Efficient implementation of Galerkin meshfree methods for large-scale problems with an emphasis on maximum entropy approximants

Christian Peco, Daniel Millán, Adrian Rosolen and Marino Arroyo*

LaCàN, Universitat Politècnica de Catalunya (UPC), Barcelona 08034, Spain

Abstract

In Galerkin meshfree methods, because of a denser and unstructured connectivity, the creation and assembly of sparse matrices is expensive. Additionally, the cost of computing basis functions can be significant in problems requiring repetitive evaluations. We show that it is possible to overcome these two bottlenecks resorting to simple and effective algorithms. First, we create and fill the matrix by coarse-graining the connectivity between quadrature points and nodes. Second, we store only partial information about the basis functions, striking a balance between storage and computation. We show the performance of these strategies in relevant problems.

Keywords: meshfree methods, local maximum entropy, sparse matrix efficient assembly, matrix structure creation, optimal memory storage, code optimization

1. Introduction

Meshfree methods have emerged in recent years as a viable alternative to finite elements in a number of applications, see [1, 2, 3, 4, 5] for a detailed review. These methods are based on basis functions that do not rely on a mesh. As a consequence, many of the requirements associated with the quality of the elements in traditional finite element method (FEM) are relaxed or disappear, but this extra flexibility raises new challenges in the numerical implementation [6]. Meshfree methods also present several advantages such as basis functions with high-order continuity, robustness in dramatic grid deformations [7, 8, 9], and easier local adaptivity [10, 11]. Galerkin meshfree methods require a quadrature mesh to perform numerical integration, commonly requiring a higher number of quadrature points to accurately integrate the weak form due to their nonpolynomial nature and nonelement-wise support [12, 13]. Additionally, most of the meshfree methods present an awkward treatment of essential boundary conditions due to nonsatisfaction of the Kronecker delta property [3, 14].

*Correspondence to: marino.arroyo@upc.edu

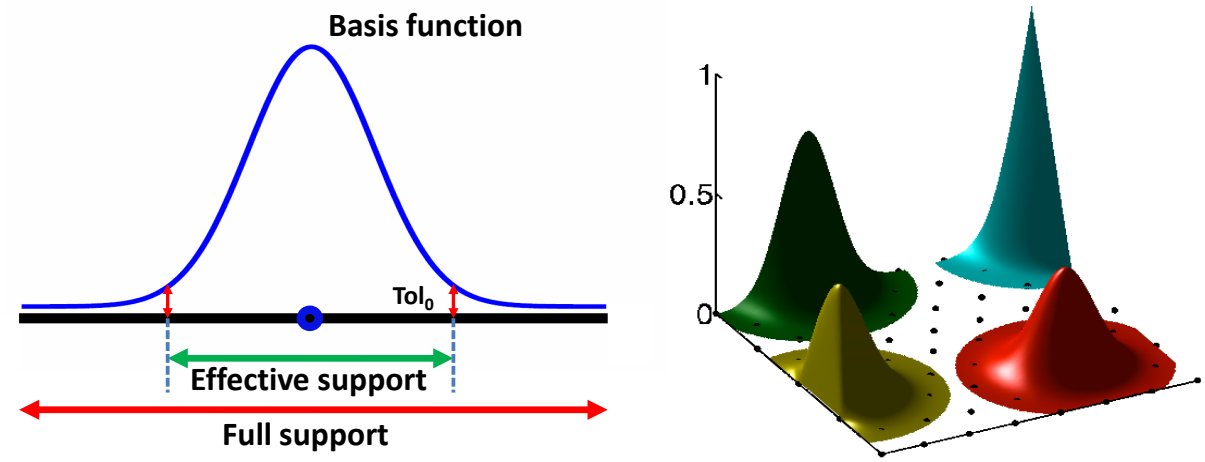

Figure 1: Full support of some meshfree basis functions, such as local maximum entropy approximants, covers the convex hull of the computational domain. The effective numerical support radius r_a is determined by a cutoff basis function value $\texttt{Tol}_0$ (left). Representation of two-dimensional LME approximants basis functions (right). Notice the noninterpolant character and the smoothness of the basis functions, and the fulfillment of a weak Kronecker-delta at the boundary of the convex hull.

Since smoothed particle hydrodynamics [15], a variety of techniques have emerged, such as reproducing kernel particle method [16], partition of unity finite element method [17], element free Galerkin [18] and the method of finite spheres [19, 20], to mention a few. We resort in this work to the local maximum entropy (LME) approximation schemes, a meshfree method inspired on information theory that generates nonnegative and smooth basis functions (see [21, 22, 23, 24, 25] for a detailed description, properties and extensions). Because the LME basis functions do not satisfy the Kronecker-delta property at nodes, these schemes are referred to as approximants instead of interpolants. The capabilities of LME approximants have been examined in a variety of computational mechanics applications, such as linear and nonlinear elasticity [25, 26], plate [27] and thin-shell analysis [28, 29], convection-diffusion problems [30, 31], and phase-field models of biomembranes [32, 33] and fracture mechanics [34, 35, 36].

Like other meshfree methods, LME approximants involve a dilation or locality parameter that modulates their behavior and support. LME approximants show an exponential decay controlled by the locality parameter, and far from the boundaries they look like Gaussian weighted functions [37, 24]. Their effective support is controlled by setting a cut off or threshold value ($\texttt{Tol}_0$) below which the basis functions are taken numerically to be zero (see Fig. 1, Appendix A). The proper choice of the locality parameter is problem dependent and not easy in general, which has motivated a systematic studies for general meshfree methods [38] and for LME approximants [25] in particular. In LME approximations, the locality parameter is an aspect ratio

parameter γ, which allows us to smoothly move from linear finite elements shape functions ($\gamma > 4.0$) to more spread out approximation schemes (e.g., $\gamma = 0.6$), as illustrated in Fig. 2. In general, broader functions lead to more accurate results for problems with smooth solutions at the expense of higher computational cost and worse matrix conditioning [22, 28].

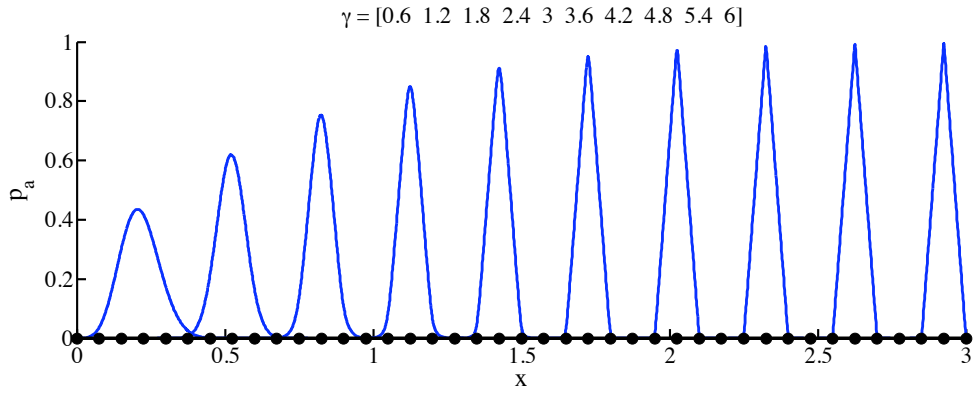

Figure 2: Seamless transition from spread-out meshfree to linear finite elements basis functions.

In contrast to conventional FEM, where the structure of the matrix is inherited from the mesh graph, stencils of meshfree schemes depend strongly on the aspect ratio parameter γ. In our experience, a noticeable run-time computational cost of meshfree methods is due to the creation of the sparse matrix structure and the assembly process, which can be specially harmful for iterative processes. These stages can be as expensive as the solver stage in two-dimensional problems and exceed it in three-dimensional ones. In a typical implementation of the assembly process in meshfree methods, the code loops over the quadrature points. The denser sparsity pattern and the large number of Gauss points required for accurate integration can make these methods unpractical for large-scale calculations. To overcome this issue, we propose here a set of algorithms based on a loop over cells/elements, as commonly done in FEM. We illustrate in this work how this simple approach reduces significantly the computational cost associated with the matrix structure creation and the assembly process.

Additionally, a widespread practice (both in FEM and in meshfree methods) is to store in memory the basis functions and their derivatives for repetitive calculations required in nonlinear iterative solvers, incremental loading, or evolution in time. In FEM, this storage is insignificant because the basis functions of the parent element are mapped to each physical element. Since this is not the case in meshfree methods, the amount of memory and its access can become a bottleneck and substantially reduce the code efficiency, especially in large-scale problems. If meshfree basis functions are not stored in memory but recomputed every time, the computational cost can also increase significantly. To alleviate this issue, we propose here a strategy that is a trade-off solution between memory storage and computational time. The technique, based on a data structure that stores only partial information about the basis functions and an algorithm to reconstruct them when needed, reduces considerably the memory usage at the ex-

pense of a minimum increment in the overall computational cost. We illustrate and exploit this concept on LME approximants.

The paper is organized as follows. In Section 2 we review the basic technicalities for a meshfree method particularized to LME approximants and the classical implementation to approximate partial differential equations (PDEs). We then propose an algorithm to speed-up the matrix assembly and an algorithm for the compressed memory storage of LME approximants in Section 3. We extensively test our proposals with numerical examples in Section 4 and finish with some concluding remarks in Section 5.

2. A standard meshfree scheme

Let $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\} \subset \mathbb{R}^d$, for $d = 1, 2, 3$, be an unstructured set of nodes used to describe a domain Ω, and $p_a(\boldsymbol{x})$ the meshfree basis function associated to the a-th node, for $a = 1, \ldots, N$. A continuous field Φ can be approximated as

$$\Phi(\boldsymbol{x}) = \sum_{a=1}^{N} p_a(\boldsymbol{x})\Phi_a,$$

where Φ_a stand for the nodal coefficients. Here we adopt the LME approximants as meshfree basis functions in a Galerkin method to approximate a general PDE. They are nonnegative, smooth, satisfy at least up to the first order consistency conditions and present a weak Kronecker-delta property. We rely on a background mesh to define the quadrature points, typically through a Gauss-Legendre quadrature rule. Since this mesh is just required to place the Gauss points, its requirements are less strict than in a mesh-based method. We use here meshes made of triangles/tetrahedra in 2D/3D, which we easily obtain via the library QHULL [39]. The procedure needed to compute the system matrix in a Galerkin meshfree approach requires mainly four steps: (i) neighborhood search, (ii) computation of the basis functions, (iii) creation of the sparse matrix structure and (iv) Gauss point-wise matrix filling. The pseudocode shown in Algorithm 1 summarizes these four steps. In the present work, we do not deal with solver performance. In the following we briefly extend on the computational implications of every step.

Algorithm 1 Pseudo-code for scheme based on a loop over quadrature points (see Section 2).

(i) Determine the neighborhood nodal index set $\mathcal{N}_y^X$ for each Gauss point.
(ii) Compute shape functions (array p_a).
(iii) Construct sparse matrix structure (arrays *ia* and *ja*).
(iv) Fill sparse matrix (array *an*) with the quadrature point loop based algorithm.

