Comput Mech (2012) 49:581–602
DOI 10.1007/s00466-011-0662-x

ORIGINAL PAPER

A fast object-oriented Matlab implementation of the Reproducing Kernel Particle Method

Ettore Barbieri · Michele Meo

Received: 20 June 2011 / Accepted: 1 November 2011 / Published online: 16 November 2011
© Springer-Verlag 2011

Abstract Novel numerical methods, known as Meshless Methods or Meshfree Methods and, in a wider perspective, Partition of Unity Methods, promise to overcome most of disadvantages of the traditional finite element techniques. The absence of a mesh makes meshfree methods very attractive for those problems involving large deformations, moving boundaries and crack propagation. However, meshfree methods still have significant limitations that prevent their acceptance among researchers and engineers, namely the computational costs. This paper presents an in-depth analysis of computational techniques to speed-up the computation of the shape functions in the Reproducing Kernel Particle Method and Moving Least Squares, with particular focus on their bottlenecks, like the neighbour search, the inversion of the moment matrix and the assembly of the stiffness matrix. The paper presents numerous computational solutions aimed at a considerable reduction of the computational times: the use of *kd*-trees for the neighbour search, sparse indexing of the nodes-points connectivity and, most importantly, the explicit and vectorized inversion of the moment matrix without using loops and numerical routines.

Keywords RKPM · Object-oriented · Meshless · Programming · *hp-adaptivity*

E. Barbieri · M. Meo
Department of Mechanical Engineering, University of Bath,
Claverton Down, Bath BA2 7AY, UK
e-mail: m.meo@bath.ac.uk

E. Barbieri (✉)
Solid Mechanics and Materials Engineering Group,
Department of Engineering Science, University of Oxford,
Parks Road, Oxford OX1 3PJ, UK
e-mail: ettore.barbieri@eng.ox.ac.uk

1 Introduction

Numerical methods are crucial for an accurate simulation of physical problems, as the partial differential equations describing them usually require approximation schemes for their solution. Approximation is necessary either for the complexity of the equations and/or for the complexity of the geometry of definition of these equations.

Among all the available numerical schemes, mesh-based methods have become the most popular tools for engineering analysis over the last decades in academic and industrial applications. The most conventional mesh-based numerical method is the Finite Element Method (FEM) well-known as the most thoroughly developed method in engineering. FEM is nowadays widely used by engineers in all fields and several well-assessed commercial codes are available.

However, in FEM it is very complicated to model the breakage into a large number of fragments as FEM is intrinsically based on continuum mechanics, where elements cannot be broken. The elements thus have to be either totally eroded or stay as a whole piece, but this leads to a misrepresentation of the fracture path; serious errors can also occur because the nature of the problem is non-linear.

To overcome these problems related with the existence of a mesh, a numerical scheme that relies only on nodes would be highly beneficial. These methods are called meshfree or meshless, since they do not need a mesh to construct the approximation of the solution of a given differential equation.

The very first meshless method was the *Smoothed Particle Hydrodynamics* (SPH), introduced for the study of unbounded astrophysical phenomena such as exploding stars and dust clouds [36]. Later Monaghan in [38–40] provided a more mathematical basis through the means of kernel estimates. Even though SPH was initially conceived for solving

astrophysics problem, in [28] SPH is applied for the first time in solid mechanics and to study dynamic material response under impact [29]. While SPH and their corrected versions are based on a strong form of the governing equations, other methods, developed in the 1990s, are based on a weak form.

One of the first methods based on a (global) weak form and was invented by Belytschko and his co-workers [5], who exploited the *Moving Least Squares* (MLS) approximation [25]. The MLS has its origins in the computer graphics to smooth and interpolate scattered data, thus it seems reasonable to apply this method when only scattered nodes are available because of a discretization. Belytschko et al. [5] refined and modified the work of Nayroles and called their method *Element Free Galerkin* (EFG), which is currently one of the most used meshless methods. This class of methods is consistent, up to *n*th degree depending on the polynomials used in the basis function, and quite stable although substantially more expensive than SPH.

In parallel, Liu et al. [32] developed the *Reproducing Kernel Particle Method* (RKPM), which, contrarily to MLS, uses wavelets theory. Surprisingly, the imposition of the reproducibility conditions led to shape functions almost identical to MLS. RKPM is the discrete counterpart of the Reproducing Kernel Method (RKM) [33]. In RKM, the key point is to restore the consistency condition for the kernel. It can be demonstrated that using a modified kernel [33] by the means of a *moment matrix*, the reproducibility condition up to order n is restored. This *moment matrix* has entries that are the kernel estimates of polynomial functions up to degree $2n$. These kernel estimates are convolution integrals; their discretization, where integrals are replaced with summations, is based on *particles*. However, an explicit computation of the integrals involved in RKM without discrete summations has been proposed in [1], though only for tensor-product kernels.

In [33,34] the wavelets character of RKPM allowed the extension of the method to multiple length scales. In [33] is proposed an approach to unify all the RKM under one large family and an extension to include time and spatial shifting. In the latter [34] the approach is particularized for its particle version RKPM. The Fourier analysis is used to address error estimation and convergence properties.

The similarities between the two methods became more evident in the *Moving Least-Square Reproducing Kernel Methods* (MLSRKM), where a general framework is introduced [26,35]. From the MLSRKM, both RKPM and MLS are derived, where the sum of the square errors can be interpreted in a continuous way if an inner product based on integrals is considered. These integrals are the same convolution integrals introduced in the *moment* matrix.

Applications of both RKPM and EFG have been quite successful in the recent years, especially in problems with discontinuities and singularities and a large number of papers can be found which dealt with these questions [2–4,9,5–7,12,13].

A good number of review papers and books on meshfree methods have been published in the recent years. The first review appeared in [17] almost at the beginning of the meshfree era, shortly followed by the more general overview [9]. While [17] is more focused on the mathematical properties of these methods, [9] is more suitable for an engineering audience, since it contains many details on the implementation of EFG, especially for crack problems.

The review [44] instead emphasizes the applications for large deformation problems and reviews with particular attention multiscale and particle methods. A subclass of particle methods is the molecular dynamics, widely employed in computational chemistry. This paper is the background of the book [27]. Updated reviews on meshfree methods can be found in [20,42].

Even though meshfree methods have attractive features, they possess a number of drawbacks, which still prevents as large acceptance by the community of engineers and researchers as FEM.

