X-CAD: Optimizing CAD Models with Extended Finite Elements

CHRISTIAN HAFNER, IST Austria
CHRISTIAN SCHUMACHER, Disney Research
ESPEN KNOOP, Disney Research
THOMAS AUZINGER, IST Austria
BERND BICKEL, IST Austria
MORITZ BÄCHER, Disney Research

For readers less familiar with moment fitting, or the method of Müller et al. [2017; 2013], we contrast our hierarchical integration with their scheme. Moreover, we provide a proof of the claim that some terms can be kept constant when computing quadrature rule derivatives. In a final section, we will provide a detailed description of the enrichment of vertices and elements for readers interested in implementing the technique.

ACM Reference Format:
Christian Hafner, Christian Schumacher, Espen Knoop, Thomas Auzinger, Bernd Bickel, and Moritz Bächer. 2019. X-CAD: Optimizing CAD Models with Extended Finite Elements. *ACM Trans. Graph.* 38, 6, Article 1 (November 2019), 6 pages. https://doi.org/10.1145/3355089.3356576

1 NUMERICAL INTEGRATION

Similar to Müller et al. [2013], we build integration schemes in a hierarchical manner (see Fig. 1): We use edge and curve rules to integrate over areas and surfaces, and area and surface rules to integrate over volumetric domains. Hereafter, we will contrast our method with Müller et al. [2013], illustrating shortcomings of their technique for our application domain. As in the main text, we use $g : \mathbb{R}^3 \to \mathbb{R}$ to denote an arbitrary function defined on the volumetric domain enclosed by the boundary representation of a CAD model.

1.1 Integrating along Edges and Curves

For the generation of integration rules along edges and curves (in blue and yellow in Fig. 1), we represent these 1D domains with discrete sets of ordered sample points. For a description of integration along straight, axis-aligned segments E, we point the reader to the main text.

To numerically integrate along planar curves (yellow in Fig. 1)

$$\int_C g(\mathbf{X})\,\mathrm{d}s \approx \sum_{j=1}^{n} w_j g(\mathbf{X}_j),$$

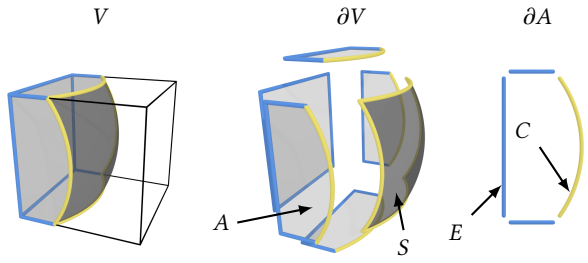


Fig. 1. **Nesting of Integration Rules** To construct rules for integration over volumes V (left), we rely on rules for integration over the volume's boundary ∂V, decomposed into planar areas A (in light gray) and curved surfaces S (in dark gray). To generate rules for area and surface integrals, in turn (middle), we make use of the divergence theorem, expressing integrals along the boundary ∂A with integrals along straight E and curved C segments (right).

Müller et al. [2013] propose to express curve integrals with sums of integrals along edges, then construct quadrature rules using moment-fitting.

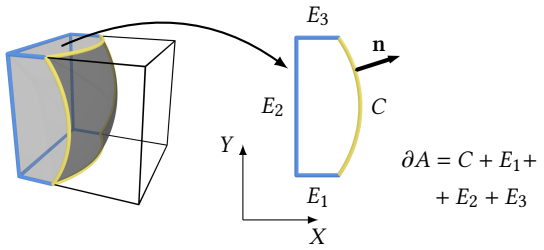


Fig. 2. **Integrating along Curves** To integrate along planar curves C, Müller et al. [2013] propose to construct a divergence-free basis to express curve integrals with a sum of integrals along straight edges (E_1, E_2, E_3). Normals $\mathbf{n}$ point outward and lie in the same plane as the curve.