The objective of step (i) is to compute the so-called neighbor lists, which can be interpreted as the counterpart of the mesh connectivity in FEM where the neighbor lists are given by the

mesh itself. In a meshfree scheme this is made by specialized algorithms, i.e. neighbor searchers which identify the relationship between the quadrature points and the nodes. We will refer to the neighbor lists as *primal* and *dual lists* [28], which are complementary. In particular, a dual list identifies the quadrature points that are influenced by a particular node i.e. the quadrature points falling within the effective support of a nodal basis function. Conversely, the primal list contains the nodes that influence a particular quadrature point.

Formally, let $Y = \{\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_L\} \subset \Omega$ be a set of quadrature points. The dual list containing the nearest points from Y associated with a node $\boldsymbol{x}_a \in X$ can be defined as follows

$$\widetilde{\mathcal{N}}^Y_{\boldsymbol{x}_a} = \{k \in \{1, 2, .., L\} \mid |\boldsymbol{y}_k - \boldsymbol{x}_a| < r_a\},$$

where r_a is the effective support radius of the shape function associated to the a-th node (which is determined by a user defined cutoff value $\texttt{Tol}_0$, see Fig. 1). In the same way, the primal list containing the nodes from X associated with a particular quadrature point $\boldsymbol{y}_k \in Y$ is defined as

$$\mathcal{N}^X_{\boldsymbol{y}_k} = \{a \in \{1, 2, .., N\} \mid |\boldsymbol{y}_k - \boldsymbol{x}_a| < r_a\}.$$

The primal and dual lists are used afterwards in steps (ii), (iii) and (iv) to compute the basis functions, identify the nonzero positions in the sparse matrix and perform the assembly. Both lists can be obtained by simply invoking a neighbor searcher. The neighbor finding problem is standard, and comes in two flavors, namely finding the k-first neighbors or finding neighbors within a range. In our codes, we resort to the approximate nearest neighbor searching library [40], whose computational cost scales as $O(N \log N)$, where N is the number of nodes.

In step (iii) the nonzero elements in the global matrix are identified using algorithms that postprocess the neighbor lists. This information is critical to properly store the matrix in a sparse scheme and perform the filling. There are many methods for storing sparse matrices (see, for instance, [41] and [42]). We follow here the compressed sparse row (CSR) storage, which is a proper choice for codes written in C/C++ due to its memory layout. In CSR, a matrix is given in terms of three lists. The first list, *ia*, is an array of integers that stores the total number of nonzeros up to each row. Its dimension is the number of rows plus one, the first position being filled with a zero. The second and third lists are arrays of integers and doubles, *ja* and *an*, have as dimension the number of nonzeros in the matrix, and store the column index position and the associated matrix entry. We understand the sparse matrix structure creation as the collection of algorithms required to obtain the lists *ia* and *ja*. An efficient way to compute this structure based on a loop over quadrature points is presented in Appendix C. The standard algorithm loops over the primal lists of the Gauss points associated to each nodal dual list, such that the nonzero entries of the sparse matrix are identified when two nodes appear together in at least one primal list. As a result, the sparse structure construction becomes increasingly expensive as the number of quadrature points and the support of the basis functions becomes larger.

In step (iv) the nonzero positions of the system sparse matrix (*an*) are filled in an operation dependent on the PDE. Pursuing a rational memory access, standard Galerkin meshfree algorithms rely on a loop over the Gauss points, each contributing with a local dense matrix. The

number of rows of this local matrix is equal to the cardinality of the primal list $|\mathcal{N}_y^X| = n$ times the number of scalar fields in the problem. Again, the computing time of this step is directly penalized by the increase of quadrature points and by the support size of the basis functions. Furthermore, the local matrices have to be assembled into the global matrix. Since the global matrix is sparse, a search is required to identify the global position to be filled. This concept is illustrated in the upper part of Fig. 3.

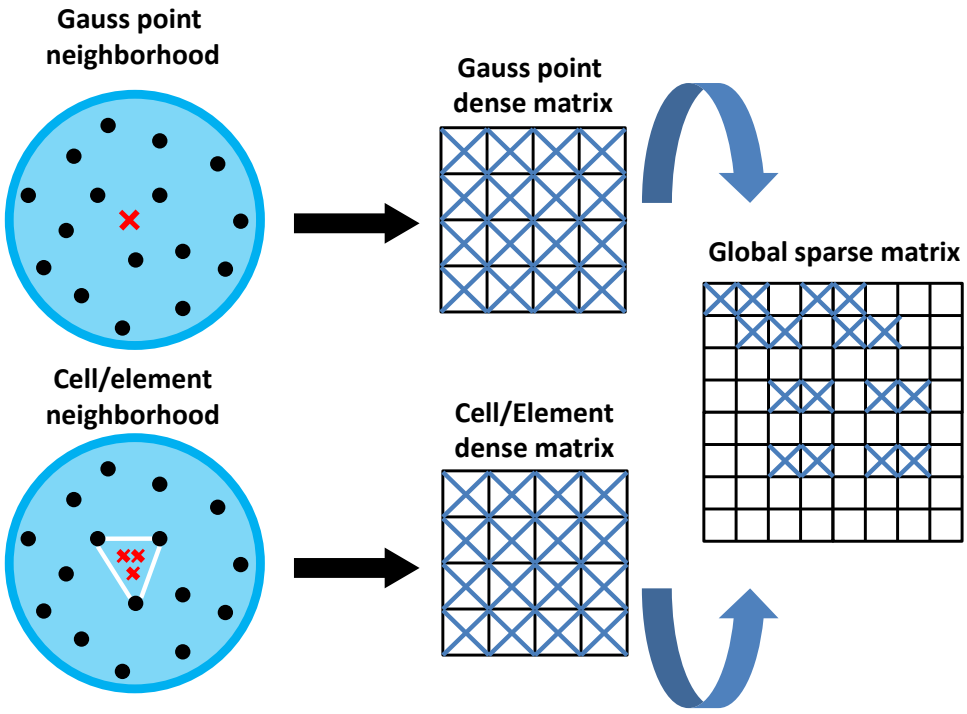

Figure 3: Filling algorithm from neighbor lists to global sparse matrix. Nodal list of neighbors (black dots) can be computed for an individual integration point (red X, top) or for a cell/element (white triangle, bottom). Dense submatrices are generated from these neighborhoods and assembled into the global matrix. Cell/element algorithm improves memory management since the resulting dense submatrix condenses information coming from several integration points.

3. Meshfree optimization concepts

We present here two optimizations to improve the efficiency when facing the bottlenecks described in Section 1. First we describe in Section 3.1 a neighborhood coarsening algorithm that considerably speed-up steps (iii) and (iv). Finally, a reduced storage strategy that mitigates the memory requirements when storing the basis functions is detailed in Section 3.2.

3.1. Neighborhood coarsening algorithm

A simple idea to alleviate the computational cost of the global sparse matrix is to coarse-grain the neighbor primal lists. The key point is to generate a list for each cell/element of a

defined coarsening mesh rather than one per Gauss point. The coarsening mesh provides us with a structure to group the primal lists of the Gauss points contained in the cell/element. Without loss of generality, a straightforward and natural choice for the coarsening mesh is the quadrature mesh cells/elements needed in most Galerkin meshfree methods to perform the numerical integration. In this way the complexity added by the increase of Gauss points due to accuracy requirements is removed and the neighbor lists are generated disregarding the number of integration points. We present next details of this procedure.

Once the coarsening mesh is set, we start with a neighbor search over the nodes defining the mesh. This allows us to obtain nodal-based primal lists rather than primal lists for quadrature points. To obtain the cell/element primal lists, the primal lists of its associated nodes are simply merged. More specifically, we define

$$\mathcal{N}_{el} = \bigcup_{a \in \mathcal{T}_{el}} \mathcal{N}^X_{\boldsymbol{x}_a},$$

where $\mathcal{T}_{el}$ is an index set containing the nodal indexes of the el-th cell/element (e.g the mesh connectivity). Note that the $\mathcal{N}_{el}$ list is applicable to all of the integration points inside the cell/element, regardless their number. This merging operation is negligible in terms of computational time, and give us the possibility to work from now on with cell/element primal lists rather than with integration point based lists. We illustrate this concept in Fig. 4. The examples shown in Section 4 use the quadrature mesh as coarsening mesh and follow the proposed unifying criterium.

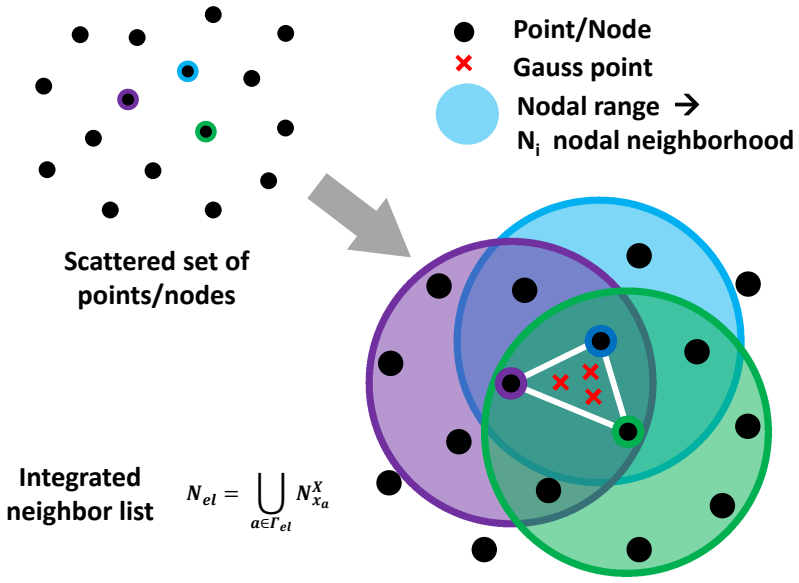

Figure 4: Integrated neighborhood concept. The new cell/element neighborhood is described by the union of nodal vertices lists of neighbors. The triangular elements given by the quadrature mesh are used here as background cell/element generator.

Vertex merging is a proper strategy when the support size r_a is large relative to the mesh size h_a, which is the usually the case in meshfree methods. Conceivably, it could be the case that the merging of the vertices lists would lead to some loss of information. A node could be influencing an integration point inside a cell/element without influencing any of the vertices containing it, e.g. in highly distorted triangles in 2D. If these unlikely events need to be absolutely ruled out, it is always possible to construct the unified lists by merging the neighbor lists of the Gauss points belonging to a cell/element. In our experience, however, this never happens when using a quadrature mesh; the agreement in numerical integration benchmarks is perfect, and for this reason we recommend the proposed vertex merging to create the cell/element lists.

The creation and filling algorithms can be now based on cell/element neighbor lists, which greatly speeds-up the computations. The structure creation is simplified since only the nodes and cells/elements are involved in the whole procedure. Now the nonzero positions are identified by looping over cell/element neighbor lists instead of looping over Gauss points neighbor lists. The filling of the matrix benefits in two distinct ways. Firstly, the element-wise approach leads to cell/element dense matrices. These local matrices are efficiently filled since just a loop over the neighbor list of the cell/element and a loop over the cell/element Gauss points are required. Secondly, only one dense cell/element matrix is assembled into the global matrix, hence the memory access is improved, as illustrated in the lower part of Fig. 3.