This paper discusses these challenges and provides numerous expedients for an efficient programming of RKPM, with the aim of improving the computational time of meshfree methods. Often the researcher willing to enter this field must spend precious time in understanding the method and developing its own meshfree software. Despite the wide literature on meshfree methods, little literature [16,24,42] is available on practical implementation of these methods. This paper therefore attempts to fill this gap. The object-oriented environment within Matlab provides ease of implementation and allows to focus on the challenging aspects; indeed the code described in this paper could be the basis of further development in compiled object-oriented programming languages such as C++.

The paper is structured as follows: Sect. 2 provides a detailed description of the RKPM, 3 shows the major challenges in RKPM/MLS and proposed practical programming solutions, while Sect. 4 compares the computational times of such solutions for a three-dimensional test case. Conclusions and proposals for future improvements are discussed in Sect. 5.

2 Description of the RKPM

The RKPM is the discrete counterpart of the RKM, which has its origins in the wavelets theory. In the RKM, the approximation $u^h(\mathbf{x})$ of a generic field variable $u(\mathbf{x})$ defined on a domain $\Omega \subset \mathbb{R}^3$ is given by

$$u^h(\mathbf{x}) = \int_\Omega C(\mathbf{x}, \mathbf{x} - \mathbf{x}')w(\mathbf{x} - \mathbf{x}')u(\mathbf{x}')\mathrm{d}\Omega' \tag{1}$$

where $C(\mathbf{x})$ is a correction to the kernel $w(\mathbf{x})$ that restores the reproducibility conditions. By *reproducibility* is intended the ability of $u^h(\mathbf{x})$ to *reproduce* a set of basis functions, once introduced in $u(\mathbf{x})$ in Eq. 1. For instance, for a polynomial basis of degree n and $d = \dim(\Omega)$

$$\mathbf{p}(\mathbf{x}) = \left\{ \mathbf{x}^{\alpha} : \sum_{i=1}^{d} \alpha_i \leq n \right\} \tag{2}$$

the reproducibility conditions are, substituting $\mathbf{p}(\mathbf{x})$ in $u(\mathbf{x})$ in Eq. 1

$$\int_{\Omega} C(\mathbf{x}, \mathbf{x} - \mathbf{x}') w(\mathbf{x} - \mathbf{x}') \mathbf{p}(\mathbf{x}') \mathrm{d}\Omega' = \mathbf{p}(\mathbf{x}) \tag{3}$$

It can be shown that, for polynomial basis (2) Eq. 1 is

$$u^h(\mathbf{x}) = \mathbf{p}^T(\mathbf{x}) \mathbf{M}(\mathbf{x})^{-1} \int_{\Omega} \mathbf{p}(\mathbf{x}) w(\mathbf{x} - \mathbf{x}') u(\mathbf{x}') \mathrm{d}\Omega' \tag{4}$$

where

$$\mathbf{M}(\mathbf{x}) = \int_{\Omega} \mathbf{p}(\mathbf{x}) \mathbf{p}^T(\mathbf{x}) w(\mathbf{x} - \mathbf{x}') \mathrm{d}\Omega' \tag{5}$$

is called the *moments matrix*.

Immediate proof follows by substituting $\mathbf{p}^T(\mathbf{x})$ in $u(\mathbf{x})$ in Eq. 4, obtaining $u^h(x) = \mathbf{p}^T(\mathbf{x})$, which means that the approximation can reproduce exactly all the polynomials in the basis function.

Sometimes it is preferred a scaled and translated version of Eq. 4, to alleviate the ill-conditioning of the moment matrix.

$$u^h(\mathbf{x}) = \mathbf{p}^T(\mathbf{0}) \mathbf{M}(\mathbf{x})^{-1} \int_{\Omega} \mathbf{p}\left(\frac{\mathbf{x}' - \mathbf{x}}{\rho}\right) w\left(\frac{\mathbf{x}' - \mathbf{x}}{\rho}\right) u(\mathbf{x}') \mathrm{d}\Omega' \tag{6}$$

where ρ is the size of the support of the kernel and

$$\mathbf{M}(\mathbf{x}) = \int_{\Omega} \mathbf{p}\left(\frac{\mathbf{x}' - \mathbf{x}}{\rho}\right) \mathbf{p}^T\left(\frac{\mathbf{x}' - \mathbf{x}}{\rho}\right) w\left(\frac{\mathbf{x}' - \mathbf{x}}{\rho}\right) \mathrm{d}\Omega'. \tag{7}$$

In [1], integrals in RKM have been resolved for a general domain of a complex shape. RKPM is the discretized version of RKM, in fact considering Eqs. 6 and 7 and substituting integrals with summation

$$u^h(\mathbf{x}) = \mathbf{p}^T(\mathbf{0}) \mathbf{M}(\mathbf{x})^{-1} \sum_{I=1}^{N} \mathbf{p}\left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right) w\left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right) \Delta V_I U_I, \tag{8}$$

ΔV_I is a measure (length, area or volume) of the discretized subdomain associated with particle I and U_I is the Ith nodal field variable (not to be confused with the actual value of the field variable at node I)

$$\mathbf{M}(\mathbf{x}) = \sum_{I=1}^{N} \mathbf{p}\left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right) \mathbf{p}^T\left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right) w\left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right) \Delta V_I. \tag{9}$$

Therefore the shape function for node I for RKPM is given by

$$\phi_I(\mathbf{x}) = C_I(\mathbf{x}) w\left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right) \Delta V_I, \tag{10}$$

$$\underbrace{C_I(\mathbf{x})}_{1\mathrm{X}1} = \underbrace{\mathbf{p}^T(\mathbf{0})}_{1\mathrm{X}k} \underbrace{\mathbf{M}(\mathbf{x})^{-1}}_{k\mathrm{X}k} \underbrace{\mathbf{p}\left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right)}_{k\mathrm{X}1}, \tag{11}$$

where k is the number of functions in the *basis*.

Derivatives for the shape functions (10) are given by

$$\frac{\partial \phi_I}{\partial x} = \frac{\partial C_I(\mathbf{x})}{\partial x} w(\mathbf{x}) + C_I(\mathbf{x}) \frac{\partial w(\mathbf{x})}{\partial x}, \tag{12}$$

where

$$\frac{\partial C_I}{\partial x} = \mathbf{p}^T(\mathbf{0}) \left[\frac{\partial \mathbf{M}(\mathbf{x})^{-1}}{\partial x} \mathbf{p} + \mathbf{M}(\mathbf{x})^{-1} \frac{\partial \mathbf{p}}{\partial x} \right] \tag{13}$$

and

$$\frac{\partial \mathbf{M}(\mathbf{x})^{-1}}{\partial x} = -\mathbf{M}^{-1} \frac{\partial \mathbf{M}}{\partial x} \mathbf{M}^{-1}. \tag{14}$$

The calculation of the derivatives of the inverse of the moment matrix can be accelerated with the means of *LU* factorization [8].

