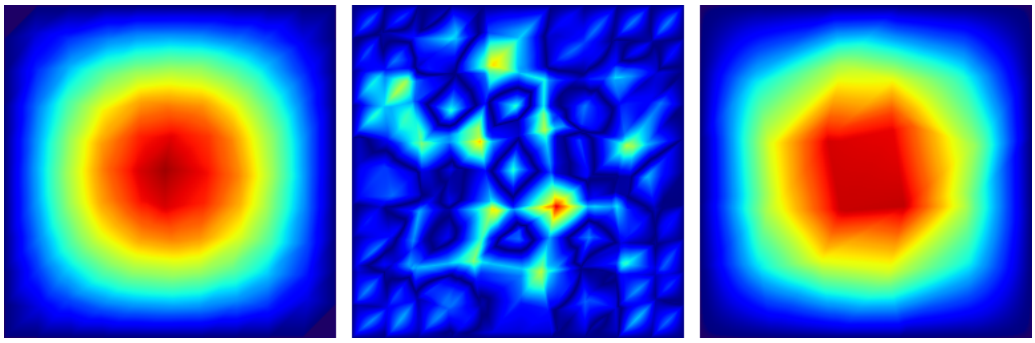


Figure 13: *(left)* Slice of a $\mathbb{Q}_1$ solution of the sine bump problem. *(center)* Slice of absolute difference of both solutions. *(right)* Slice of a $\mathbb{P}_1$ solution of the sine bump problem.

This case study shows that our distance computation allows to quantitatively compare different finite element solutions on complicated and large meshes in a reasonable time. It was applied to the simple displacement formulation of the linear elasticity, but it can be easily adapted to compare solutions on more complicated approximation spaces (Discontinuous-Galerkin, mixed finite elements, etc) and other norms such as H^1.

4.5 Information extraction and interactive visualization

Even if our initial goal is to compute global distances, our pipeline computes the field difference at each sample point. This information can be relevant for further analysis.

Visualization and export In our implementation, the GPU memory contains the values of f, g and $f - g$ at sample points in textures T_f, T_g, T_d at each slice. Visualizing these textures in a graphical interface is straightforward and one can manually examine the field values, as illustrated in figure 13. The visual approach turns to be a valuable tool when trying to understand how element shapes, or any other properties, affect the numerical solutions. The textures can be exported as images, or the whole field voxelization can be saved in a file and analyzed in another software with advanced visualization features (e.g. iso-values visualization, transparency).

Potential extensions More specific tasks can also be easily implemented: extracting the samples where the distance is superior to a given threshold, computing local distances for each cell, etc.

In this work, we only considered the basics L^1, L^2, L^∞ distances, which may not be suited to measure relevant information for certain type of problems. For instance, if one is interested in describing a localized wave front, a global L^p distance, where all the domain will average down the distance, is probably not a relevant measure. Our approach which relies on a voxel grid can be easily adapted, for instance by adding voxel weights, higher for samples closer to the area of interest and lower elsewhere, or by only computing the distance in a localized window.

4.6 Limitations

Our distance approximation suffers from limitations due to the inherent usage of regular sampling, mapping approximation for curved elements and GPU hardware.

Regular sampling. The distance is determined by the values of the fields at regularly spaced samples. We can imagine cases where this sampling totally misses the relevant values of the inputs. This should not happen for sufficiently smooth fields, such as Lagrange finite element solutions, but the users should be aware that regular sampling do not capture efficiently high-frequency features.

The regular sampling has a direction bias (axis of the bounding box) that affects slightly the voxel-approximation of the mesh boundaries. In the same spirit, if one is studying piecewise discontinuous fields, the sampling of the interior interfaces will be affected by the bounding box axis. We claim that these flaws are negligible in practice for most applications (when using a large amount of samples), but one should be aware of them.

Curved element mapping approximations. Non-affine element mappings are approximated by piecewise-affine mappings (i.e. decomposition in tetrahedra). The error due to this approximation depends of the subdivision level. As shown in Fig. 6, this error can become dominant when considering highly accurate finite element fields. A first solution is to increase the subdivision level, but high subdivision levels ($sl > 3$) slow down significantly our approach. An alternative approach could be to use our element decomposition as a first approximation of the reference coordinates, then to refine the coordinates with a few iterations of a Newton-Raphson algorithm applied to the forward mapping.