As we show later in Section 4.1, this optimization maintains constant the computational time associated with the matrix-pattern creation algorithm regardless the number of integration points used. This fact significantly alleviates one of the main disadvantages of meshfree methods, namely the large number of quadrature points needed as compared to piecewise polynomial approximants. Furthermore, the granularity of the element-level approach is better suited for parallel computing, minimizing memory access and limiting data exchange. The pseudo-code for the procedure is summarized in Algorithm 2.

Algorithm 2 Pseudo-code for scheme based on a loop over cells/elements (see Section 3).

(i) Compute adjacency lists for nodes $\mathcal{N}^X_{\boldsymbol{x}_a}$ and process cell/element lists $\mathcal{N}_{el} = \bigcup_{a\in\mathcal{T}_{el}} \mathcal{N}^X_{\boldsymbol{x}_a}$.

(ii) Compute shape functions (array p_a).

(iii) Construct sparse matrix structure (arrays *ia* and *ja*).

(iv) Fill sparse matrix (array *an*) with the cell/element loop based algorithm.

3.2. Compressed meshfree basis functions storage

A standard practice in the numerical treatment of PDEs is to store in memory the basis functions and their derivatives at each Gauss point. This strategy decreases considerably the computational cost in problems involving nonlinear iterative solvers or evolution problems on Lagrangian meshes. While this storage is insignificant in FEM, in meshfree methods the amount of memory (as quantified later) and its access can become a bottleneck and substantially reduce

the code efficiency. Here we propose a storage concept that is based on finding structures that optimally synthesize the basis function information at each integration point, striking a trade-off between memory storage and computation time. We generate data structures that store only partial information about the basis functions and an algorithm to reconstruct them when needed, reducing considerably the memory usage at the expense of a marginal increment in the overall computational cost.

For a general meshfree method, and considering the Galerkin approximation of a fourth-order PDE, the *full storage* of the basis functions requires $M_{FS} = L \cdot \bar{n} \cdot [1 + d + d(d+1)/2] = L \cdot \bar{n} \cdot \left(1 + \frac{3}{2}d + \frac{1}{2}d^2\right)$ doubles. In this equation, L is the total number of quadrature points, $\bar{n}$ is the mean cardinality of the primal lists, 1 accounts for the basis functions themselves, d for their gradients, and $d(d+1)/2$ for the Hessian, which is a symmetric matrix. In a for fourth-order PDE we typically have $\bar{n} \approx 65$ in 2D and $\bar{n} \approx 380$ in 3D. As a result, the memory requirements rapidly become unaffordable.

Focusing on LME approximants, we recall that the basis functions are obtained by means of a nonlinear optimization problem at each evaluation point with d unknowns, where d is the spatial dimension. This optimization problem yields the Lagrange multiplier associated with first-order consistency conditions. Once the Lagrange multiplier is known, an explicit expression for the basis functions, its gradient and its Hessian is explicit (see Appendix A). Even if the nonlinear optimization problem is relatively easy to solve by Newton's method, it accounts for a significant part of the basis function evaluation time.

A straight-forward alternative to the *full storage* method would be to simply store the d reals in the Lagrange multiplier at each quadrature point. Analyzing in detail the structure of the explicit formulae for the basis functions and derivatives, it is easy to identify a set of matrices and vectors whose size is independent on $\bar{n}$, and some of which involve summations over $\bar{n}$. Thus, storing these arrays saves significant computation time at a limited memory cost. As detailed in Appendix B, this simple observation suggests the *optimal or compressed storage*, which only involves $M_{OS} = L \cdot \left(2 + \frac{7}{2}d + 2d^2 + \frac{1}{2}d^3\right) \approx L \cdot (2+d) \cdot \left(1 + \frac{3}{2}d + \frac{1}{2}d^2\right)$ doubles. As the mean cardinality is in general much greater than the spatial dimension, i.e. $\bar{n} \gg (2+d)$, from the ratio $M_{FS}/M_{OS} = \bar{n}/(2+d)$ it is clear that the memory usage decreases significantly when the compressed storage technique is used, as can be observed in Table 1.

In Section 4.2 we apply this strategy to a fourth-order PDE requiring the storage of the values, gradients and Hessians of the LME basis functions. We quantify the memory usage and computational time devoted to evaluate the basis functions for both the full and for the optimized storage implementations.

4. Numerical examples

The performance of the optimizations presented in Section 3 are studied here in two boundary-value problems. We compare the proposed implementation of Galerkin meshfree methods with the standard implementation in terms of storage and computational time. Since we only change the way operations are organized and information stored, the numerical solutions with both

Table 1: Quantification and comparison of memory usage between the methods of full and optimal or compressed storage of local maximum-entropy basis functions and their derivatives. Here, $\bar{n}$ is the mean cardinality of primal lists, L is the number of Gauss points and d is the spatial dimension.

	Full storage	Optimal storage
Total memory usage	$M_{FS} = L \cdot \bar{n} \cdot \left(1 + \frac{3}{2}d + \frac{1}{2}d^2\right)$ $\bar{n} \gg (2 + d)$	$M_{OS} = L \cdot \left(2 + \frac{7}{2}d + 2d^2 + \frac{1}{2}d^3\right)$ $M_{OS} \approx L \cdot (2 + d) \cdot \left(1 + \frac{3}{2}d + \frac{1}{2}d^2\right)$
Comparison	$M_{FS}/M_{OS} \approx \bar{n}/(2 + d) \gg 1$	

implementations are indistinguishable. We focus in the four steps presented in Section 2 and leave aside the solver stage. As we specify in Section 1, in our experience the analized steps can be comparable in computational time to the solver in 2D problems and exceed it in 3D. In the first example the neighbor coarsening procedure from Section 3.1 is applied to a 2D heat diffusion and a 3D linear elasticity problems, whereas the compressed basis functions storage detailed in Section 3.2 is exercised in a nonlinear fourth-order phase-field PDE. Both problems use uniform grids that ensure a quite constant number of nodal neighbors for every integration point and facilitate the comparison. In the first example we use the quadrature mesh and the vertex merging approach to generate the unified primal lists, as proposed in Section 2. These stages can be as expensive as the solver stage in two-dimensional problems and exceed it in three-dimensional ones, particularly in problems with vectorial fields.

4.1. The neighborhood coarsening algorithm in 2D and 3D problems

We exercise first the neighborhood coarsening algorithm on a benchmark heat equation in 2D, which is a scalar problem. The sparse matrix structure creation and assembly using the proposed neighborhood coarsening for scalar and vectorial problems is detailed in Appendix C, along with a description of the data structures and a C/C++ pseudo-code.

The diffusion boundary problem is defined as follows:

$$\begin{aligned} \Delta T &= f, \quad \text{in } \Omega, \\ T &= T_0, \quad \text{on } \Gamma_D, \end{aligned}$$

where T is the temperature field, $\Omega = (0, 1) \times (0, 1)$ the domain, f is an arbitrary source of heat and T_0 is the prescribed temperature on the Dirichlet's boundary Γ_D. We consider $T_0 = 0$ on $\Gamma_D = \partial\Omega$ and assume a source $f = 2y$. The solution obtained using LME approximants with a uniform grid of points of 100×100 is depicted in Fig. 5.

The entries of the stiffness matrix take the standard form

$$K_{ab} = \int_\Omega \nabla p_a \cdot \nabla p_b \, d\Omega,$$

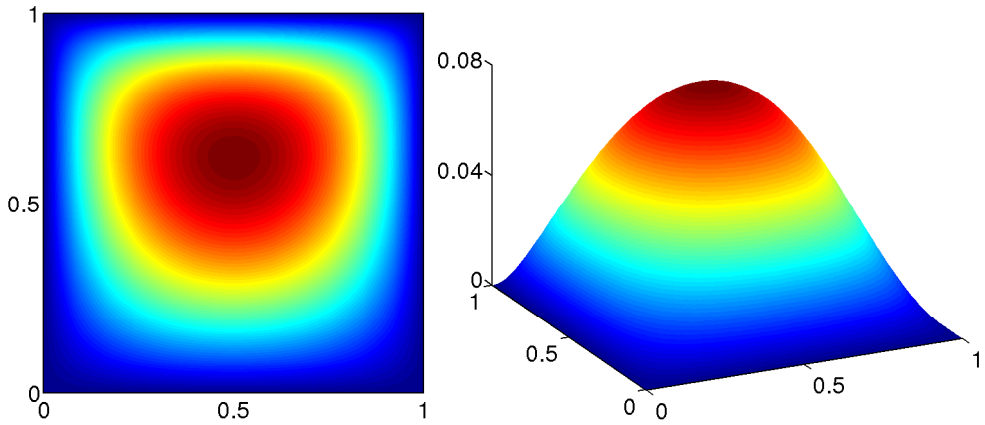

Figure 5: 2D and 3D views of the solution for a heat equation with a source. We use LME approximants, a uniform mesh of 100×100 nodes, $\gamma = 1.6$ and 6 Gauss points per triangular cell/element.

where Ω cannot be reduced to a set of elements as in FEM. A quadrature rule is defined on a background integration mesh over the whole domain (that may or may not coincide with the coarsening mesh) as

$$K_{ab} = \sum_{k=1}^{L} \nabla p_a(\mathbf{y}_k) \cdot \nabla p_b(\mathbf{y}_k)\, \omega_k,$$

where ω_k stand for the Gauss points quadrature weights in physical space.

The performance of the algorithm based on looping over quadrature points depends on the number of nodes used to discretize the domain, the number of Gauss points per quadrature cell and the size of the support of the basis functions (linked to the aspect ratio parameter γ). In Fig. 6 we show a representative performance, reporting the computational time spent in the main four stages of the algorithm described in Section 2, i.e. (i) neighborhood search, (ii) shape functions, (iii) matrix structure creation and (iv) matrix structure filling. The plots show computational time vs degrees of freedom (number of nodes) for different combinations of the aspect ratio parameter $\gamma = 0.8, 1.6, 4.0$ and three and twelve Gauss points per element. The left-upper chart shows FEM-like LME approximants ($\gamma = 4.0$) with three points per integration element. In this case, the bottleneck in the shape functions calculation. In the other plots we can see how increasing the support size (decreasing γ) or/and increasing Gauss points per element dramatically rises the cost of structure creation and structure filling in comparison with the FEM-like shape functions. The upper charts in Fig. 7 illustrate the computing time growth for different number of Gauss points, whereas the lower charts depict the growth when changing the parameter γ. The results of both figures highlight the need for speed-up techniques in steps (iii) and (iv) when the size of the system increase for spread-out basis functions ($\gamma = 0.8$, large support) that require accurate numerical integration.