The computational burden of MLS/RKPM shape functions is showed in Eq. 8: neighbour search, computation of the moment matrix and its inversion. In the next sections, a solution to these and other issues is proposed. In fact, it is possible to accelerate remarkably the inversion of the moment matrix and its derivatives without using any numerical routines (such as Gauss elimination, LU factorization, etc…) and, most importantly, it is possible to avoid such routines at each point of evaluation.

2.1 The kernel function

The Eq. 10 states that $C_I(\mathbf{x})$ is a corrective term for the single weight (or *kernel*) function centered in $\mathbf{x}_I$.

The radius of this support is given by a parameter called *dilatation parameter* or *smoothing length* which can be indicated as ρ_I or d_I or a_I to avoid confusion with the mass density. According to the norm considered, the shape of the support may vary, for example it could be a circle but also a rectangle (Fig. 1b).

The *dilatation parameter* is crucial for the accuracy, the stability of the algorithm and plays the role of the element size in the FE, although its effects on all these aspects have not been rigorously demonstrated yet. Moreover, when variable dilatation parameters are used, particular care must be

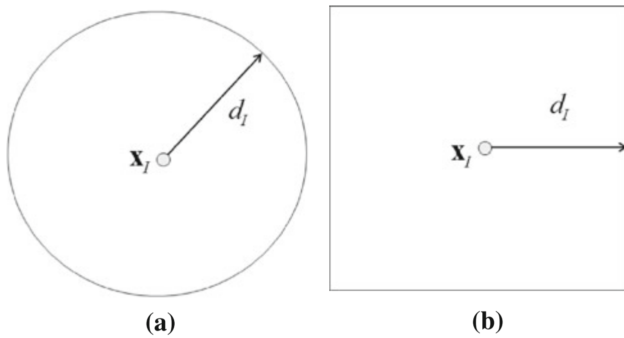


Fig. 1 Examples of compact support. **a** Circular support. **b** Rectangular support

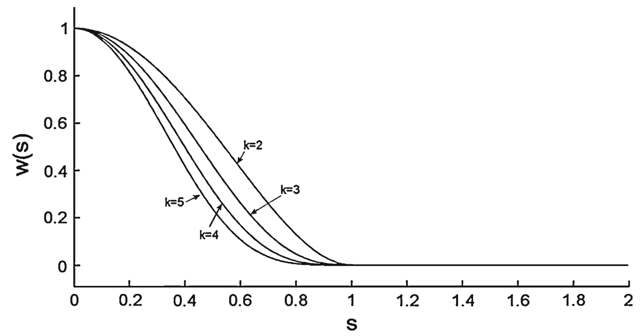


Fig. 2 Window function (19) for different values of k

taken in the choice of these quantities. As a rule of thumb, with a background element mesh, it is practical to choose

$$\rho_I = \alpha \max_e l_{eI} \tag{15}$$

where α[1] is a scaling factor depending on the degree of the polynomial basis and l_{eI} is the length of the eth edge of the all the elements having $\mathbf{x}_I$ as a node.

For a non-uniform nodal discretization, (15) assigns different dilatation parameters to each particle. This is the case of discretization with significant change in node density (e.g. to capture stress concentration, or in adaptivity). As a drawback, the resulting approximation (regardless the differential equation) is poor, since MLS *approximants* [43] degrade in the presence of sudden change in the support size. A remedy is presented in [43]: for example, smoothing lengths can be averaged over the domain. Nonetheless, the role of the dilatation parameters is still not well understood and it represents an obscure area in MLS and RKPM, which requires further mathematical work. A deeper look in the field of the approximation theory will hopefully clarify the role of the smoothing lengths and provide a more rigorous rule for their choice.

The compactness of the kernel guarantees the sparsity and the *bandedness* of the stiffness matrix in a Galerkin formulation, which is particularly useful for the computer algorithms of matrix inversion. The final characteristic of weight functions is its functional form [20].

Assuming

$$s = \frac{\|\mathbf{x} - \mathbf{x}_I\|}{a} \tag{16}$$

the properties of kernel functions are

$$w(s) \geq 0 \quad s \in [0, 1], \tag{17a}$$

$$w(1) = 0, \tag{17b}$$

$$\int_0^1 w(s)\mathrm{d}s = \frac{1}{2}, \tag{17c}$$

$$\lim_{a\to 0} w(s) = \delta(s), \tag{17d}$$

$$w(s) \text{ is monotonically decreasing with } s. \tag{17e}$$

Properties (17a) and (17b) guarantee that the kernel has compact support, while (17c) is a normalization property. Property (17d) assures that at the limit the kernel becomes a *Dirac function*. Some functional expressions of the kernel could be for example the *3rd order spline*

$$w(s) = \begin{cases} \frac{2}{3} - 4s^2 + 4s^3 & 0 \leq s \leq \frac{1}{2} \\ \frac{4}{3} - 4s + 4s^2 - \frac{4}{3}s^3 & \frac{1}{2} < s \leq 1 \\ 0 & s > 1 \end{cases} \tag{18}$$

which is $C^2(\Omega)$[2] or more generally the $2k$*th order spline* (Fig. 2)

$$w(s) = \begin{cases} (1 - s^2)^k & 0 \leq s \leq 1 \\ 0 & s > 1 \end{cases} \tag{19}$$

which is $C^{k-1}(\Omega)$ with $k > 1$.

The order of continuity of a kernel function is important because it influences the order of continuity of the shape functions. First partial derivatives of the kernel with respect to the variable x_k can be evaluated using chain rule

$$\frac{\partial w}{\partial x_k} = \frac{\partial w}{\partial s}\frac{\partial s}{\partial x_k}. \tag{20}$$

3 Issues and practical programming solutions

The major factors hindering the spreading of the meshfree methods are of practical nature. Indeed, for Eq. 8, it is necessary to invert a matrix $\mathbf{M}$, called the *moment matrix*, for each point of interest. If the number of these points is large,

[1] To avoid singularity of the moment matrix, α should be at least $2 - 2.2$ for quadratic basis in two dimensions, at least 2.4 for cubic polynomials and so on.

[2] The notation $C^k(\Omega)$ indicates a function which is continuous on the domain Ω along with its derivatives up to order k.