They propose to use a basis spanning the space of vector-valued polynomials $\mathbf{P}_i : \mathbb{R}^2 \to \mathbb{R}^2, i = 1, \ldots, m$ up to a certain degree that have zero divergence, i.e. $\nabla \cdot \mathbf{P}_i = 0$. For example, to integrate along a planar curve with constant Z-coordinate (see Fig. 2 left), they solve a linear system $\mathbf{A}\mathbf{w} = \mathbf{b}$ for the unknown weights collected in the n-vector $\mathbf{w}$. To form the $m \times n$-system matrix, they take the dot product between basis functions, evaluated at quadrature points (X_j, Y_j), with the corresponding outward-facing in-plane normal (Fig. 2 right), resulting in entries $A_{ij} = \mathbf{P}_i(X_j, Y_j) \cdot \mathbf{n}(X_j, Y_j)$. Note

Authors' addresses: Christian Hafner, IST Austria; Christian Schumacher, Disney Research; Espen Knoop, Disney Research; Thomas Auzinger, IST Austria; Bernd Bickel, IST Austria; Moritz Bächer, Disney Research.

0730-0301/2019/11-ART1 $15.00
https://doi.org/10.1145/3355089.3356576

that, unlike for Newton-Cotes rules, the right-hand side with entries $b_i = \int_{\partial A} \mathbf{P}_i(X,Y)\cdot\mathbf{n}(X,Y)\,ds$ cannot be analytically integrated. This is because parts of the integration domain are curved.

However, because the boundary ∂A of the domain A can be partitioned into the curved domain C and a set of straight edges E_k (compare with Fig. 2), we can rewrite the integral as

$$\int_C \mathbf{P}_i \cdot \mathbf{n}\,ds = \int_{\partial A} \mathbf{P}_i \cdot \mathbf{n}\,ds - \sum_k \int_{E_k} \mathbf{P}_i \cdot \mathbf{n}\,ds. \tag{1}$$

The key benefit of using a divergence-free basis becomes apparent when we apply the divergence theorem to the integral over ∂A

$$\int_{\partial A} \mathbf{P}_i \cdot \mathbf{n}\,ds = \int_A \nabla\cdot\mathbf{P}_i\,dA = 0. \tag{2}$$

Hence, we can numerically integrate along *coordinate axes* to compute the right-hand side entries $b_i = -\sum_k \int_{E_k} \mathbf{P}_i(X,Y)\cdot\mathbf{n}(X,Y)\,ds$.

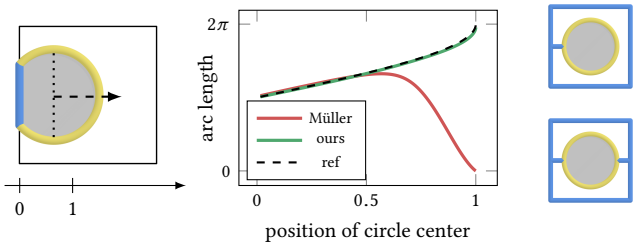
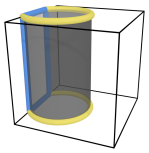
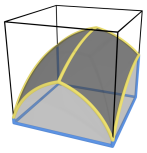


Fig. 3. **Integrating Features** While our integration scheme integrates features smaller than the grid resolution accurately (green curve in error plot, middle), Müller et al.'s method [2013] fails to compute the arc length of a circle C if the length of edge segment E becomes too short (red curve, analytical arc length in black). To mitigate inaccuracies in Müller et al.'s scheme, we could introduce cuts (right): note that a single cut (top) is not sufficient; only if we introduced two cuts integration accuracy is acceptable (bottom).



Targeting numerical integration over *implicitly* defined domains, Müller et al.'s scheme [2013] works well if the straight edge segments are sufficiently long. However, for our application domain, their scheme is ill-suited as we illustrate with the intersection of a cylindrical feature (see inset on the right) with one of the grid planes in Fig. 3: the integration error increases with decreasing length of the edge segment E (in blue) if we compute the arc length of the circular intersection curve (yellow) with an integral along C (see error plot, middle). If the cylinder intersects the grid face in the interior (right), the domain C is *closed* and the integral $\int_C \mathbf{P}_i \cdot \mathbf{n}\,ds$ is zero by construction. While we could introduce controlled cuts (right) to mitigate the problem, it is unclear *how* we could detect problematic cases, and *where* to best introduce cuts. Note that for our cylinder example, one cut (right, top) is not sufficient (integral remains zero). Only if we introduced a second cut (right, bottom), we could integrate along the circle with sufficient accuracy.