GPU limitations. Our approach relies heavily on the processing power of GPU hardware, thus a first limitation is to have a sufficiently powerful graphics card. In practice, this excludes computers with a processor-integrated GPU. Considering the driver limitations, our implementation requires OpenGL 4.3, which may not be available on all operating systems. However, it is totally possible to implement our approach on CPU and it would probably have reasonable performance with enough optimization (parallelization, SIMD operations).

Considering the numerical accuracy of our implementation, all the computations are done with 32-bits floats. Support for double precision (64-bits) is poor in OpenGL and it would be more practical to implement it with a computing framework such as OpenCL or CUDA. The usage of 32-bits float has not been an issue in our experience but it could be one when processing very large meshes.

In our implementation, both meshes and their associated field coefficients are uploaded into the GPU memory at the initialization. So a direct limitation is that the inputs should fit in the memory. With our GPU, this limit is 8 GB, which is enough to deal with pretty large meshes. But for larger ones, or for small GPU capacities, a simple improvement is to decompose both inputs into smaller meshes using standard mesh decomposition techniques and then to process them sequentially.

For a given slice, we compute all the sample values at the same time. For large meshes, this can requires to use a large pixel grid (e.g. $5k \times 5k$), which is not well suited for GPUs which are optimized for rendering at screen resolutions. A possibility is to decompose the slices into smaller sub-slices and to process them sequentially (i.e. slice zooms). This decomposition could also be efficient to deal with large meshes composed of many holes or sparse structures, as one could pre-process the model and build a tree to know in advance which sub-slices contain samples or not.

5 Conclusion

In this work, we have presented an efficient L^p distance computation method between fields defined on distinct 3D unstructured meshes of the same model. The computation is based on a regular sampling of both input fields. The efficiency of the approach relies on exploiting GPU hardware that allows to process a very large number of samples in a short amount of time. The accuracy of the computation has been discussed with an analytical solution and the convergence has been shown for various test cases. The speed of our solution makes it a practical and interactive tool, useful for developing and analyzing numerical methods, mainly in the context of the finite element framework. The element-based approach makes it flexible: our implementation naturally supports non-conforming meshes, discontinuous finite element solutions or meshes mixing different type of elements (e.g. hex-dominant meshes).

The approach we propose relies on two parameters: the number of samples and the subdivision level for curved elements. Users should always check that the chosen values are suited for the considered application. For instance, an insufficient element decomposition can lead to distorted convergence rates in finite element convergence analysis. But as the distance computation is fast, it is easy to run the same computation with increased parameter values and to verify that the computed distances are the same.

Our implementation of the distance computation is open-source and can be re-used or modified to accommodate specific needs. The distance computation is extensible: other norms, such as H^1, can be supported. It only requires small adaptations such as changing the *field evaluation* and *slice contribution* steps. One can also extends it by introducing mesh decompositions for very large meshes or zooms to study localized features in the fields (e.g. wavefront).

Acknowledgment

This work was supported by ERC grant ShapeForge (StG-2012-307877) and région Lorraine (France).

References

[1] *Geogram: a programming library of geometric algorithms.* `http://alice.loria.fr/software/geogram/doc/html/index.html`.

[2] *Mfem: Modular finite element methods.* `mfem.org`.

[3] D. Akio and A. Koide, *An efficient method of triangulating equi-valued surfaces by using tetrahedral cells*, IEICE Transactions on Information and Systems, 74 (1991), pp. 214–224.

[4] F. Alauzet, *A parallel matrix-free conservative solution interpolation on unstructured tetrahedral meshes*, Computer Methods in Applied Mechanics and Engineering, 299 (2016), pp. 116–142, doi:10.1016/j.cma.2016.01.021.

[5] M. Brasher and R. Haimes, *Rendering planar cuts through quadratic and cubic finite elements*, in Visualization, 2004. IEEE, IEEE, 2004, pp. 409–416.

[6] P. Bussetta, R. Boman, and J.-P. Ponthot, *Efficient 3d data transfer operators based on numerical integration*, International Journal for Numerical Methods in Engineering, 102 (2015), pp. 892–929, doi:10.1002/nme.4821.