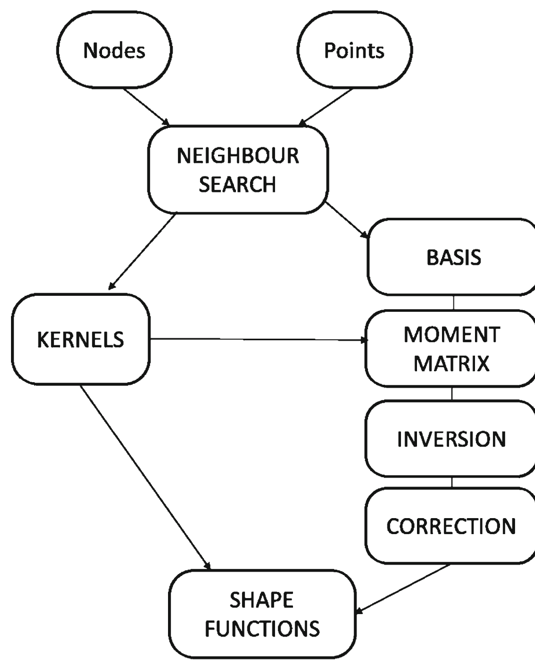


Fig. 3 Flowchart of the *classic* algorithm in meshfree methods

then this operation must be repeated several times, raising the computational cost.

Another issue regards the search for points located within the support radius. In fact, for Eq. 16, meshfree shape functions are *compact support* functions, which means that shape functions are zero outside a ball of radius ρ_I centered in the Ith node. Therefore, in Eq. 8, the summation can be limited to the nodes that have $\mathbf{x}$ in their support. Hence, to evaluate the shape functions, the following operations must be performed at every point $\mathbf{x}$ (Fig. 3) [20,24]:

– search for neighbors within the set of nodes
– calculate the moment matrix

$$\mathbf{M}(\mathbf{x}) = \sum_{I=1}^{N} \mathbf{p}\left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right) \mathbf{p}^T \left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right) w \left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right) \Delta V_I. \quad (21)$$

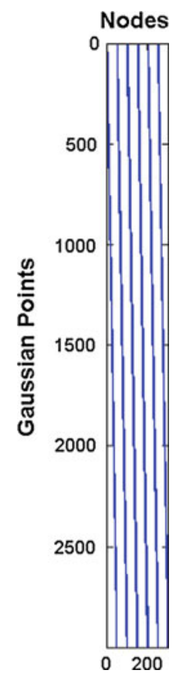
– invert the moment matrix
– apply the corrective term to the weight function

$$\phi_I(\mathbf{x}) = C_I(\mathbf{x}) w \left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right) \Delta V_I, \quad (22)$$

$$C_I(\mathbf{x}) = \mathbf{p}^T(\mathbf{0}) \mathbf{M}(\mathbf{x})^{-1} \mathbf{p} \left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right). \quad (23)$$

Also, derivatives are usually of interest in Galerkin formulations, since it is necessary to calculate the strain-displacement linear operator as in Eq. 123.

Fig. 4 Sparsity pattern for meshfree shape functions on a segment



These operations are carried out within a loop over the Gaussian points and therefore the overall code is considerably slowed down.

In this section some remedies to these implementation issues in RKPM/MLS will be presented.

As a general guideline, the double-looping over the nodes and the Gaussian points should be avoided as much as possible. This is true for every programming language but it is more evident for Matlab, where significant speeds-up can be achieved with loop *vectorization*. Moreover, the Gaussian points are usually more numerous than the nodes, therefore instead of looping over the Gaussian points and looking for the nodes, it is more convenient to focus on nodes and search for the Gaussian points included in the support.

In the following sections the typewriter character `A` will be used to indicate the computer variable, while **A** will indicate the mathematical entity (vector, matrix, etc…).

3.1 The neighbour search: *kd*-trees

Meshfree shape functions are *compact support functions*, as stated in Eq. 16 and properties (17a) and (17b).

The purpose of the compactness is to *discretize* the domain, as in FE, i.e. to give a local character to the approximation. As a result, stiffness matrix is *sparse*, i.e. with most entries equal to zero. The compactness of the shape functions can be exploited to better store the matrix entries and to efficiently evaluate the integrals in Galerkin formulations. In fact, Fig. 4 shows the sparsity pattern arising from a typical meshfree discretization over a line segment. If n_g is the number of evaluation points (rows) and n_s the number of nodes,

the meshfree shape functions resulting from a discretization can be stored in a $n_g \times n_s$ matrix PHI.

For example, for a fixed column[3] PHI contains the values of the jth shape function on all the points that belong to the support of the jth node.

$$\texttt{PHI(:, j)} \quad j = 1, \ldots, n_s. \tag{24}$$

Since on the same evaluation point there should be enough nodes for the meshfree approximation,[4] analogously, for a fixed row PHI contains the values of the shape functions of the nodes that have the ith evaluation point within their support.

$$\texttt{PHI(i, :)} \quad i = 1, \ldots, n_g. \tag{25}$$

The sparsity diagram showed in Fig. 4 is a representation of the non-zero values of PHI, where a blue dot corresponds to a non-null entry. Firstly, it can be noted that the number of evaluation points is usually much larger than the number of nodes. These evaluation points are usually Gaussian points, if a variational weak form is used.

Secondly, it can be clearly seen that the vast majority of entries is zero, because for each node, only the nearest Gaussian points are considered. Thus, it seems reasonable to store the shape functions matrix as a *sparse* matrix rather than as a full matrix. The only values stored are then the row-index, the column-index and the non-zero entry. A full matrix would have had all the values, even the zero ones, arranged in a *table-like* manner.

As an example of the amount of memory saved, a factor called density d is normally calculated for sparse matrices. It is the ratio of the number of non-zero values n_{nz} and the total number of elements of the matrix, i.e. the product of the total number of rows r for the total number of columns c. For meshfree shape functions, this factor could be a few percentages or even less, depending on the dilatation parameter.

$$d = \frac{n_{nz}}{rc}. \tag{26}$$

In order to obtain this sparse matrix, it is crucial to use an algorithm capable of searching the evaluation points within a certain radius (Fig. 5).