A case that arises in our application domain, and that Müller et al. [2013] cannot handle with their technique, is the integration along *spatial* curves formed by two or more NURBS patches that intersect within hexahedral elements (see inset on the left).

To accurately integrate along planar grid-patch or spatial patch-patch intersections, we parameterize curves $\mathbf{r}(t) = [X(t), Y(t), Z(t)]^T$ with t varying between the two end points a and b, and form line integrals

$$\int_C f(\mathbf{X})\,ds = \int_a^b f(\mathbf{r}(t))\,\|\mathbf{r}'(t)\|\,dt, \tag{3}$$

where we assume $\mathbf{r}(t)$ to be sufficiently smooth for an analytical derivative $\mathbf{r}' = \frac{d\mathbf{r}}{dt}$ to exist. Analogously to the axis-aligned case, we use a Gauss-Legendre rule for integration.

For planar curves parallel to one of the coordinate planes, one of the three coordinates remains fixed, e.g. $f(X(t), Y(t), Z)$ for a planar curve parallel to the XY-plane. However, an explicit differentiation between the planar or spatial case is not necessary.

While we can expect intersection curves to be sufficiently smooth, we cannot, in general, extract analytical parameterizations. Hence, we represent parameterizations with a set of sample points $\mathbf{r}_j$, approximating its parametric form with a Lagrange interpolating polynomial of degree d

$$\mathbf{r}(t) = \sum_{j=0}^{d} \left(\prod_{k=0,k\neq j}^{d} \frac{t - t_k}{t_j - t_k} \right) \mathbf{r}_j. \tag{4}$$

Because the integrand g is well-defined in a neighborhood of C, the accuracy of the Gauss-Legendre scheme is preserved if we choose d sufficiently large [Atkinson and Venturino 1993]. Hence, the resulting scheme with approximate $\mathbf{r}$ has excellent accuracy.

1.2 Integrating over Areas and Surfaces

For a discussion of area integrals, we point the reader to the main text, focusing below description on differences between the two techniques for integrals over curved 2D domains.

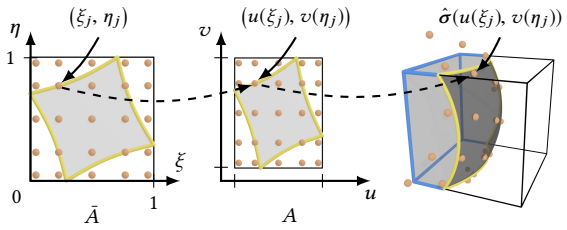


Fig. 4. **Integrating over Surfaces** To integrate over curved domains S (right), we first compute an axis-aligned bounding box in the parameter domain (middle), then transform the integral to the isoparametric domain (left). In the domain $\bar{A}$, we use the *same* quadrature points (ξ_j, η_j) as for area integrals, transforming them back to spatial coordinates $\hat{\boldsymbol{\sigma}}(u(\xi_j), v(\eta_j))$ after rule construction.

Surface Integrals. For surface integrals

$$\int_S f(\mathbf{X})\,\mathrm{d}S \approx \sum_{j=1}^{n} w_j g(\mathbf{X}_j), \tag{5}$$

we utilize the parametric form of NURBS patches to determine the weights w_j and quadrature points $\mathbf{X}_j$, while Müller et al. [2013] propose to use a divergence-free basis. To contrast our technique, we first summarize Müller et al.'s approach.

Analogously to curve integrals, Müller et al. [2013] propose to replace the m monomials with an associated divergence-free $3m$-basis spanned by functions $\mathbf{P}_i : \mathbb{R}^3 \to \mathbb{R}^3$ that are multiplied with the outward-facing *surface* normal when evaluating entries $A_{ij} = \mathbf{P}_i(\mathbf{X}_j) \cdot \mathbf{n}(\mathbf{X}_j)$ and $b_i = \int_S \mathbf{P}_i(\mathbf{X}) \cdot \mathbf{n}(\mathbf{X})\,\mathrm{d}S$. With the help of the divergence theorem and the partitioning of the boundary ∂V of volume V into planar areas A_k and curved surface S (compare with Fig. 1 middle), they solve the system for the right-hand side $b_i = -\sum_k \int_{A_k} \mathbf{P}_i(\mathbf{X}) \cdot \mathbf{n}(\mathbf{X})\,\mathrm{d}A$ instead. However, the resulting rules suffer from similar issues as their one-dimensional counterparts: if the areas A_k become to small in size, the error increases uncontrollably.