We proceed then to analyze the proposed cell/element scheme and review the performance of critical stages (iii) and (iv). In the upper part of Fig. 8 the computational time vs number of

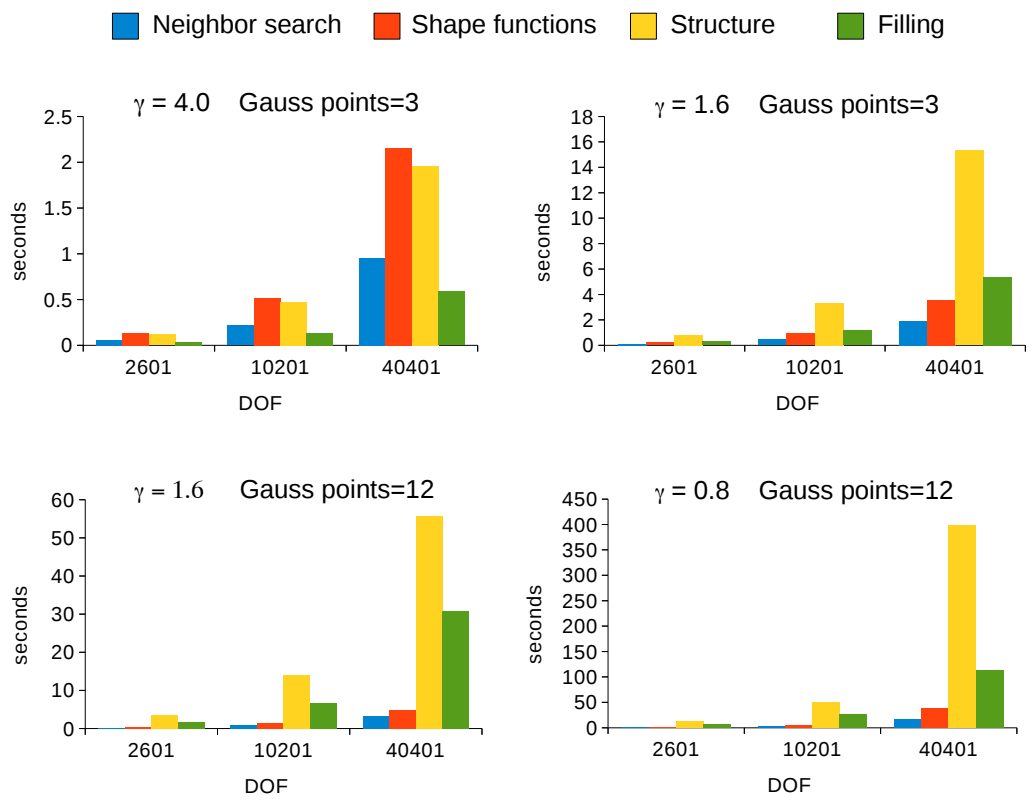

Figure 6: Computational time vs grid size for different values of Gauss points per cell and γ. From left to right, bars correspond to stages: (i) neighborhood search, (ii) shape functions, (iii) matrix structure creation and (iv) matrix structure filling.

Gauss points is shown for $\gamma = 0.8, 1.6, 4.0$. The juxtaposed bars in the figures compare the standard and the new implementations. We can see how the gain in performance grows as the support size increases, giving greater speed-ups as γ decreases e.g. ten times faster for twelve Gauss points and $\gamma = 0.8$. Notice also that the matrix structure creation is completely insensitive in the new implementation to the number of quadrature points per element, see Fig. 8. No speed-up is observed when a small number of integration points is used. We show in the lower panels of Fig. 8 the filling algorithm computational time vs number of Gauss points for $\gamma = 0.8, 1.6, 4.0$. We observe the same pattern of speed-ups when γ decreases. Notice that although the speed-up is considerable, the filling operations do depend on the number of quadrature points in the nee algorithm, but far less critically than in the standard implementation. Nevertheless, the filling time is greatly reduced with the proposed approach, particularly for large supports i.e. five times smaller for $\gamma = 0.8$ and twelve Gauss points.

Finally, the growth of computational time as a function of system size is presented in Fig. 9. The standard implementation using twelve Gauss points is shown as reference. We conclude that our proposal is significantly more efficient than the algorithm based on looping over quadrature points, and that the improvement increases as the number of quadrature points and support

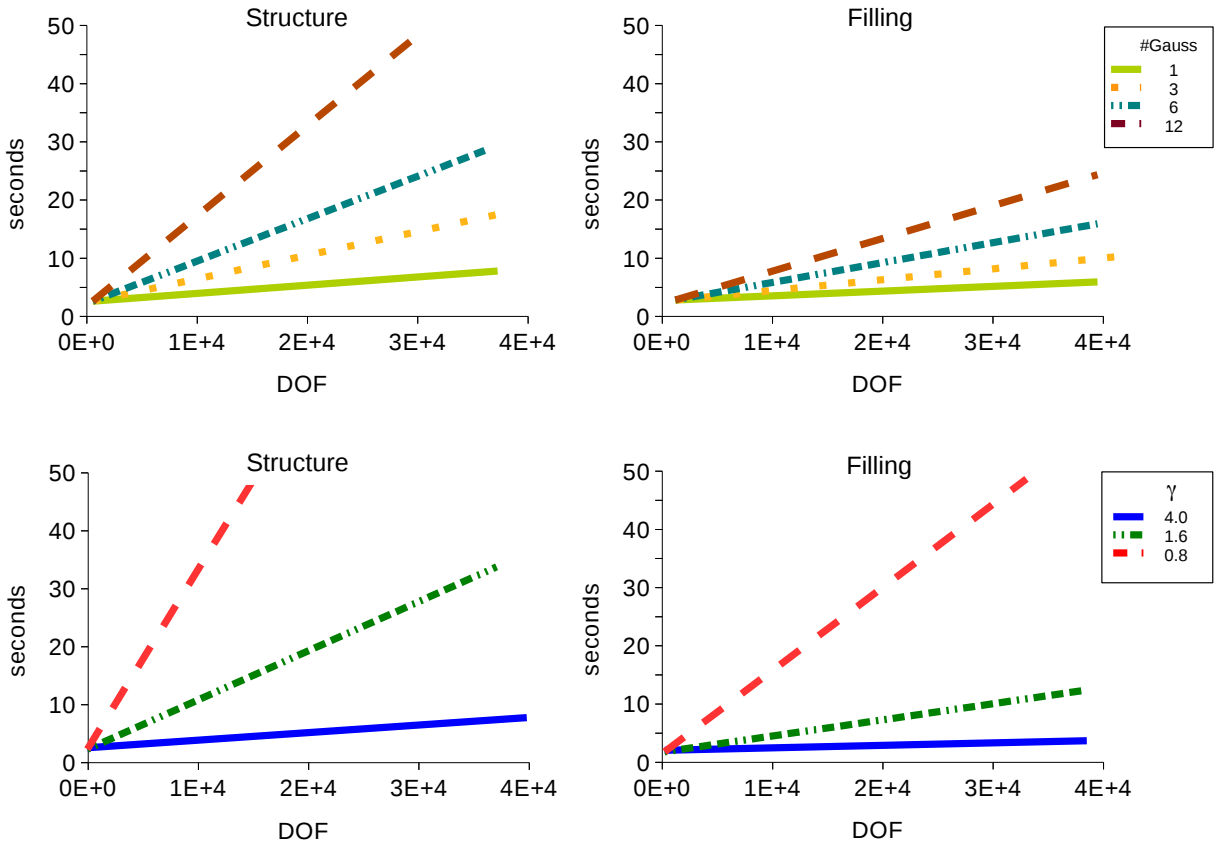

Figure 7: Comparison between growth for matrix structure creation (left) and filling algorithms (right) with the grid size for different values of Gauss points per cell (top, γ=1.6) and for different values of parameter γ (bottom, Gauss points = 6).

size become larger.

We test now the neighborhood coarsening algorithm in a 3D linear elasticity problem, where a cube of $l = 1.0$ m is stretched (we consider a Young's modulus of $E = 0.01$ GPa and Poisson's ratio $\nu = 0.2$). We present the results for a $20 \times 20 \times 20 = 8000$ node set with 4, 11 and 15 Gauss points per quadrature cell. We summarize in Fig. 10 the results. The time invested in the creation of the structure, which is omitted in the figure, is small (4 s) and remains constant regardless of the Gauss point number. We focus here in the filling time, which is dominant in 3D, and show that the proposed method is quite insensitive to the number of quadrature points, while the computational time for standard implementation rapidly increases with the number of quadrature points. The filling time for 15 quadrature points is 4 times longer with the standard implementation. Solver time for this problem is around 100 s.

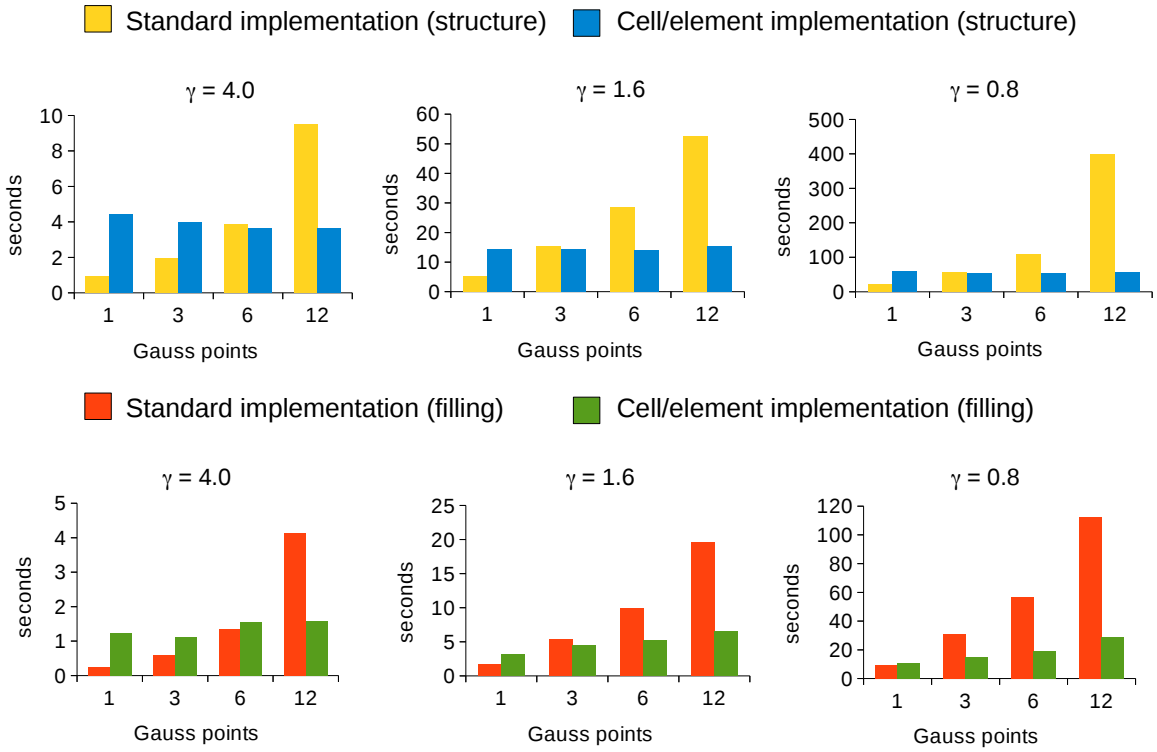

Figure 8: Structure creation (top) and filling (bottom) computational time vs Gauss points for decreasing γ (increasing support). Standard and new implementation are shown (left and right bars, respectively). DOF = 40,401.