Given a set of n_s approximation nodes (variable GRIDSET)

$$S_1 = \{\mathbf{x}_J : J = 1, \ldots, n_s\} \subset \mathbb{R}^k \quad k = 1, 2, 3 \tag{27}$$

and a second set of n_g evaluation points (variable GGRID[5])

$$S_2 = \{\mathbf{x}_I : I = 1, \ldots, n_g\} \subset \mathbb{R}^k \quad k = 1, 2, 3 \tag{28}$$

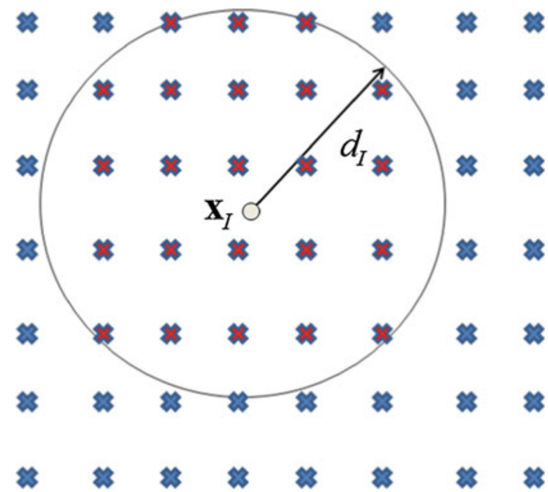


Fig. 5 Neighbour search: *red crosses* evaluation points inside the support; *blue crosses* evaluation points outside the support. (Color figure online)

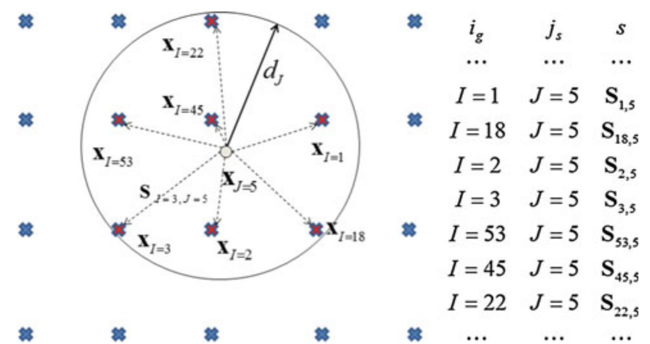


Fig. 6 Neighbour search: example of the mapping in Eq. 31

the complete *distance matrix* **S** would be a matrix $n_g \times n_s$ where

$$\mathbf{S}_{IJ} = \frac{|\mathbf{x}_I - \mathbf{x}_J|}{\rho_J}, \tag{29}$$

where ρ_J is the dilatation parameter for the Jth node.

After **S** matrix is constructed, for Eq. 16 only the values less than or equal to one must be considered. Let us call n_{nz} the number of elements in **S** satisfying

$$\mathbf{s} = \{I \in 1, \ldots n_g, J \in 1, \ldots n_s : \mathbf{S}_{IJ} \leq 1\} \tag{30}$$

then, the output of such procedure would eventually be 3 vectors of length n_{nz}, $\mathbf{i_g}$, $\mathbf{j_s}$ and **s** such that (Fig. 6)

$$\mathbf{S}_{\mathbf{i_g}_i, \mathbf{j_s}_i} = \mathbf{s}_i \quad i = 1, \ldots, n_{nz}. \tag{31}$$

This is a well-known problem in the field of computational geometry, called *neighbour search* or *range query*.

A *naive* implementation would loop over the Gaussian points S_2, loop again over the nodes S_1, compare the distance with the dilatation parameter and then decide if that Gaussian point belongs to the nodal support. Because of this double loop, this *brute force* algorithm (also known as *exhaustive*

[3] It is used the Matlab notation where the colon : in a matrix means variation over that dimension.

[4] To avoid ill-conditioning of the moment matrix [20,42].

[5] In principle different from GRIDSET, but not necessarily.

search or *sequential search*) has a computational cost $\mathcal{O}(N^2)$, with N the number of search points. If N is a large number, then the computational cost becomes prohibitive. Yet, this algorithm for its simplicity is widely used in the meshfree community as affirmed in [22] and reported in several papers [9,16,24,42].

The brute force algorithm is nonetheless inefficient when the number of nodes grows considerably. It is then not recommended especially for large scale simulations. The neighbour search is indeed recognized as one of the major bottlenecks for the meshfree technology. For a more efficient code, improvements should be directed also in this sense.

An attempt towards this scope is reported in [30] where a *bucket algorithm* is used to reduce the range of search. The algorithm allows a maximum number of nodes to fall within each bucket. If the number of nodes in the bucket exceeds a certain pre-fixed value, then the bucket is split in two parts. The subdivision is iterated until every bucket has a number of nodes inferior to a defined one. This is roughly the concept behind a more powerful tool, i.e. *kd-tree*.

A *kd-tree* is a generalization of binary trees to k-dimensional data. Binary trees are studied in *computer science* as *data structures* that emulate a hierarchical tree with a set of linked nodes. Every node is connected to at most 2 other nodes, called *children*. If each node is connected to 4 children, then the tree is called a *quadtree*, if connected to 8 children the tree is a *octree*. The quadtree idea is identical to the *bucket* idea: it is normally used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions [19]. The three-dimensional analogue is called *octree*. Quadtrees have been used in meshfree methods as background *mesh* generators [11,21,23,37] and also for adaptivity [45]. The use of binary trees is different in this context. In this paper binary trees are used not to generate a hierarchical mesh but rather as a tool for building a connectivity map between nodes and Gaussian points.

In a recent work [18] *binary tree* methods have been applied in meshfree methods, although for a different meshless method (Radial Basis Function).

A *kd*-tree (where k stays for k-dimensional tree) is a space-partitioning data structure for organizing N points in a k-dimensional space. A complete exposition of *kd-trees* is beyond the aims of this paper, but main concepts will be briefly recalled. The interested reader can refer to specialized textbooks as [15] or [41] for more formal definitions and details. The space-partitioning achieved by hierarchically decomposing a set of points into smaller sets of data, until each subset has the same number of points. Figure 7a shows the final space partition obtained with the *kd-tree*, whilst Fig. 7b shows the correspondent hierarchical data structure.

The first step consists in picking a point (the *root*) and considering a plane perpendicular to one of the axis and passing through the root. The whole domain is then split in two halves, each one containing roughly the same number of nodes. For each of the two sub-partitions, a node is chosen again, but this time the splitting plane will be on a different dimension of the previous one. The two nodes are called *children*. The procedure is then repeated for the children, generating other children, until the final sub-partitions will not contain any node. These final children are called *leaves*. From the root to the leaves, every node of the tree represents a point. In meshfree methods, two steps are necessary to build the final connectivity matrix: the construction of the tree and the *range query*. The computational cost for building the tree is $\mathcal{O}(kN \log N)$ for points in $\mathbb{R}^k$ and the range query takes only $\mathcal{O}(\log N)$, while for the brute force algorithm the cost for range query is $\mathcal{O}(N)$. In [24] it is recommended to use $\mathcal{O}(\log N)$ algorithm for the neighbour search.