As described in the main text, we instead make use of the parameterization $\hat{\boldsymbol{\sigma}}$ of NURBS patches, expressing surface integrals as area integrals in parameter space

$$\int_S f(\mathbf{X})\,\mathrm{d}S = \int_A g(\hat{\boldsymbol{\sigma}}(u,v))\,\|\hat{\boldsymbol{\sigma}}_u(u,v) \times \hat{\boldsymbol{\sigma}}_v(u,v)\|\,\mathrm{d}A. \tag{6}$$

We then proceed analogously to area integrals, transforming the integral with mappings $u(\xi)$ and $v(\eta)$ to the isoparametric domain to make the axis-aligned bounding box of the transformed A coincide with the unit square (see Fig. 2), followed by moment-fitting.

1.3 Integrating over Volumes

To integrate over volumes

$$\int_V f(\mathbf{X})\,\mathrm{d}V \approx \sum_{j=1}^{n} f(\mathbf{X}_j), \tag{7}$$

we proceed analogous to area integration (compare with Fig. 5).

Transformation. We first transform the domain V to make its bounding box $[a,b] \times [c,d] \times [e,f]$ coincide with the unit cube, applying the non-uniform scaling

$$\underbrace{\begin{pmatrix} X(\xi) \\ Y(\eta) \\ Z(\zeta) \end{pmatrix}}_{\mathbf{X}(\boldsymbol{\xi})} = \underbrace{\begin{pmatrix} b-a & & \\ & d-c & \\ & & f-e \end{pmatrix}}_{\mathbf{S}} \underbrace{\begin{pmatrix} \xi \\ \eta \\ \zeta \end{pmatrix}}_{\boldsymbol{\xi}} + \underbrace{\begin{pmatrix} a \\ c \\ e \end{pmatrix}}_{\boldsymbol{t}}. \tag{8}$$

The transformed integral

$$\int_{\bar{V}} f(\mathbf{X}(\boldsymbol{\xi}))\,\det\left(\frac{\partial \mathbf{X}(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}}\right)\,\mathrm{d}\bar{V} \tag{9}$$

is then multiplied with the constant determinant of the Jacobian of the mapping $\det(\mathbf{S}) = (b-a)(d-c)(f-e)$.

Moment-Fitting. Choosing a m-basis with three variables, we form the system $A_{ij} = p_i(\boldsymbol{\xi}_j)$ and $b_i = \int_{\bar{V}} p_i(\boldsymbol{\xi})\,\mathrm{d}\bar{V}$. $\boldsymbol{\xi}_j$ are Gauss-Legendre quadrature points in the unit cube [Müller et al. 2013]. Note that $\mathbf{A}$ is again the same for *all* our volume integrals, reducing the time complexity of our hierarchical scheme significantly.

To evaluate the right-hand side, we form the antiderivative

$$\mathbf{P}_i(\boldsymbol{\xi}) = \frac{1}{3}\begin{pmatrix} \int p_i(\xi,\eta,\zeta)\,\mathrm{d}\xi \\ \int p_i(\xi,\eta,\zeta)\,\mathrm{d}\eta \\ \int p_i(\xi,\eta,\zeta)\,\mathrm{d}\zeta \end{pmatrix} \tag{10}$$

and apply the divergence theorem

$$\int_{\bar{V}} p_i(\boldsymbol{\xi})\mathrm{d}\bar{V} = \int_{\partial\bar{V}} \mathbf{n}(\boldsymbol{\xi}) \cdot \mathbf{P}_i(\boldsymbol{\xi})\,\mathrm{d}\bar{S}. \tag{11}$$

Because the boundary $\partial\bar{V}$ of the volume $\bar{V}$ is partitioned into area $\bar{A}$ and surface integrals $\bar{S}$, we can reuse rules developed in the previous section, establishing a second and final layer of nesting.