[7] P. E. Farrell, *The addition of fields on different meshes*, Journal of Computational Physics, 230 (2011), pp. 3265–3269, doi:10.1016/j.jcp.2011.01.028.

[8] P. E. Farrell, M. D. Piggott, C. C. Pain, G. J. Gorman, and C. R. G. Wilson, *Conservative interpolation between unstructured meshes via supermesh construction*, Computer Methods in Applied Mechanics and Engineering, 198 (2009), pp. 2632–2642, doi:10.1016/j.cma.2009.03.004.

[9] Y. Feng, D. Perić, and D. Owen, *A non-nested galerkin multi-grid method for solving linear and nonlinear solid mechanics problems*, Computer Methods in Applied Mechanics and Engineering, 144 (1997), pp. 307–325, doi:10.1016/s0045-7825(96)01183-8.

[10] C. Geuzaine, B. Meys, F. Henrotte, P. Dular, and W. Legros, *A galerkin projection method for mixed finite elements*, IEEE transactions on magnetics, 35 (1999), pp. 1438–1441, doi:10.1109/20.767236.

[11] B. Haasdonk, M. Ohlberger, M. Rumpf, A. Schmidt, and K. G. Siebert, *Multiresolution visualization of higher order adaptive finite element simulations*, Computing, 70 (2003), pp. 181–204.

[12] X. Jiao and M. T. Heath, *Common-refinement-based data transfer between non-matching meshes in multiphysics simulations*, International Journal for Numerical Methods in Engineering, 61 (2004), pp. 2402–2427, doi:10.1002/nme.1147.

[13] A. O. Leone, P. Marzano, E. Gobbetti, R. Scateni, and S. Pedinotti, *Discontinuous finite element visualization*, in Proceedings 8th International Symposium on Flow Visualization, 1998.

[14] M. Livesu, A. Sheffer, N. Vining, and M. Tarini, *Practical hex-mesh optimization via edge-cone rectification*, ACM Transactions on Graphics (TOG), 34 (2015), p. 141, doi:10.1145/2766905.

[15] S. Menon and D. P. Schmidt, *Conservative interpolation on unstructured polyhedral meshes: An extension of the supermesh approach to cell-centered finite-volume variables*, Computer Methods in Applied Mechanics and Engineering, 200 (2011), pp. 2797–2804, doi:10.1016/j.cma.2011.04.025.

[16] B. Nelson and R. M. Kirby, *Ray-tracing polymorphic multidomain spectral/hp elements for isosurface rendering*, IEEE Transactions on Visualization and Computer Graphics, 12 (2006), pp. 114–125, doi:10.1109/tvcg.2006.12.

[17] B. Nelson, R. M. Kirby, and R. Haimes, *Gpu-based interactive cut-surface extraction from high-order finite element fields*, IEEE Transactions on Visualization and Computer Graphics, 17 (2011), pp. 1803–1811, doi:10.1109/tvcg.2011.206.

[18] J. Peraire, J. Peiro, and K. Morgan, *Multigrid solution of the 3-d compressible euler equations on unstructured tetrahedral grids*, International Journal for Numerical Methods in Engineering, 36 (1993), pp. 1029–1044, doi:10.1002/nme.1620360610.

[19] K. Perlin, *An image synthesizer*, SIGGRAPH Comput. Graph., 19 (1985), pp. 287–296, doi:10.1145/325165.325247.

[20] S. Plimpton, B. Hendrickson, and J. Stewart, *A parallel rendezvous algorithm for interpolation between multiple grids*, in Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC '98, Washington, DC, USA, 1998, IEEE Computer Society, pp. 1–8, doi:10.1109/sc.1998.10032.

[21] J.-F. Remacle, N. Chevaugeon, E. Marchandise, and C. Geuzaine, *Efficient visualization of high-order finite elements*, International Journal for Numerical Methods in Engineering, 69 (2007), pp. 750–771, doi:10.1002/nme.1787.

[22] H. Si, *Tetgen, a delaunay-based quality tetrahedral mesh generator*, ACM Transactions on Mathematical Software (TOMS), 41 (2015), p. 11, doi:10.1145/2629697.

[23] S. R. Slattery, P. P. H. Wilson, and R. P. Pawlowski, *The data transfer kit: A geometric rendezvous-based tool for multiphysics data transfer*, in Proceedings of the 2013 International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering, 2013.