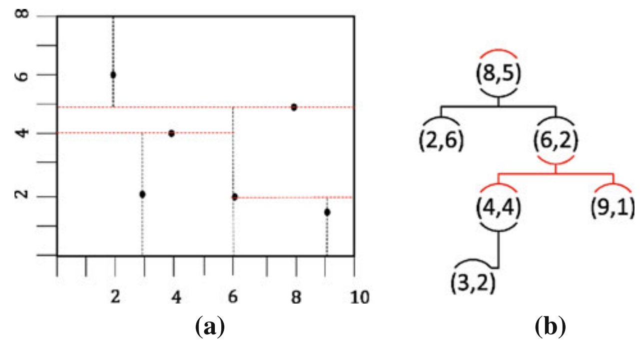


Fig. 7 Example of kd-tree. **a** The space-partitioning. **b** The final kd-tree

3.2 Speed-up using a background mesh: a mixed *kd-tree-elements* approach

A further speed-up in neighbour search can be achieved whenever a background element mesh is available for integration purposes and when the discretization nodes coincide with the nodes of the integration mesh. In fact, each quadrature element, by construction, has Gaussian points inside the element. Therefore, if the vertices of such quadrature element are inside the support of a generic node $\mathbf{x}_I$, then also the correspondent Gaussian points will be inside the support of $\mathbf{x}_I$ (Fig. 8a). However, it may happen that the element is partially contained in the support, i.e. some vertices are inside the support, and others are outside (Fig. 8b). This means that some Gaussian points, very few, will be actually outside the support but still be considered inside the support. Nevertheless, since the value of the kernel in such points is zero, these Gaussian points will be automatically excluded from the computation of the shape functions when constructing the sparse indexing. Alternatively, by exploiting the fast logical indexing, a double check of the distance vs. the support radii could be performed to exclude those Gaussian points,

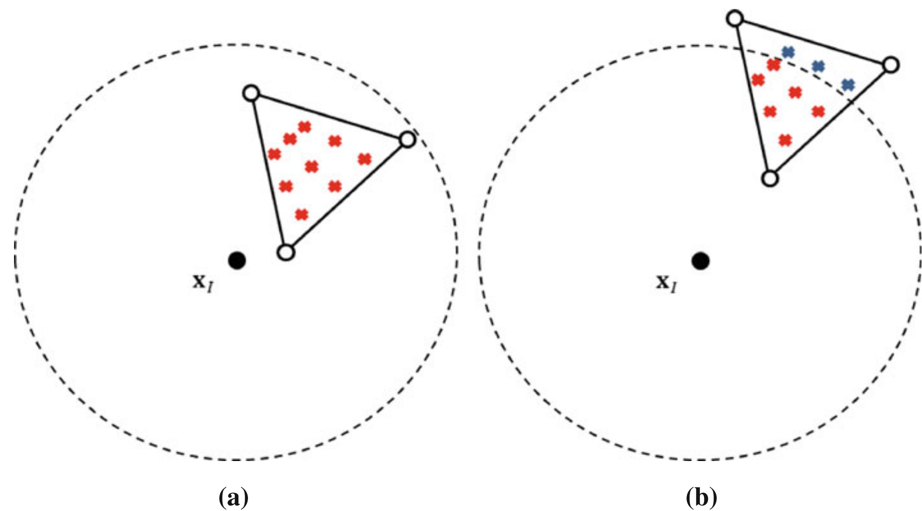


Fig. 8 Neighbour search with a background mesh. **a** Element completely inside the support. **b** Element cut by the support

Table 1 Computational times (expressed in seconds) for range query using different algorithms

Nodes	Gaussian points	Vectorized ES	Unvectorized ES	kd-tree	kd-tree with mesh
1000	95,526	**2.9297**	92.2132	18.0684	11.7255
1680	167,670	**9.066**	227.5757	50.4801	18.9538
2800	307,152	**27.1781**	708.9204	162.5948	37.9888
3150	346,032	65.6346	987.3208	209.0918	**50.8274**
3782	432,540	5940.9861	1588.1562	310.9347	**60.6987**
4320	504,630	20133.9203	1881.3168	415.4892	**68.1305**

Bold values indicate the lowest value
ES Exhaustive search

belonging to elements cut by the support, but outside the support (blue crosses in Fig. 8b).

Knowing the inverse mapping between elements and nodes (i.e. which elements belong to each node), the search can then be performed only inside the set of vertices of the elements, when this set coincides with S_1 (27). Since the number of nodes it is usually less than the number of Gaussian points, this approach notably reduces the computational time. However, it can only be used in combination with elements. When a different type of integration is implemented, for example regular background grid or the discretization nodes do not coincide with the vertices of the mesh, it is necessary to perform the full *kd*-tree neighbour search, which has nevertheless an advantageous $\mathcal{O}(\log N)$ cost for range query. It must be remarked, however, that this approach assumes a certain regularity of the mesh (i.e. the mesh must be *quasi-uniform*). In fact, the approach fails if the mesh is not *covered* [14], meaning that for each element of the mesh, there exist a certain number of nodes whose support overlap with the element. To be more rigorous, the algorithm in [14] should be used to check the mesh coverage. For triangular and tetrahedral mesh and circular supports, this algorithm becomes particularly simple, leading to (little) extra computational costs. However, current FEM mesh generators have advanced *accuracy checks* features for meshes, therefore can provide high quality elements, ensuring safe applicability of the proposed algorithm, particularly if the support size is chosen according to Eq. 15.

3.3 Results

Numerical tests for a three-dimensional model of a bar have been performed to assess the performances of three range searching algorithms: brute-force (or exhaustive search), the kd-tree and the kd-tree with a background mesh. Both number of nodes of set S_1 (27) and number of Gaussian points (set S_2) (28) were varied. The test involved the computation of the three indices $\mathbf{i_g}$, $\mathbf{j_s}$ and $\mathbf{s}$ as showed in Fig. 6, over a tetrahedral background mesh.

For the brute force (or exhaustive search ES) algorithm, two different versions have been used, one *vectorized* and the other *unvectorized*. The vectorized version relies on the highly optimized Matlab built-in function `bsxfun` but creates the full size $n_g \times n_s$ distance matrix $\mathbf{S}$ (Eq. 29). The *unvectorized* version instead, explicitly loops over the nodes, computes the squared distance between the Ith node and the Gaussian points (the second inner loop), compares the squared distances with the square of the support size (instead of their square roots, for efficiency) and extracts the three indices. The unvectorized version, therefore, does not create the full distance matrix. For efficiency, the square roots are calculated only at the end of the loops.