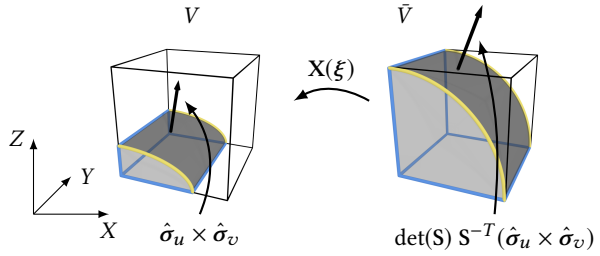


Fig. 5. **Integrating over Volumes** To integrate over a volumetric domain V, we non-uniformly scale (and translate) the axis-aligned bounding box of the volume to the unit cube $\bar{V}$. For consistency, normals $\hat{\boldsymbol{\sigma}}_u \times \hat{\boldsymbol{\sigma}}_v$ on curved surfaces need to be transformed before we can apply surface area rules. We use the linear transformation rule for cross products to do so.

An important detail is that the area and surface domains need to be transformed to account for the non-uniform scaling $\boldsymbol{\xi} \mapsto \mathbf{S}\boldsymbol{\xi} + \mathbf{t}$ to the unit cube. While the determinant of the Jacobian of the mapping is taken into account in Eq. 9 already, positions and vectors have to be transformed accordingly. This means that for our area and surface integrals, we compose the affine mapping (scaling) with the parameterization $(u,v) \mapsto \hat{\boldsymbol{\sigma}}(u,v)$. Our surface integrals require taking special care: while we apply the full transformation to points $\mathbf{x} = \mathbf{S}\hat{\boldsymbol{\sigma}}(u,v) + \mathbf{t}$, only the scaling $\mathbf{S}$ matters when transforming vectors $\hat{\boldsymbol{\sigma}}_u$ and $\hat{\boldsymbol{\sigma}}_v$ (see Fig. 5). We use the rule for linear transformations of cross products to account for this scaling in our surface integrals

$$\int_A f(\mathbf{X})\|\mathbf{S}\hat{\boldsymbol{\sigma}}_u \times \mathbf{S}\hat{\boldsymbol{\sigma}}_v\|\,\mathrm{d}A = \int_A f(\mathbf{X})\det(\mathbf{S})\|\mathbf{S}^{-T}\hat{\boldsymbol{\sigma}}_u \times \hat{\boldsymbol{\sigma}}_v\|\,\mathrm{d}A. \tag{12}$$

1.4 Technical Details

Polynomial Bases. As suggested by Müller et al. [2013], we use an orthonormalized basis, running a Gram-Schmidt algorithm with inner product

$$(p_1, p_2) = \int_{[0,1]^{dim}} p_1 p_2\,d\boldsymbol{\xi} \tag{13}$$

on pairs of basis functions. As observed by Müller et al. [2013], the use of an orthonormalized basis significantly increases numerical stability (rows of matrices $\mathbf{A}$ are sufficiently independent, even for higher-order polynomials).

2 QUADRATURE RULE DERIVATIVES

Quadrature rules for integrating over areas, surfaces, and volumes rely on placing quadrature points within an approximate bounding box of the respective domain. Bounding boxes are updated after every shape parameter change in order to keep a good approximation at all times. As a consequence, quadrature point locations stay within, or at least close to, the integration domain, which stabilizes the quadrature procedure. Thus, not only quadrature weights, but also quadrature point locations, depend on shape parameters. Initially, this seems like an adverse effect, because the dependence causes an additional term to appear in the expression for shape derivatives of the simulation result. Here, we prove the claim made in Sec. 6.3 of the main article, namely, that this terms exactly cancels another term, and thus *simplifies* computation instead of complicating it.