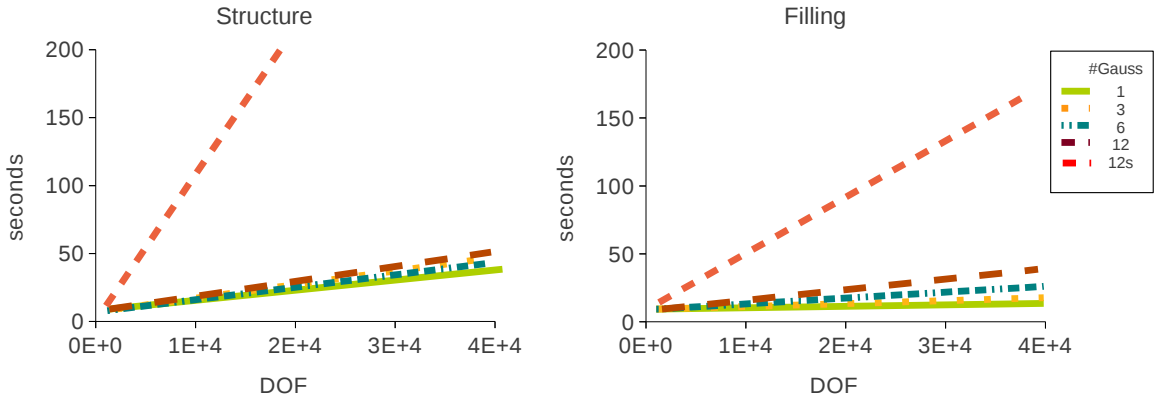

Figure 9: Growth of the computational time as a function of the size of the system for the matrix structure creation (left) and filling (right), using the new implementation different numbers of Gauss points per cell and $\gamma = 0.8$. For comparison purposes, the standard implementation using 12 Gauss points (legend 12s) is presented.

4.2. The compressed meshfree basis functions storage applied to a phase-field fracture model

We present here the results of the proposed memory storage strategy for the LME approximants. We compare the full storage and optimized schemes in a fourth-order PDE problem

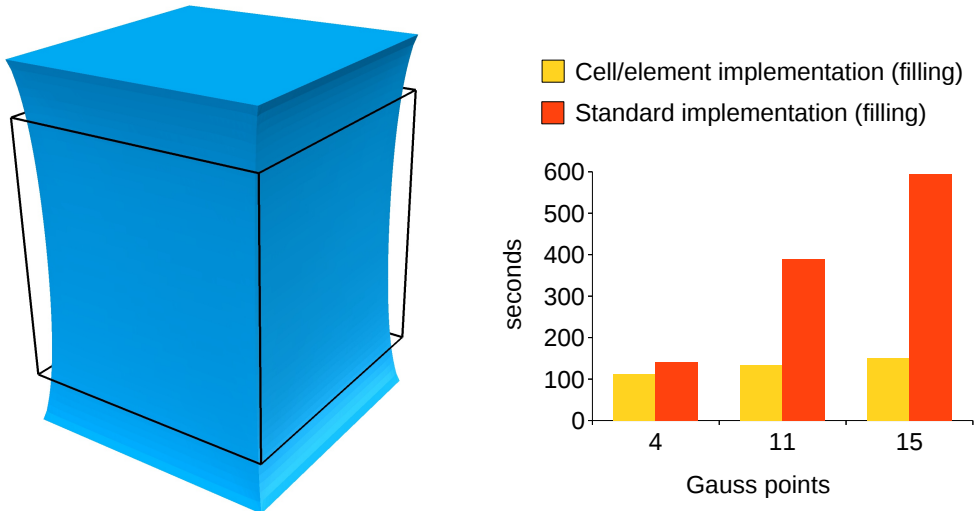

Figure 10: Deformed configuration for 3D linear elasticity benchmark (left). Computational time for the filling algorithm using the new and the standard implementation for different numbers of Gauss points per cell, $\gamma = 0.8$ and $24,000$ degrees of freedom.

requiring the values, gradients and Hessians of the basis functions. In a variational model of fracture, the phase-field PDE results from the following functional

$$\int_\Gamma G_c \, d\Gamma = \int_\Omega G_c \left[\frac{(1-\phi)^2}{4l_0} + \frac{l_0}{2}|\nabla\phi|^2 + \frac{l_0^3}{4}\Delta\phi^2 \right] d\Omega,$$

where G_c is the critical fracture energy density, ϕ the phase-field, and l_0 the parameter controlling the width of the approximation of the crack. We illustrate a typical solution in Fig. 11. More details about this particular model can be found in [43].

We focus on the amount of doubles that need to be stored when using a standard and the optimized scheme, and also on the impact on computational time of the structure filling routine for the global matrix. The latter requires retrieving the stored basis functions in the usual approach, and partially recomputing them in the optimized storage approach. The structure creation step is completely independent on evaluation/retrieval of the basis functions, and for this reason we do not report it here. As can be observed in the left panel of Fig. 12, the optimized storage strategy decreases the memory requirements by an order of magnitude; the ratio of memory requirements is about 20. For this two-dimensional problem we use $\gamma = 1$, leading to a mean value of 72 neighbors per integration point. Here we use six Gauss points by element.

We analyze now the computational time invested in the filling of the global matrix. We can observe in the right panel of Fig. 12 that the memory optimized storage is marginally slower than the standard routine. The extra operations to retrieve the basis functions and its derivatives, see Appendix A, is partially compensated by a more efficient access to the memory, resulting in running time increments of about 10%.

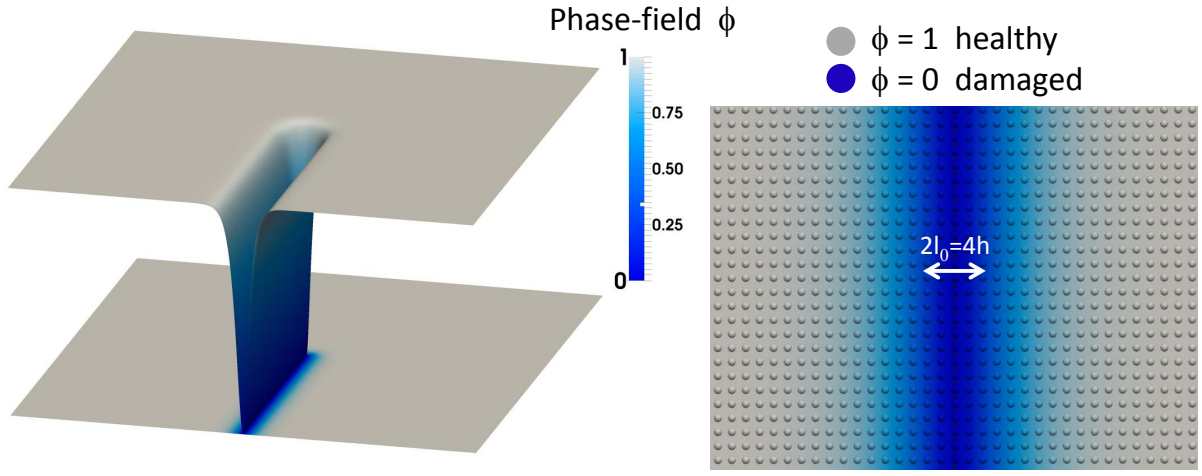

Figure 11: Fourth-order phase-field solution for a crack. Phase-field values range from 1 to 0 signaling the progressive damaging of the material (left). The ratio between the crack width parameter l_0 and the nodal spacing h is 2 (right).

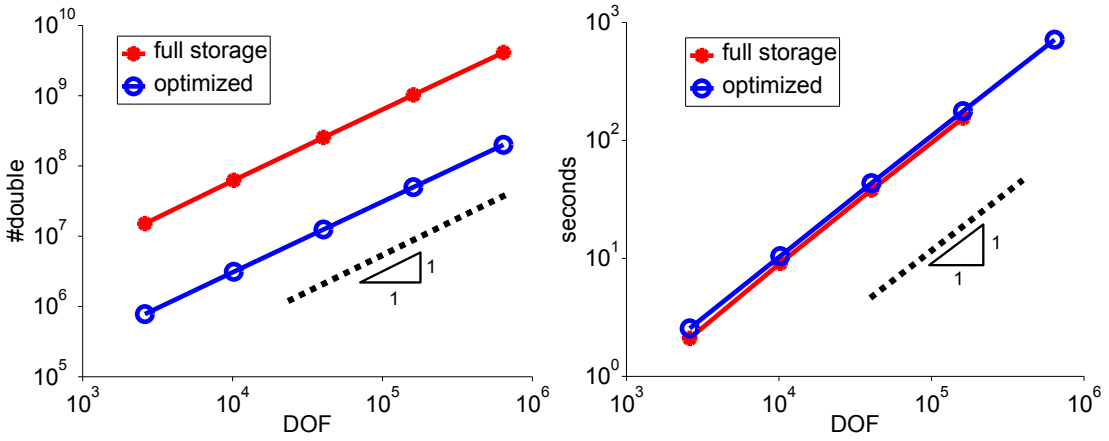

Figure 12: Comparison between the full storage and optimized implementations. The plots show the number of doubles stored in the global matrix vs the number of degrees of freedom (left) and the computational time invested in the filling and assembly of the global matrix vs the number of degrees of freedom (right).

5. Conclusions

We have presented two optimization procedures to mitigate two fundamental bottlenecks in Galerkin meshfree methods: matrix assembly and basis functions storage. We have shown how the sparse structure creation and filling of the system matrix become critical in a meshfree context when either the support size of the basis functions or the number of integration points increases. We have introduced a simple coarse-graining procedure for matrix structure creation and filling, where we change from an integration point perspective to one based on cells/elements. As a result of this optimization, the dependence of the computational time on

the number of integration points is completely severed in the sparse structure creation and dramatically decreased in the matrix assembly. We tested the new implementation on a 2D heat diffusion PDE and 3D linear elasticity problem, speeding-up ten times the structure creation and five times the filling in the case of twelve(2D)/fifteen(3D) Gauss points and $\gamma = 0.8$. Additionally, a compressed memory storage for LME approximants has been introduced to alleviate memory requirements. We have shown how this methodology can recover with minimal computational overhead the basis functions, gradients and Hessians that are repeatedly required in large-scale nonlinear or evolution problems, hence reducing drastically the amount of memory by 20−fold for a scalar fourth-order PDE in 2D.

Further research in Galerkin meshfree methods should focus on tridimensional problems and the study of proper parallelization algorithms for supercomputing. We have successfully parallelized the presented techniques using state-of-the-art scientific codes such as PETSc (portable, extensible toolkit for scientific computation library, [44]) and ParMetis [45] for reordering and partitioning. Our current experience on a supercomputing facility further highlights the importance of optimizations such as those presented here in large-scale vectorial problems in 3D. Models showing an intrinsic high computational cost such as the phase-field approaches can particularly benefit from this concept due to the easy parallelization of the algorithms presented. The approximation of phase-field models with LME in biomembrane dynamics [32, 33] and fracture mechanics [35, 36] are successful examples of these optimization procedures.