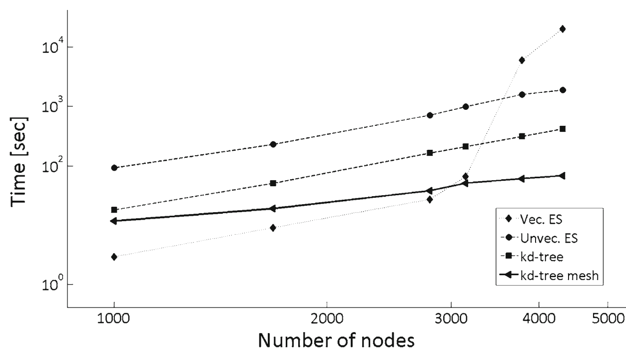


Fig. 9 Computational times for range query using different algorithms. *ES* Exhaustive search. *Axis* are in log-scale

Table 1 shows a comparison between the four approaches. It can be seen that vectorized brute force can be faster than the other methods for *relatively* small data-sets. For large data-sets, however, since it computes the full distance matrix, ES demands much more memory (sometimes causing out-of-memory errors) and, subsequently, larger computational times (the *jump* in Fig. 9). The non-vectorized approach is always slower than the other methods, except when the vectorized brute force fails. Therefore, for small data sets, the vectorized brute force algorithm must be preferred over the *kd-tree*: this is the case, for example, of the Gaussian points at the boundaries used to impose the natural or the essential boundary conditions, or for the bulk points of *small* numerical models. However, with the increase of the number of nodes (and Gaussian points), the brute force becomes prohibitive. For relatively *large* data-sets, for example for the points inside the domain, it is much more convenient to use one of the *kd*-tree algorithms, preferably the one using the background mesh. A robust code should be able to switch between the two methods, for example, when the size of the distance matrix exceeds a predefined number.

3.4 The moment matrix

The key of the reproducing properties of RKPM is the moment matrix $\mathbf{M}(\mathbf{x})$ (Eq. 9, here recalled):

$$\mathbf{M}(\mathbf{x}) = \sum_{I=1}^{n_s} \mathbf{p}\left(\frac{\mathbf{x}-\mathbf{x}_I}{\rho}\right)\mathbf{p}^T\left(\frac{\mathbf{x}-\mathbf{x}_I}{\rho}\right) w\left(\frac{\mathbf{x}-\mathbf{x}_I}{\rho_I}\right), \tag{32}$$

where ρ is the mean of all the dilatation parameters and $\mathbf{p}$ is the polynomial basis. For linear reproducing properties in $2D$ is, for example,

$$\mathbf{p}^T\left(\frac{\mathbf{x}-\mathbf{x}_I}{\rho}\right) = \left[1 \;\; \frac{x-x_I}{\rho} \;\; \frac{y-y_I}{\rho}\right]. \tag{33}$$

Once the connectivity vectors are obtained (31), the values $\mathbf{s}$ (16) can be used to calculate the weight function and its derivatives. Indeed, the connectivity vectors `ig`, `js` and `s` allow the evaluation of the window functions directly on the non-zero values of the compact support functions. With the opportune correction (10), these values will become the shape functions. The role of the connectivity vectors is crucial for the code, since they are computed only once (with an efficient algorithm $\mathscr{O}(\log N)$) and all the subsequent operations can then be carried out *vector-wise* on these indices. This operation is called *loop vectorization*. Moreover, the computations are executed only on non-zero values of the weight functions, avoiding any unnecessary computation.

3.4.1 The window function handle

In `Matlab` it is possible to define an object called *function handle*, which allows a quick evaluation of a function. The function handle is defined using the following command:

$$\texttt{Kernel} = \texttt{@(s)}\;\; 1-6s^2+8s^3-3s^4. \tag{34}$$

where, for example, the 4th order spline is used as *kernel*; however, different kernels can be used by changing this command line and, of course, the corresponding derivative in Eq. 62. In this case $@(s)$ indicate an *anonymous* variable. Such *mute* variable can be replaced with the actual variable `s` (30) and calculate the expression contained in the object `Kernel` over all the values contained in **s** simply by executing the command

$$\texttt{PSI} = \texttt{Kernel(s)}, \tag{35}$$

where the vector `PSI`[6] contains the non-zero values of the weight functions for all the nodes and for all the Gaussian points.

Once obtained the values `PSI`, the following operations are necessary to calculate the entries of the moment matrix:

- the computation of the *scaled coordinates* $\xi_I = \frac{x-x_I}{\rho}$ and $\eta_I \frac{y-y_I}{\rho}$,
- the computation of the polynomial basis (33),
- the computation of the sum of the polynomial basis (32).

3.5 Computation of scaled coordinates

Supposing `GRIDSET` is the coordinate list of the set S_1 and `GGRID` the coordinate list of the set S_2 and `CSI` is the computer implementation of $\xi_I\ \forall I = 1,\ldots,n_s$ and `ETA` the computer variable for $\eta_I\ \forall I = 1,\ldots,n_s$, then the computation of ξ_I and η_I is done in the following way:

$$\texttt{CSI} = (\texttt{GGRID(ig, 1)} - \texttt{GRIDSET(js, 1)})\texttt{./mrho} \tag{36}$$

$$\texttt{ETA} = (\texttt{GGRID(ig, 2)} - \texttt{GRIDSET(js, 2)})\texttt{./mrho} \tag{37}$$

where `mrho` is ρ the average of the dilatation parameters.

[6] `PSI` is used instead of `W` to avoid confusion with the Gaussian weights.

3.5.1 Computation of the polynomial basis

Similarly to the command (34), it is possible to define the polynomial basis as *function handle*. For a more neat programming style, it is opportune to group these functions in an array, or *cell* array.

A *cell array* is similar to a multidimensional array, but with the interesting feature that data are allowed to be not necessarily of the same type.

```
PolyBasis.p{1} = @(x,y) 1          (38)
PolyBasis.p{2} = @(x,y) x          (39)
PolyBasis.p{3} = @(x,y) y          (40)
PolyBasis.dpdx{1} = @(x,y) 0       (41)
PolyBasis.dpdx{2} = @(x,y) 1       (42)
PolyBasis.dpdx{3} = @(x,y) 0       (43)
PolyBasis.dpdy{1} = @(x,y) 0       (44)
PolyBasis.dpdy{2} = @(x,y) 0       (45)
PolyBasis.dpdy{3} = @(x,y) 1       (46)
```

Let us define a function handle of the type

```
P_handle = @(P) {P(CSI,ETA)}       (47)
```

where `CSI` is (36) and `ETA` is (37). In this way, by simply calling

```
P = cellfun(P_handle,PolyBasis.p(1:3)).'
                                   (48)
```

the values of the polynomial basis are readily calculated. Moreover, with the call

```
DPDX=cellfun(P_handle,PolyBasis.dpdx(1:3)).'
                                   (49)
DPDY=cellfun(P_handle,PolyBasis.dpdy(1:3)).'
                                   (50)
```

the derivatives of the polynomial basis can also be automatically calculated, without the need of defining dedicated function handles for the derivatives.