First, we give a concrete example to demonstrate the scope of extra computation that location dependence usually introduces: the shape derivatives of the equilibrium state in a hyperelastic simulation. Let the total potential energy $E(\mathbf{u}, \mathbf{p})$ be given as a function of displacements $\mathbf{u}$ and shape parameters $\mathbf{p}$. The equilibrium state $\mathbf{u}^*$ for a given shape parameter value $\mathbf{p}^*$ is characterized by $\partial_{\mathbf{u}} E(\mathbf{u}^*, \mathbf{p}^*) = 0$. The implicit function theorem shows that there exists a function $\mathbf{u}(\mathbf{p})$ in a neighborhood of $\mathbf{p}^*$ such that $\mathbf{u}(\mathbf{p}^*) = \mathbf{u}^*$ and $\partial_{\mathbf{u}} E(\mathbf{u}(\mathbf{p}), \mathbf{p}) = 0$ in that neighborhood. The derivative of that function at $\mathbf{p}^*$ is given by

$$\partial_{\mathbf{u},\mathbf{u}} E(\mathbf{u}^*, \mathbf{p}^*)\, \mathrm{d}_{\mathbf{p}} \mathbf{u}(\mathbf{p}^*) = \partial_{\mathbf{u},\mathbf{p}} E(\mathbf{u}^*, \mathbf{p}^*). \tag{14}$$

Note that code for computing the energy Hessian, $\partial_{\mathbf{u},\mathbf{u}} E$, is part of any finite-element package, and thus readily available. The right-hand side, $\partial_{\mathbf{u},\mathbf{p}} E$, is the shape derivative of the global force vector and is given by a sum of contributions from all quadrature points in all elements. We focus our attention to the contribution from a single element, and its set of quadrature points with locations $\mathbf{X}_j \in \mathbb{R}^3$ and weights $w_j \in \mathbb{R}$. Writing $\mathbf{g}$ for the force density, and $\mathbf{f}_e = \partial_{\mathbf{u}} E_e$ for the element force vector, integrated over an element domain Ω, we have

$$\mathrm{d}_{\mathbf{p}} \mathbf{f}_e = \mathrm{d}_{\mathbf{p}} \int_\Omega \mathbf{g}(\mathbf{X})\, \mathrm{d}\mathbf{X} = \mathrm{d}_{\mathbf{p}} \sum_j w_j(\mathbf{p}^*) \mathbf{g}(\mathbf{X}_j(\mathbf{p}^*)) \tag{15}$$

$$= \sum_j \left(\mathrm{d}_{\mathbf{p}} w_j(\mathbf{p}^*)\, \mathbf{g}(\mathbf{X}_j) + w_j(\mathbf{p}^*)\, \partial_{\mathbf{X}} \mathbf{g}(\mathbf{X}_j)\, \partial_{\mathbf{p}} \mathbf{X}_j(\mathbf{p}^*) \right). \tag{16}$$

The problematic term in this last sum is $\partial_{\mathbf{X}} \mathbf{g}$, the spatial derivatives of the finite-element force density. This quantity is not required for standard finite-element computations, and is not part of most commercial FE packages. Thus, custom code needs to be generated for every material model and choice of element. Especially for non-linear material models, the complexity of this code considerably adds to the runtime of computing shape derivatives.

We show that the sum over the second term in Eq. 16 exactly cancels a subexpression of the sum over the first term. To this end, we split $\mathrm{d}_{\mathbf{p}} w_j$ into the direct contribution from changing the shape parameters, and thus the integration domain, and the contribution from changing the bounding box in which the quadrature points are placed. The parameters of the bounding box are represented by $\mathbf{t}(\mathbf{p})$, and the total derivative of $w_j(\mathbf{p}, \mathbf{t}(\mathbf{p}))$ wrt $\mathbf{p}$ is given by

$$\mathrm{d}_{\mathbf{p}} w_j = \partial_{\mathbf{p}} w_j + \partial_{\mathbf{t}} w_j\, \partial_{\mathbf{p}} \mathbf{t}. \tag{17}$$

For the quadrature point locations, we have $\partial_{\mathbf{p}} \mathbf{X}_j = \partial_{\mathbf{t}} \mathbf{X}_j\, \partial_{\mathbf{p}} \mathbf{t}$, because $\mathbf{X}_j$ depends on $\mathbf{p}$ only through $\mathbf{t}$.