[24] G. M. Treece, R. W. Prager, and A. H. Gee, *Regularised marching tetrahedra: improved iso-surface extraction*, Computers & Graphics, 23 (1999), pp. 583–598, doi:10.1016/s0097-8493(99)00076-x.

[25] M. Üffinger, S. Frey, and T. Ertl, *Interactive high-quality visualization of higher-order finite elements*, in Computer Graphics Forum, vol. 29, Wiley Online Library, 2010, pp. 337–346, doi:10.1111/j.1467-8659.2009.01603.x.

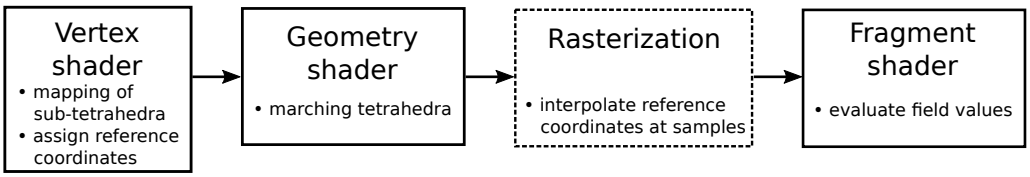


Figure 14: OpenGL rendering pipeline used to sample a field on a pixel grid

A GPU implementation details

Graphic cards (GPU) deal efficiently with highly parallelizable tasks and have dedicated hardware to rasterize triangles and to compute linear interpolation inside them. The goal of our implementation is to take full advantage of these properties.

Our implementation follows the algorithms 1 and 3. The adapted version corresponding to our OpenGL implementation in summarised with algorithm 4. The different steps are discussed in the next paragraphs.

Algorithm 4 Overview of OpenGL implementation

for each slice k in $[1, n_s]$ **do**
  render the field slice of f in texture T_f ▷ *rendering pipeline*
  render the field slice of g in texture T_g ▷ *rendering pipeline*
  compute the texture difference $T_d = T_f - T_g$ ▷ *texture difference*
  reduce texture T_d in the k-th column of the texture T_N ▷ *texture reduction*
end for
transfer the texture T_N from the GPU to the CPU
compute the global distance from T_N values

A.1 Rendering pipeline

Instead of rendering to the screen, we render to a *framebuffer object* which is a rectangular grid of size $n_x * n_y$, such as 1000x800. In practice, the size of the grid is determined by the resolution parameter h which determines the voxel grid dimensions (samples are equally spaced in the three axis). With OpenGL, instead of rendering RGBA colors, it is possible to produce floating-point values (32 bits), so all our texture data types are GL_FLOAT.

The *rendering pipeline* is called at each slice of a mesh via a call to the glDrawElementsInstanced() [4] command, with the primitive argument GL_LINES_ADJACENCY, which indicates that we pack vertices four by four for processing by the geometry shader. An overview of our rendering pipeline is given in Fig. 14.

When calling glDrawElementsInstanced(), we draw multiple *instances* of a group of elements. In our case, the group of elements corresponds to our reference element. For tetrahedral meshes, it will be a single tetrahedron. In the case of curved elements, it will be the sub-tetrahedra of the reference element decomposition.

The OpenGL computing blocks are called shaders. Each shader, or stage, is responsible for certain tasks. In our implementation, the tasks are distributed as follows:

[4] OpenGL 4.5 Reference Pages `https://www.opengl.org/sdk/docs/man4/`

a) The *vertex shader* has tetrahedra in input (the element decomposition) and it applies the element mappings. The reference coordinates are associated to each tetrahedron vertices (e.g. $(0, 0, 0)$ is associated to the first vertex of the current tetrahedron, $(1, 0, 0)$ to the second one, etc.). The mapping coefficients are passed to the vertex shader via OpenGL attributes, which change for each instance.

b) The *geometry shader* is executed one time for each tetrahedra, it runs the marching tetrahedra algorithm and outputs 0, 1 or 2 triangles that have the interpolated reference coordinates at their vertices.

c) The *rasterization step* is done automatically by OpenGL. It interpolates linearly the reference coordinates at pixel centers of the grid which are inside the triangle and it calls the fragment shader for each of them.

d) The *fragment shader* is responsible for producing the rendering output. In our approach, it receives the reference coordinates (from the rasterization) and compute the field values by applying the interpolation function (which requires access to the field coefficients). We further discuss this step in the next paragraph.