Acknowledgments

We acknowledge the support of the European Research Council under the European Community's 7th Framework Programme (FP7/2007-2013)/ERC grant agreement nr 240487, and of the Ministerio de Ciencia e Innovación (DPI2011-26589). CP acknowledges FPI-UPC Grant and FPU Ph. D. Grant (Ministry of Science and Innovation, Spain).

Appendix A. Optimal storage of local maximum-entropy approximants

We review here the calculation of local maximum-entropy (LME) basis functions and their derivatives. We represent spatial gradients of scalar functions by ∇, and we denote by $D\boldsymbol{f}(\boldsymbol{x})$ the matrix of partial derivatives for vector-valued functions. The subindexes a and b refer to nodes. Summation is not implied for repeated node indices (see [22, 25, 28] for further explanation).

Let X be a set of N scattered nodes $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N\} \subset \mathbb{R}^d$, where $d = 1, 2, 3$ is the spatial dimension, and their associate set of locality parameters $\{\beta_1, \beta_2, \dots, \beta_N\} \subset \mathbb{R}$. Given a point $\boldsymbol{x}$, recall that the primal list $\mathcal{N}_{\boldsymbol{x}}^X$ contains the indices of the nodes affecting $\boldsymbol{x}$. The evaluation of the basis function corresponding to the nodal point a is computed as

$$p_a(\boldsymbol{x}) = \frac{\exp[-\beta_a|\boldsymbol{x} - \boldsymbol{x}_a|^2 + \boldsymbol{\lambda}^* \cdot (\boldsymbol{x} - \boldsymbol{x}_a)]}{Z(\boldsymbol{x})}, \tag{A.1}$$

where $Z(\boldsymbol{x})$ is a partition function

$$Z(\boldsymbol{x}) = \sum_{a \in \mathcal{N}_{\boldsymbol{x}}^X} \exp[-\beta_a |\boldsymbol{x} - \boldsymbol{x}_a|^2 + \boldsymbol{\lambda}^* \cdot (\boldsymbol{x} - \boldsymbol{x}_a)],$$

and the Lagrange multiplier $\boldsymbol{\lambda}^*$ is the minimizer of the cost function $\ln Z(\boldsymbol{x}, \boldsymbol{\lambda})$ [22], that is

$$\boldsymbol{\lambda}^*(\boldsymbol{x}) = \arg\min_{\boldsymbol{\lambda} \in \mathbb{R}^d} \ln Z(\boldsymbol{x}, \boldsymbol{\lambda}).$$

The first spatial derivatives of the basis functions (gradient) are computed as [22, 25]

$$\nabla p_a(\boldsymbol{x}) = p_a \left[\boldsymbol{r}_\beta - \boldsymbol{M}_a (\boldsymbol{x} - \boldsymbol{x}_a) \right] \in \mathbb{R}^d, \tag{A.2}$$

where

$$\boldsymbol{r}_\beta(\boldsymbol{x}) = 2 \sum_{b \in \mathcal{N}_{\boldsymbol{x}}^X} \beta_b p_b (\boldsymbol{x} - \boldsymbol{x}_b) \in \mathbb{R}^d, \quad \boldsymbol{M}_a = 2\beta_a \boldsymbol{I} - D\boldsymbol{\lambda} \in \mathbb{R}^{d \times d}, \quad D\boldsymbol{\lambda}(\boldsymbol{x}) = \left(\boldsymbol{J}_\beta - \boldsymbol{I} \right) \boldsymbol{J}^{-1} \in \mathbb{R}^{d \times d},$$

$$\boldsymbol{J}_\beta(\boldsymbol{x}) = 2 \sum_{b \in \mathcal{N}_{\boldsymbol{x}}^X} \beta_b p_b \, (\boldsymbol{x} - \boldsymbol{x}_b) \otimes (\boldsymbol{x} - \boldsymbol{x}_b) \in \mathbb{R}^{d \times d}, \text{ and } \boldsymbol{J}(\boldsymbol{x}) = \sum_{b \in \mathcal{N}_{\boldsymbol{x}}^X} p_b (\boldsymbol{x} - \boldsymbol{x}_b) \otimes (\boldsymbol{x} - \boldsymbol{x}_b) \in \mathbb{R}^{d \times d}.$$

The second spatial derivatives of the basis functions (Hessian matrix) can be written as [28]

$$\begin{aligned} H p_a(\boldsymbol{x}) = \; & p_a \left[\boldsymbol{r}_\beta - \boldsymbol{M}_a (\boldsymbol{x} - \boldsymbol{x}_a) \right] \otimes \left[\boldsymbol{r}_\beta - \boldsymbol{M}_a (\boldsymbol{x} - \boldsymbol{x}_a) \right] \\ & + p_a \left[\boldsymbol{r}_\beta \otimes \boldsymbol{r}_\beta + \boldsymbol{r}_\beta \otimes \boldsymbol{j}_a + \boldsymbol{j}_a \otimes \boldsymbol{r}_\beta + \left(\boldsymbol{r}_\beta \cdot \boldsymbol{j}_a \right) \boldsymbol{I} \right] \\ & + p_a \left[2(\bar{\beta} - \beta_a) \boldsymbol{I} - \boldsymbol{Q} - \boldsymbol{j}_a \cdot \boldsymbol{T} \right] \quad \in \mathbb{R}^{d \times d}, \end{aligned} \tag{A.3}$$

where

$$\boldsymbol{j}_a = \boldsymbol{J}^{-1} (\boldsymbol{x} - \boldsymbol{x}_a) \in \mathbb{R}^d, \quad \boldsymbol{Q}(\boldsymbol{x}) = \sum_{b \in \mathcal{N}_{\boldsymbol{x}}^X} p_b \boldsymbol{M}_b (\boldsymbol{x} - \boldsymbol{x}_b) \otimes \boldsymbol{M}_b (\boldsymbol{x} - \boldsymbol{x}_b) \in \mathbb{R}^{d \times d},$$

$$\bar{\beta}(\boldsymbol{x}) = \sum_{b \in \mathcal{N}_{\boldsymbol{x}}^X} \beta_b p_b \in \mathbb{R}, \text{ and } \boldsymbol{T}(\boldsymbol{x}) = \sum_{b \in \mathcal{N}_{\boldsymbol{x}}^X} p_b (\boldsymbol{x} - \boldsymbol{x}_b) \otimes \boldsymbol{M}_b (\boldsymbol{x} - \boldsymbol{x}_b) \otimes \boldsymbol{M}_b (\boldsymbol{x} - \boldsymbol{x}_b) \in \mathbb{R}^{d \times d \times d}.$$

Appendix B. Quantification of memory usage

We quantify here the memory usage for two different strategies to store local maximum-entropy basis functions and their derivatives: the *full storage* and the *optimal* or *compressed storage* methods.

The basis functions are usually computed and stored in memory for a given a set of L quadrature points $\{\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_L\} \subset \mathbb{R}^d$ and the associated set of primal lists $\{\mathcal{N}_{\boldsymbol{y}_1}^X, \mathcal{N}_{\boldsymbol{y}_2}^X, \ldots, \mathcal{N}_{\boldsymbol{y}_L}^X\}$ (see Section 2). By defining as $n_k = |\mathcal{N}_{\boldsymbol{y}_k}^X|$ the cardinality corresponding to the primal list of the

Table B.2: Quantification of memory usage for two methods that store local maximum-entropy basis functions and their derivatives. The *optimal* or *compressed storage* (OS) technique needs approximately $L\cdot(2+d)\cdot\left(1+\frac{3}{2}d+\frac{1}{2}d^2\right)$ doubles, while the *full storage* (FS) method demands $L\cdot\bar{n}\cdot\left(1+\frac{3}{2}d+\frac{1}{2}d^2\right)$ doubles, where L is the number of quadrature points, d the spatial dimension, and $\bar{n}\gg(2+d)$ the mean cardinality of the primal lists. The ratio M_{FS}/M_{OS} shows that memory usage decreases significantly when the compressed storage technique is used.

	Full storage		Optimal storage	
	Variable	Memory usage	Variable	Memory usage
Basis functions	p_a	$L\cdot\bar{n}$	$Z(\boldsymbol{x})$ $\boldsymbol{\lambda}(\boldsymbol{x})$	L $L\cdot d$
First spatial derivatives	∇p_a	$L\cdot\bar{n}\cdot d$	$\boldsymbol{r}_\beta(\boldsymbol{x})$ $D\boldsymbol{\lambda}(\boldsymbol{x})$	$L\cdot d$ $L\cdot d\cdot(d+1)/2$
Second spatial derivatives	Hp_a	$L\cdot\bar{n}\cdot d\cdot(d+1)/2$	$\bar{\beta}(\boldsymbol{x})$ $\boldsymbol{J}(\boldsymbol{x})$ $\boldsymbol{Q}(\boldsymbol{x})$ $\boldsymbol{T}(\boldsymbol{x})$	L $L\cdot d\cdot(d+1)/2$ $L\cdot d\cdot(d+1)/2$ $L\cdot d\cdot d\cdot(d+1)/2$
Total memory usage	$M_{FS}=L\cdot\bar{n}\cdot\left(1+\frac{3}{2}d+\frac{1}{2}d^2\right)$ $\bar{n}\gg(2+d)$		$M_{OS}=L\cdot\left(2+\frac{7}{2}d+2d^2+\frac{1}{2}d^3\right)$ $M_{OS}\approx L\cdot(2+d)\cdot\left(1+\frac{3}{2}d+\frac{1}{2}d^2\right)$	
Comparison	$M_{FS}/M_{OS}\approx\bar{n}/(2+d)\gg 1$			

quadrature point $\boldsymbol{y}_k$, we can construct the set of cardinalities $\{n_1,n_2,\ldots,n_L\}\subset\mathbb{R}$. To simplify the calculations, we define the mean cardinality as $\bar{n}=(\sum_{k=1}^{L}n_k)/L$.

The *full storage* (FS) method demands a massive usage of memory because basis functions and first and second derivatives associated to all the nodal points, and evaluated at all the quadrature points, need to be stored in memory. The calculation of memory usage is straightforward from the analysis of Eqs. A.1, A.2 and A.3 (see Table B.2 for a summary): $M_{FS}=L\cdot\bar{n}\cdot(1+d+d(d+1)/2)=L\cdot\bar{n}\cdot\left(1+\frac{3}{2}d+\frac{1}{2}d^2\right)$ doubles, where here and elsewhere we exploit the symmetry of matrices (here the Hessian) to reduce storage. On the other hand, the *optimal* or *compressed storage* (OS) method only requires the storage of some specific variables associated to the quadrature points. We quantify the memory usage of this method in Table B.2: $M_{OS}=L\cdot\left(2+\frac{7}{2}d+2d^2+\frac{1}{2}d^3\right)\approx L\cdot(2+d)\cdot\left(1+\frac{3}{2}d+\frac{1}{2}d^2\right)$ doubles. As the mean cardinality is regularly much greater than the spatial dimension, i.e., $\bar{n}\gg(2+d)$, from the ratio $M_{FS}/M_{OS}=\bar{n}/(2+d)$ we can conclude that the memory usage decreases significantly when the compressed storage technique is used.