Let $\mathbf{t}^*$ denote the bounding box parameters for the current shape parameter values $\mathbf{p}^*$. To show the claim, we define modified weights $\tilde{w}_j(\mathbf{p}) := w_j(\mathbf{p}, \mathbf{t}^*)$ that retain constant bounding box parameters regardless of $\mathbf{p}$. It follows from the definition that $\tilde{w}_j(\mathbf{p}^*)$ equals $w_j(\mathbf{p}^*, \mathbf{t}(\mathbf{p}^*))$. However, their shape derivatives are generally not the same at $\mathbf{p}^*$ because $\tilde{w}_j$ lacks the dependence on $\mathbf{t}$. Similarly, we define $\tilde{\mathbf{X}}_j := \mathbf{X}_j(\mathbf{p}^*)$. The crucial point is that $(\tilde{\mathbf{X}}_j, \tilde{w}_j(\mathbf{p}))$ define a valid quadrature rule not only at $\mathbf{p}^*$, but also in a neighborhood around $\mathbf{p}^*$. This is because the correctness of a quadrature rule does not depend on a specific choice of $\mathbf{t}$, and thus keeping $\mathbf{t} = \mathbf{t}^*$ constant while $\mathbf{p}$ varies is valid.

This argument lets us replace the quadrature rule based on w_j and $\mathbf{X}_j$ in Eq. 15 with the modified rule. This leads to

$$\mathrm{d}_{\mathbf{p}} \mathbf{f}_e = \mathrm{d}_{\mathbf{p}} \int_\Omega \mathbf{g}(\mathbf{X})\, \mathrm{d}\mathbf{X} = \mathrm{d}_{\mathbf{p}} \sum_j \tilde{w}_j(\mathbf{p}^*) \mathbf{g}(\tilde{\mathbf{X}}_j)$$

$$= \sum_j \partial_{\mathbf{p}} w_j(\mathbf{p}^*)\, \mathbf{g}(\mathbf{X}_j(\mathbf{p}^*)). \tag{18}$$

Comparing Eq. 18 to Eq. 16, we see that changing $\mathrm{d}_{\mathbf{p}} w_j$ to $\partial_{\mathbf{p}} w_j$ lets us effectively drop the second term containing the spatial derivatives of $\mathbf{g}$.

The consequence is that, even though we adapt the quadrature point locations to the integration domain after every parameter change, we may compute shape derivatives *as though* we kept them constant, without introducing error. This argument applies to any quantity that is integrated over the domain of an element, or over any area or surface for which quadrature rules have been constructed. It is thus not necessary to ever compute spatial derivatives of these quantities in order to find their shape derivatives.

3 VERTEX & ELEMENT ENRICHMENT

This section serves as a guide to implementing the extended finite-element method as outlined in Sec. 5.2 of the main article. First, we define the inputs to the enrichment procedure and some terminology. The simulation grid is a regular 3-dimensional grid which consists of *vertices*, edges, faces, and *cells*. Every cell is incident to eight vertices. The CAD *model*, which serves as the simulation domain, is embedded in the simulation grid. If the intersection between the model and a cell is non-empty, we refer to the connected components of this intersection as *subvolumes*.

A vertex which is incident to a cell intersecting the model is called a *node*. Every node has one or more sets of degrees of freedom (*dof*). A set of *dof* refers to the primary variable in the PDE that will be solved, e.g., a coordinate triple of displacements for an elastostatic problem. Every subvolume defines an *element*, which is associated with eight sets of *dof*. These sets of *dof* need to be appropriately

chosen from the eight nodes incident to the cell containing that subvolume.

The output of the enrichment stage is a list of *dof*, each associated with a node location, and a list of elements, each associated with eight sets of *dof*. This is exactly the structure of a finite-element mesh with linear hexahedral elements, and any finite-element code for meshes of this type will work with our enriched mesh. Here, the word "enrichment" is a slight abuse of language, since we effectively duplicate nodes instead of enriching them, and distribute distinct node copies to elements that are topologically separated. The details of this procedure are illustrated in pseudocode below.