If the field is composed of multiple types of elements (e.g. tetrahedra and hexahedra), we call the rendering pipeline one time for each type of element. If it is composed of multiple finite element basis, we also call the rendering for each one.

Field values interpolation One could transfer the reference coordinates from the GPU to the CPU and evaluate the interpolation functions on the CPU. However, data transfers between CPU and GPU are slow, and transferring two large grid of values (one per field) at each slice takes far more time than the rendering, thus we advocate to maximise the amount of computations done on the GPU and to minimize the number of transfers. So, we evaluate the interpolation functions on the GPU. This requires to upload (only one time at the initialization) the field coefficients to the GPU memory and to access them during the field evaluation in the fragment shader. Consider a mesh with one million cells, 10 degrees of freedom per element, e.g. $\mathbb{P}_2$ tetrahedra, and a vectorial field of dimension 3, e.g. displacement in linear elasticity. In such case, there are 30 millions floating-point values which need to be accessed by the GPU. Such amount of data does not fit in the memory of textures, uniform buffer objects or buffer textures. One possibility could be to divide the data in many textures but this would complicate the implementation. Instead we use the Shader Storage Buffer Objects (SSBO) that have been introduced in OpenGL 4.3., as their maximal size is typically the GPU memory.

The **output** of the rendering pipeline is a texture with the field values (or *no data value* at pixels which are not inside the domain). If the field is scalar, the texture format is GL_RED (single value per pixel). If the field is vectorial, we use respectively the types GL_RG, GL_RGB, GL_RGBA for dimensions two, three and four. For higher dimensions, one needs to use multiple output textures.

A.2 Texture difference

As both textures T_f, T_g associated to fields f and g are available on the GPU memory, we can easily compute the difference T_d in another texture by drawing a quad that call the fragment shader for each pixel. Then, the fragment shader computes a simple subtraction (that acts component per component if the field is vectorial).

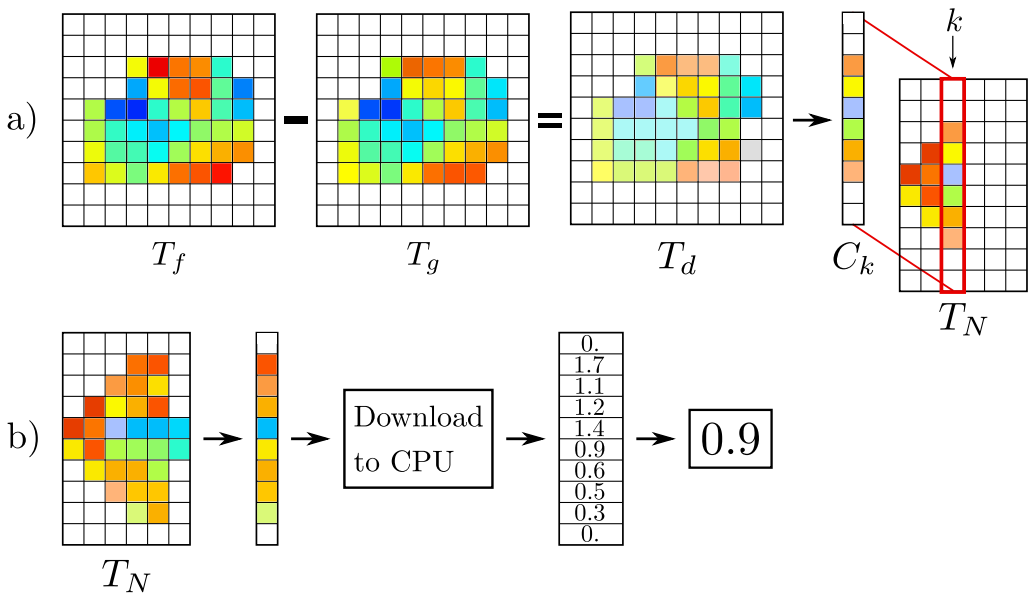


Figure 15: Distance computation from the field samplings. (a) Processing of slicing plane k. (b) Final reduction of the slice contributions to the global distance

A.3 Texture reduction

As we said before, transfer times between GPU and CPU are slow and can quickly become the performance bottleneck. So we process the difference texture T_d directly on the GPU.