Appendix C. Data structure and specialized algorithms

We detail here the data structures and algorithms proposed to handle more efficiently sparse matrices in the context of meshfree methods. The data structures are specifically designed to store the neighborhood index sets for particular "entities" (elements, nodes, or quadrature points). The algorithms described are responsible for the creation of the sparse matrix structure and the assembly process.

Appendix C.1. Data structure to store neighbor lists

The data structure to store neighbor lists is inspired in the compressed sparse row storage format (see Section 2) and consists of two arrays, one indicating the number of neighboring points to an entity (*pointer_array*), and the other containing the index or identification number of each one of these points (*index_array*). Depending on the kind of assembly process, we need to construct at least two of the following four neighborhood index sets:

- Primal lists: set of neighboring nodes to each quadrature point. These lists, stored in the arrays is_n and js_n, are required for the assembly process based on a loop over the quadrature points (see Section 2).
- Dual lists: set of the neighboring quadrature points to each nodal point (see Section 2 for details). These lists, stored in the arrays in_s and jn_s, are dual to the primal lists.
- Lists of the neighboring nodal points to each cell/element, stored in the arrays ie_n and je_n. These sets, defined in Section 3.1, are needed for the assembly process based on a loop over the cell/elements.
- Lists of the neighboring cell/elements to each nodal point (arrays in_e and jn_e). These lists are dual to those contained in the set of arrays ie_n and je_n.

We use here the primal lists to explain how the information of the neighborhood index sets is stored in the arrays *pointer_array* and *index_array*, which in this work are respectively referred as is_n and js_n. Given a set of L quadrature points $\{\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_L\} \subset \mathbb{R}^d$ and the associated set of primal lists $\{\mathcal{N}^X_{\boldsymbol{y}_1}, \mathcal{N}^X_{\boldsymbol{y}_2}, \ldots, \mathcal{N}^X_{\boldsymbol{y}_L}\}$, where d is the spatial dimension, $\mathcal{N}^X_{\boldsymbol{y}_k} = \{a \in \{1, 2, .., N\} \mid p_a(\boldsymbol{y}_k) > \texttt{Tol}_0\}$ the primal list for the quadrature point $\boldsymbol{y}_k$, N the total number of nodes, $\texttt{Tol}_0$ a numerical tolerance, and $p_a(\boldsymbol{y}_k)$ the evaluation at the point $\boldsymbol{y}_k$ of the basis function corresponding to the node a, the information stored in the arrays is the following:

- is_n: the component $p + 1$ of this array is defined as $is\_n(p + 1) = \sum_{k=1}^{p} |\mathcal{N}^X_{\boldsymbol{y}_k}|$. In other words, the element (or position) $p + 1$ of the array contains the summation of the cardinalities of the primal lists associated with the first p quadrature points. Note that the first component is always zero and the length of the array is $\dim(is\_n) = L + 1$.

- js_n: this array, which stores consecutively in memory all the primal lists, is defined as $js\_n = (\mathcal{N}^X_{y_1}, \mathcal{N}^X_{y_2}, \ldots, \mathcal{N}^X_{y_L})$. The length of the array is $\dim(js\_n) = \sum_{k=1}^{L} |\mathcal{N}^X_{y_k}|$, where the cardinality can be different for each quadrature point. Note that the order of the quadrature points is important and, in general, $\dim(js\_n) \ll N \cdot L$.

Appendix C.2. Algorithms for matrix structure creation and assembly process

The algorithms implemented to create the sparse matrix structure and the assembly process are presented here in a C/C++ pseudo-code (declarations are left out for the sake of clarity). The three routines detailed in the subsequent sections are:

- CreateElementBasicStructure1D(): this algorithm creates the arrays $ia1$ and $ja1$ of the sparse matrix structure for the case in which the physical field is scalar. The neighbor lists is_n and js_n are used in the method based on a loop over the quadrature points, and the lists ie_n, je_n in the cell/element scheme.

- CreateStructureND(): extension of the previous algorithm to the n-dimensional case, i.e., when the physical field is vectorial. The arrays created are denoted by ia and ja.

- FillStructureND(): algorithm to fill the array an by executing the operations implemented in the pointer function $*pfunction$. The arrays ia and ja are needed in the assembly process to loop over the rows and columns of the sparse matrix.

Appendix C.2.1. CreateElementBasicStructure1D()

```
/* 1. Method based on a loop over the quadrature points */
// Input data:
//   - is_n and js_n: lists of the neighboring nodes to the quadrature points
//   - in_s and jn_s: lists of the neighboring quadrature points to the nodal points

// Creation of list ia1 for the case in which the physical field is scalar
iwa=new int[nPts];  // nPts: number of nodal points
for (i=0;i<nPts;i++) iwa[i]=0;  // auxiliary arrays
sumrow=0; l_ia1=nPts+1; ia1=new int[nPts+1]; ia1[0]=0;  // auxiliary arrays

// loop over matrix rows
for (i=0;i<nPts;i++){
    // loop over the neighboring quadrature points to a nodal point
    for (j=in_s[i];j<in_s[i+1];j++){
        // loop over the neighboring nodes to a quadrature point
        for (k=is_n[jn_s[j]];k<is_n[jn_s[j]+1];k++) iwa[js_n[k]]=1;
    }
    // loop over all the nodes
    for (kk=0;kk<nPts;kk++){ sumrow+=iwa[kk]; iwa[kk]=0; }
```

```
    ia1[i+1]=ia1[i]+sumrow;
    sumrow = 0;
}

// Creation of list ja1 for the case in which the physical field is scalar
l_ja1=ia1[nPts];
ja1=new int[ia1[nPts]];
std::set<int> row_list;
std::set<int>::const_iterator

// loop over matrix rows
for (i=0;i<nPts;i++){
    // loop over the neighboring quadrature points to a nodal point
    for (j=in_s[i];j<in_s[i+1];j++){
        // loop over the neighboring nodes to a quadrature point
        for (k=is_n[jn_s[j]];k<is_n[jn_s[j]+1];k++) row_list.insert(js_n[k]);
    }
    sit (row_list.begin()),
    send(row_list.end());
    for (kk=0;sit!=send;++sit,kk++) ja1[ia1[i]+kk]=*sit;
    row_list.clear();
}

/* 2. Method based on a loop over cell/elements */
// Input data:
//   - ie_n and je_n: lists of the neighboring nodal points to the elements
//   - in_s and jn_s: lists of the neighboring quadrature points to the nodal points

// Creation of dual lists in_e, jn_e
in_e=new int[nPts+1];
jn_e=new int[ie_n[nElem]];
for (i=0;i<nPts+1;i++) in_s[i]=0;
for (i=0;i<nElem;i++) for (j=ie_n[i];j<ie_n[i+1];j++) in_e[je_n[j]+1]+=1;
for (i=0;i<nPts;i++) in_e[i+1]+=in_e[i];

count=new int[nPts];
for (i=0;i<nPts;i++) count[i]=0;

// loop over elements
for (i=0;i<nElem;i++) {
    // loop over the neighboring nodes to an element
    for (j=ie_n[i];j<ie_n[i+1];j++){
        jn_e[in_s[je_n[j]]+count[je_n[j]]]=i;
```

```
        count[je_n[j]]+=1;
    }
}

// Creation of list ia1 for the case in which the physical field is scalar
iwa=new int[nPts];
for (i=0;i<nPts;i++) iwa[i]=0;
sumrow=0;
l_ia1=nPts+1; ia1=new int[nPts+1]; ia1[0]=0;

// loop over the rows
for (i=0;i<nPts;i++){
    // loop over the neighboring elements to a node
    for (j=in_e[i];j<in_e[i+1];j++){
        pos=ie_n[jn_e[j]];
        for (k=pos;k<ie_n[jn_e[j]+1];k++) iwa[je_n[k]]=1;
    }
    for (kk=0;kk<nPts;kk++){
        sumrow+=iwa[kk];
        iwa[kk]=0;
    }
    ia1[i+1]=ia1[i]+sumrow;
}

// Creation of list ja1 for the case in which the physical field is scalar
l_ja1=ia1[nPts];
ja1=new int[ia1[nPts]];
std::set<int> row_list;
std::set<int>::const_iterator

// loop over the rows
for (i=0;i<nPts;i++){
    // loop over the neighboring elements to a node
    for (j=in_e[i];j<in_e[i+1];j++){
        pos=ie_n[jn_e[j]];
        for (k=pos;k<ie_n[jn_e[j]+1];k++) row_list.insert(je_n[k]);
    }
    sit (row_list.begin()),
    send(row_list.end());
    for (kk=0;sit!=send;++sit,kk++) ja1[ia1[i]+kk]=*sit;
    row_list.clear();
}
```

Appendix C.2.2. CreateStructureND()

```
l_ia=(l_ia1-1)*nDim+1;
ia=new int[l_ia];
l_ja=l_ja1*nDim*nDim;
ja=new int[l_ja];
an=new double[l_ja];  // matrix array

// Creation of ia for the case in which the physical field is vectorial
ia[0]=0;
for (i=0;i<l_ia1-1;i++){
    size=ia1[i+1]-ia1[i];
    for (j=0;j<nDim;j++) ia[nDim*i+1+j]=ia[nDim*i]+(j+1)*(size*nDim);
}

// Creation of ja for the case in which the physical field is vectorial
for (i=0;i<l_ia1-1;i++){
    size=ia1[i+1]-ia1[i];
    for (j=0;j<size;j++){
        siseJ=nDim*ja1[ia1[i]+j];
        for (k=0;k<nDim;k++){
            for (kk=0;kk<nDim;kk++) ja[ia[nDim*i+k]+(nDim*j+kk)]=siseJ+kk;
        }
    }
}
```