Node Definition. Algorithm 1 shows how a list of (un-enriched) nodes is generated from the simulation mesh and subvolumes. It is assumed that index conversions are available, e.g., that an array `svToCell` gives the index of the cell in which a subvolume is contained, and that `cellToVertices` gives the indices of all vertices incident to a cell.

ALGORITHM 1: Node Definition

```
Data: svToCell, cellToVertices, numSV
Output: a list of all vertices that are nodes
nodes ← ∅
for i ← 1 to numSV do
    cell ← svToCell[i]
    foreach j ∈ cellToVertices[cell] do
        nodes ← nodes ∪{j}
    end
end
return nodes
```

Vertex Enrichment. The goal of this stage is twofold: first, to determine the number of sets of *dof* at a node, and second, to group subvolumes in the neighborhood of the node if they belong to the same set of *dof*. Algorithm 2 takes a node as input and analyzes the subvolumes within the cells incident to this node. These subvolumes are separated into connected components, where two subvolumes are said to be connected if they are path-connected *within* the cells incident to the node. The number of connected components gives the number of sets of *dof* at the node. The algorithm uses `nodeToCells`, which gives all cells incident to a node, `cellToSv`, which gives all subvolumes contained in a cell, and `adjacentSv`, which gives all subvolumes adjacent to a given subvolume.

Element Enrichment. At this stage, we generate one element for every subvolume, and assign *dof* indices to it. This is again done in a node-centric way, using the connected-component information generated in the previous step. We assume that the sets of *dof* for a node i are indexed contiguously through $k, k+1, \ldots, k+m$. Every element has eight nodes associated with it, namely, the eight nodes incident to the cell that contains the element. The local index of a node within an element, i.e., an integer between 0 and 7, is assumed to be known via the function `localIndexOfNodeInElement`.

Algorithm 3 assigns the sets of *dof* of one particular node to all elements in its neighborhood. Once this algorithm has been executed for all node indices, every element will have received all

ALGORITHM 2: Vertex Enrichment

```
Data: nodeToCells, cellToSv, adjacentSv
Input : the index of a node i
Output: the connected components of subvolumes in the
        neighborhood of node i
cells ← nodeToCells[i]
sv ← ∅
foreach c ∈ cells do
    sv ← sv ∪ cellToSv[c]
end
components ← ∅
compIndex ← 0
while sv ≠ ∅ do
    c ← sv.anyElement()
    sv ← sv \ {c}
    stack ← {c}
    comp ← ∅
    while stack ≠ ∅ do
        s ← stack.top()
        stack.pop()
        comp ← comp ∪ {s}
        foreach n ∈ adjacentSv[s] do
            if n ∈ sv then
                sv ← sv \ {n}
                stack.push(n)
            end
        end
    end
    components[compIndex] ← comp
    compIndex ← compIndex + 1
end
return components
```

eight of its *dof* indices. The result is a two-dimensional n-by-8 array `elementDof`, which contains in the i-th row the *dof* indices of the i-th element. Together with a list of all node locations, with duplicates indicating multiple sets of *dof* per node, this is the standard format of a linear hexahedral finite element mesh, and can be used as such. The only modification to a standard finite-element code is that quadrature rules need to be generated for every element instead of using tabulated rules.

ALGORITHM 3: Element Enrichment

```
Data: elementDof, components
Input: the index of a node i, the index k of the first set of dof
       belonging to node i
for j ← 0 to components[i].size() − 1 do
    comp ← components[i]
    foreach el ∈ comp do
        v ← localIndexOfNodeInElement (el,i)
        elementDof[el][v] ← k + j
    end
end
```

REFERENCES

K Atkinson and Ezio Venturino. 1993. Numerical evaluation of line integrals. *SIAM journal on numerical analysis* 30, 3 (1993), 882–888.

B. Müller, S. Krämer-Eis, F. Kummer, and M. Oberlack. 2017. A high-order discontinuous Galerkin method for compressible flows with immersed boundaries. *Internat. J. Numer. Methods Engrg.* 110, 1 (2017), 3–30.

B Müller, F Kummer, and M Oberlack. 2013. Highly accurate surface and volume integration on implicit domains by means of moment-fitting. *Internat. J. Numer. Methods Engrg.* 96, 8 (2013), 512–528.