We store the contribution of a slice to the global distances in a texture T_N. T_N is a texture of size $n_y * n_s$ in which each column is the contribution from the slice difference k, denoted C_k. The slice contribution C_k is a column vector where each component is the sum of values along the associated row of T_d. This is illustrated in Fig. 15.a. We do not compute only one contribution value per slice because we want to exploit the GPU parallelism, and aggregating the values of one column to one value would not be efficient (the texture dimensions are not power of two).

When the field is vectorial, we combine the different components with root mean square formula during the reduction. Also, we apply the adapted p-power of the L^p-distance, or we took the absolute maximum of the difference for the L^∞-distance. To compute the global distance in the end, one needs to know the number of samples inside the domain for each slice. This number is stored as a component of the T_N texture, which is multivalued.

Eventually, one needs to combine the values of T_N into one 1D texture, then to transfer it on the CPU, then to process it to get the global distance (illustration in Fig. 15.b.).

For efficiency, we compute multiple distances at a time by storing various contributions in the multivalued texture T_N. In our implementation, we compute the L^1, L^2, L^∞ distances but this can be easily changed.

Remark 1 *It is entirely possible to adopt other reduction strategies to go from 2D textures to one-value contributions, but one should be aware that our textures are not power of two nor squares.*

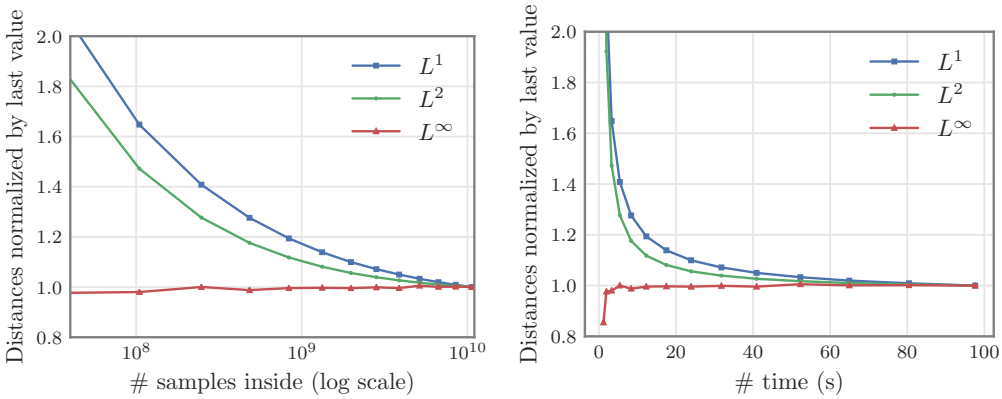


Figure 16: Convergence and performance on large tetrahedral meshes (2 905k and 2 350k cells) with high-order polynomials (P_5 and P_7) *(left)* Convergence of the distance computations with respect to the number of samples inside both meshes. *(right)* Convergence with respect to the computation time.

B Example with high-order polynomials on large meshes

In this example, we compute the distances between two fields defined on two unrelated tetrahedral meshes of the model *40 heads* shown in Fig. 10. This 3D model has many complicated geometrical features and covers only 20% of its bounding box. The field is a Perlin noise [19] projected onto two approximation spaces: the first one uses $\mathbb{P}_5$ elements (56 degrees of freedom) on 2 905k tetrahedra and the other one uses $\mathbb{P}_7$ elements (120 degrees of freedom) on 2 305k tetrahedra. These spaces have 64M and 146M degrees of freedom respectively. This example involves large voxel grids (up to 3752x3067x2200) and requires $3GB$ of memory on the GPU.

The convergence of the L^1, L^2 and L^∞ distance computations with respect to the number of samples and to the timings are shown in Fig. 16. We see that the convergence is slower than with the low-order examples as it requires many samples (similarly to quadrature requirements for such orders). Moreover, evaluating high-order polynomials at each sample is more expensive. However, it remains reasonably fast (one minute) considering the size of the discretized fields (64M vs 146M degrees of freedom).

Our implementation scales up to Lagrange elements of order 9 but it starts to be slow because we did not implement optimized evaluations of high-order polynomials yet.