Appendix C.2.3. FillStructureND()

```
/* 1. Method based on a loop over the quadrature points */

M=new double[nDim*nNNMax*nDim*nNNMax];  // quadrature point local matrix

for (k=0;k<sPts;k++){  // loop over quadrature points (sPts is the number of Gauss points)
    size=is_n[k+1]-is_n[k];
    for (i=0;i<nDim*nNNMax*nDim*nNNMax;i++) M[i]=0.0;
    for (i=0;i<size;i++){  // loop over neighbors
        for (j=0;j<size;j++){  // loop over neighbors
            for (ii=0;ii<nDim*nDim;ii++) A[ii]=0.0;
            (*pfunction)(A,parameters,shape_functions);  // operation --> get matrix A
            // fill local matrix M
            for (ii=0;ii<nDim;ii++)
                for (jj=0;jj<nDim;jj++)
                    M[(nDim*i+ii)*(size*nDim)+(nDim*j)+jj]=A[ii*nDim+jj];
        }
```

```
    }
}

// fill global sparse matrix an with quadrature point contribution
rows=nDim*size;
for (i=0;i<rows;i++){
    if (symmetric) j_ini=i;  // symmetric
    else j_ini=0;
    inc_i=i%nDim;
    base_row=(int)(i/nDim);  // floor row
    genrow=js_n[is_n[k]+base_row]*nDim+inc_i;

    for (j=j_ini;j<rows;j++){
        nc_j=j%nDim;
        base_col=(int)(j/nDim);  // floor row
        gencol=js_n[is_n[k]+base_col]*nDim+inc_j;

        for (kk=ia[genrow];kk<ia[genrow+1];kk++){
            if (ja[kk]==gencol){
               an[kk]+=M[i*rows+j];
               break;
            }
        }
    }
}

/* 2. Method based on a loop over cell/elements */

M=new double[nDim*nNNMax*nDim*nNNMax];  // cell/element local matrix

for (k=0;k<nElem;k++){  // loop over elements
    size=ie_n[k+1]-ie_n[k];
    for (i=0;i<nDim*nNNMax*nDim*nNNMax;i++) M[i]=0.0;
    for (i=0;i<size;i++){  // loop over neighbors
        for (j=0;j<size;j++){  // loop over neighbors
            for (ii=0;ii<nDim*nDim;ii++) A[ii]=0.0;
            (*pfunction)(A,parameters,shape_functions);  // operation --> get matrix A
            // fill local matrix
            for (ii=0;ii<nDim;ii++)
                for (jj=0;jj<nDim;jj++)
                    M[(nDim*i+ii)*(size*nDim)+(nDim*j)+jj]=A[ii*nDim+jj];
        }
    }
```

```
}

// fill global sparse matrix an with quadrature point contribution
rows=nDim*size;
for (i=0;i<rows;i++){
    if (symmetric) j_ini=i;  // symmetric
    else j_ini=0;
    inc_i=i%nDim;
    base_row=(int)(i/nDim);  // floor row
    genrow=je_n[ie_n[k]+base_row]*nDim+inc_i;

    for (j=j_ini;j<rows;j++){
        nc_j=j%nDim;
        base_col=(int)(j/nDim);  // floor row
        gencol=je_n[ie_n[k]+base_col]*nDim+inc_j;

        for (kk=ia[genrow];kk<ia[genrow+1];kk++){
            if (ja[kk]==gencol){
               an[kk]+=M[i*rows+j];
               break;
            }
        }
    }
}
```

References

1. Belytschko, T., Krongauz, Y., Organ, D., Fleming, M., Krysl, P.. Meshless methods: An overview and recent developments. *Computer Methods in Applied Mechanics and Engineering* 1996;**139**(1):3–47.

2. Li, S., Liu, W.K.. Meshfree and particle methods and their applications. *Applied Mechanics Reviews* 2002;**55**(1):1–34.

3. Fries, T., Matthies, H.. Classification and overview of meshfree methods. Tech. Rep.; Institute of Scientific Computing, Technical University Braunschweig, Germany; July 2004.

4. Huerta, A., Belytschko, T., Fernández-Méndez, S., Rabczuk, T.. *Meshfree Methods*; vol. 1 of *Encyclopedia of Computational Mechanics. E. Stein and R. de Borst and T.J.R. Hughes (eds.)*; chap. 10. John Wiley & Sons, Ltd.; 2004, p. 279–309.

5. Fasshauer, G.E.. *Meshfree Methods*; chap. 2. Handbook of Theoretical and Computational Nanotechnology. M. Rieth and W. Schommers (eds.). American Scientific Publishers; 2006, p. 33–97.

6. Nguyen, V.P., Rabczuk, T., Bordas, S., Duflot, M.. Meshless methods: A review and computer implementation aspects. *Mathematics and Computers in Simulation* 2008; **79**(3):763–813.

7. Chen, J.S., Pan, C., Wu, C.T., Liu, W.K.. Reproducing kernel particle methods for large deformation analysis of nonlinear structures. *Computer Methods in Applied Mechanics and Engineering* 1996;**139**:195–227.

8. Chen, J.S., Pan, C., Rogue, C.M.O.L., Wang, H.P.. A lagrangian reproducing kernel particle method for metal forming analysis. *Computational Mechanics* 1998;**22**:289–307.

9. Li, B., Habbal, F., Ortiz, M.. Optimal transportation meshfree approximation schemes for fluid and plastic flows. *International Journal for Numerical Methods in Engineering* 2010;**83**(12):1541–1579.

10. Combe, U.H., Korn, C.. An adaptive approach with the element-free-galerkin method. *Computer Methods in Applied Mechanics and Engineering* 1998;**162**:203–222.

11. Duarte, C.A., Oden, J.T.. An h–p adaptive method using clouds. *Computer Methods in Applied Mechanics and Engineering* 1996;**139**:237–262.

12. Dolbow, J., Belytschko, T.. Numerical integration of the galerkin weak form in meshfree methods. *Computational Mechanics* 1999;**23**:219–230.

13. Babuška, I., Banerjee, U., Osborn, J.E., Li, Q.. Quadrature for meshless methods. *International Journal for Numerical Methods in Engineering* 2008;**76**:1434–1470.

14. Fernández-Méndez, S., Huerta, A.. Imposing essential boundary conditions in mesh-free methods. *Computer Methods in Applied Mechanics and Engineering* 2004;**193**(12–14):1257–1275.

15. Monaghan, J.. An introduction to SPH. *Computer Physics Communications* 1988;**48**:89–96.

16. Liu, W., Jun, S., Zhang, Y.. Reproducing kernel particle methods. *International Journal for Numerical Methods in Fluids* 1995;**20**:1081–1106.

17. Melenk, J.M., Babuška, I.. The partition of unity finite element method : Basic theory and applications. *Computer Methods in Applied Mechanics and Engineering* 1996;**139**(1–4):289–314.

18. Belytschko, T., Lu, Y.Y., Gu, L.. Element free Galerkin methods. *International Journal for Numerical Methods in Engineering* 1994;**37**:229–256.

19. De, S., Bathe, K.J.. The method of finite spheres. *Computational Mechanics* 2000; **25**:329–345.

20. Hong, J.W., Bathe, K.J.. Coupling and enrichment schemes for finite element and finite sphere discretizations. *Computers and Structures* 2005;**83**:1386–1395.

21. Sukumar, N.. Construction of polygonal interpolants: a maximum entropy approach. *International Journal for Numerical Methods in Engineering* 2004;**61**(12):2159–2181.

22. Arroyo, M., Ortiz, M.. Local maximum-entropy approximation schemes: a seamless bridge between finite elements and meshfree methods. *International Journal for Numerical Methods in Engineering* 2006;**65**(13):2167–2202.

23. Sukumar, N., Malsch, E.A.. Recent advances in the construction of polygonal finite element interpolants. *Archives of Computational Methods in Engineering* 2006;**13**(1):129–163.

24. Sukumar, N., Wright, R.W.. Overview and construction of meshfree basis functions: From moving least squares to entropy approximants. *International Journal for Numerical Methods in Engineering* 2007;**70**(2):181–205.

25. Rosolen, A., Millán, D., Arroyo, M.. On the optimum support size in meshfree methods: a variational adaptivity approach with maximum entropy approximants. *International Journal for Numerical Methods in Engineering* 2010;**82**(7):868–895.

26. Ullah, Z., Coombs, W.M., Augarde, C.E.. An adaptive finite element/meshless coupled method based on local maximum entropy shape functions for linear and nonlinear problems. *Computer Methods in Applied Mechanics and Engineering* 2013;**267**:111–132.

27. Hale, J.S., Baiz, P.M.. A locking-free meshfree method for the simulation of shear-deformable plates based on a mixed variational formulation. *Computer Methods in Applied Mechanics and Engineering* 2012;**241-244**:311–322.

28. Millán, D., Rosolen, A., Arroyo, M.. Thin shell analysis from scattered points with maximum-entropy approximants. *International Journal for Numerical Methods in Engineering* 2011;**85**(6):723–751.

29. Millán, D., Rosolen, A., Arroyo, M.. Nonlinear manifold learning for meshfree finite deformation thin shell analysis. *International Journal for Numerical Methods in Engineering* 2013;**93**(7):685–713.

30. Nissen, K., Cyron, C.J., Gravemeier, V., Wall, W.A.. Information-flux method: a meshfree maximum-entropy petrov-galerkin method including stabilised finite element methods. *Computer Methods in Applied Mechanics and Engineering* 2012;**241-244**:225–237.

31. Wu, C.T., Young, D.L., Hong, H.K.. Adaptive meshless local maximum-entropy finite element method for convection-diffusion problems. *Computational Mechanics* 2014; **53**:189–200.

32. Rosolen, A., Peco, C., Arroyo, M.. An adaptive meshfree method for phase-field models of biomembranes. Part I: approximation with maximum-entropy approximants. *Journal of Computational Physics* 2013;**249**:303–319.

33. Peco, C., Rosolen, A., Arroyo, M.. An adaptive meshfree method for phase-field models of biomembranes. Part II: a Lagrangian approach for membranes in viscous fluids. *Journal of Computational Physics* 2013;**249**:320–336.

34. Amiri, F., Anitescu, C., Arroyo, M., Bordas, S.P.A., Rabczuk, T.. XLME interpolants, a seamless bridge between XFEM and enriched meshless methods. *Computational Mechanics* 2014;**53**:45–57.

35. Amiri, F., Millán, D., Shen, Y., Rabczuk, T., Arroyo, M.. Phase-field modeling of fracture mechanics in linear thin shells. *Theoretical and Applied Fracture Mechanics* 2014;**69**:102–109.

36. Li, B., Peco, C., Millán, D., Arias, I., Arroyo, M.. Phase-field modeling and simulation of fracture in brittle materials with strongly anisotropic surface energy. *International Journal for Numerical Methods in Engineering* 2014;:DOI: 10.1002/nme.4726.

37. Arroyo, M., Ortiz, M.. *Meshfree Methods for Partial Differential Equations III*; vol. 57 of *Lecture Notes in Computational Science and Engineering*; chap. Local Maximum-Entropy Approximation Schemes. Springer; 2007, p. 1–16.

38. Du, Q., Gunzburger, M., Ju, L.. Meshfree, probabilistic determination of point sets and support regions for meshless computing. *Computer Methods in Applied Mechanics and Engineering* 2002;**191**(13-14):1349–1366.

39. Bradford Barber, C., Dobkin, D.P., Huhdanpaa, H.. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software* 1996;**22**:469–483.

40. Mount, M.D., Arya, S.. A library for approximate nearest neighbor searching. `http://www.cs.umd.edu/~mount/ANN/`; 2010.

41. Eijkhout, V.. Distributed sparse data structures for linear algebra operations. Technical Report CS 92-169; Computer Science Department, University of Tennessee; 1992.

42. Saad, Y.. *Iterative Methods for Linear Systems*. PWS Publishing, Boston; 1996.

43. Borden, M.J., Hughes, T.J.R., Landis, C.M., Verhoosel, C.V.. A higher-order phase-field model for brittle fracture: Formulation and analysis within the isogeometric analysis framework. *Computer Methods in Applied Mechanics and Engineering* 2014;**273**:100–118.

44. Balay, S., Brown, J., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., et al. Portable, extensible toolkit for scientific computation. http://www.mcs.anl.gov/petsc; 2013.

45. Karypis, G., Kumar, V.. Metis-ParMetis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0.3. http://www.cs.umn.edu/~metis; 2009.