3.5.2 Computation of the sum of the polynomials

According to Eq. 32, in order to obtain the entries of the moment matrix, the sum of the polynomials (48) must be performed over the nodes. After (48), `P` is a 3×1 cell array. Every element of the cell contains n_{nz} values, the same number as the variables `ig`, `js` and `s`.

The sum is carried out in three steps:

1. the product

$$\mathbf{p}_i\left(\frac{\mathbf{x}-\mathbf{x}_I}{\rho_I}\right)\mathbf{p}_j\left(\frac{\mathbf{x}-\mathbf{x}_I}{\rho_I}\right)w\left(\frac{\mathbf{x}-\mathbf{x}_I}{\rho_I}\right)\quad i,j=1,\ldots,n_p, \tag{51}$$

   where n_p is the number of the polynomials included in the basis;
2. reshape of the matrix product (51);
3. sum over the columns (the second dimension) of the reshaped matrix.

The matrix product (51) originates n_p^2 vectors of size $n_{nz} \times 1$. However, since the moment matrix is symmetric, not all the products need to be stored, but only $n_p(n_p+1)/2$. Therefore in Eq. 51 the products to calculate are only the ones $i, j = 1, \ldots, n_p$ with $i \leq j$.

These *indexes* i, j for a symmetric matrix $n_p \times n_p$ can be grouped in two arrays of length $n_p(n_p+1)/2$. For example, if $n_p = 3$

```
i = [1 1 1 2 2 3]                  (52)
j = [1 2 3 2 3 3]                  (53)
```

Indexes in (52) and (53) allow the definition of a *function handle* for the products in Eq. 51

```
M_handle = @(i,j) {P{i}.*P{j}.*PSI}    (54)
```

In fact, vectors $\mathtt{P}_i$ and `PSI` are the same size and thus can be multiplied in an *element-wise* manner. The call

```
M = arrayfun(M_handle,i,j)         (55)
```

originates a cell array of size $n_p(n_p+1)/2$, where each element contains the product (51).

The next step is to reshape each element of the cell array `M` from a n_{nz} vector to a matrix $n_g \times n_s$. Luckily, the command `sparse` in `Matlab` allows this transformation without padding with zeros.

The command

```
S = sparse(i,j,s,m,n)              (56)
```

uses vectors `i`, `j`, and `s` to generate an $m \times n$ sparse matrix such that

$$S(i(k), j(k)) = s(k) \quad k = 1, \ldots, n_{nz} \tag{57}$$

if vectors `i`, `j`, and `s` are all the same length.

Additionally, any elements of `s` that are zero are ignored, along with the corresponding values of `i` and `j`. Any elements of `s` that have duplicate values of `i` and `j` are added together.

Therefore, the following command can be defined

```
spar_handle=@(C) sum(sparse(ig,js,C,ng,ns),2)
```
(58)

where ng is n_g and ns is n_s.

Command (58) incorporates 2 commands, the first one is sparse, which reshape the *anonymous* matrix C and the second one is the sum over the nodes, where 2 stands for the second dimension of the matrix, i.e. the columns. It can be seen that with the introduction of the connectivity vectors ig and js, it is no longer necessary to search for nodes at each Gaussian point for the purpose of the sum in the moment matrix. These operations can be indeed automatized once the connectivity has been defined at the beginning of the calculation.

By calling

```
M=cellfun(spar_handle,M)
```
(59)

the moment matrix **M** is finally calculated. The result of (59) is a cell array of $n_p(n_p+1)/2$ elements, each array containing n_g elements, i.e. as many as the number of the Gaussian points. By inverting **M**, it is possible to get the corrective terms for the kernels (the weight functions) that restore the reproducing properties.

3.5.3 Derivatives of the moment matrix

For completeness are here reported the commands necessary to calculate the derivatives of the moment matrix. Firstly, it is necessary to calculate the derivatives of the kernel, as in Eq. 20.

$$\frac{\partial w}{\partial x}=\frac{\partial w}{\partial s}\frac{\partial s}{\partial \xi_I}\frac{\partial \xi_I}{\partial x}=\frac{\partial w}{\partial s}\frac{\partial s}{\partial \xi_I}\frac{1}{\rho_I} \tag{60}$$

$$\frac{\partial w}{\partial y}=\frac{\partial w}{\partial s}\frac{\partial s}{\partial \eta_I}\frac{\partial \xi_I}{\partial x}=\frac{\partial w}{\partial s}\frac{\partial s}{\partial \eta_I}\frac{1}{\rho_I} \tag{61}$$

The first step is calculating $\frac{\partial w}{\partial s}$ by defining the following function handle

```
dKernel=@(s) -12s(s-1)^2
```
(62)

and then executing the command

```
dPSI=dKernel(S).
```
(63)

The derivatives $\frac{\partial s}{\partial \xi_I}$ and $\frac{\partial s}{\partial \eta_I}$ are calculated as follows

```
dDcsi=((GGRID(ig,1)-GRIDSET(js,1))./rho(js))./S
```
(64)

```
dDcsi(S==0)=1
```
(65)

```
dDeta=((GGRID(ig,2)-GRIDSET(js,2))./rho(js))./S
```
(66)

```
dDeta(S==0)=1
```
(67)

where Eqs. 65 and 67 eliminate the indeterminate form. Derivatives of the kernel functions are

```
dPSIcsi=dPSI.*dDcsi
```
(68)

```
dPSIeta=dPSI.*dDeta
```
(69)

The handles for the derivatives of the moment matrix are

```
DMx_handle=@(i,j) {(DPDX{i}.*P{j}
  +P{i}.*DPDX{j}).*PSI
  + P{i}.*P{j}.*dPSIcsi}
```
(70)

```
DMy_handle=@(i,j) {(DPDY{i}.*P{j}
  +P{i}.*DPDY{j}).*PSI
  + P{i}.*P{j}.*dPSIeta}
```
(71)

and the operations of product, reshape and sum are executed by calling these commands

```
DMx=arrayfun(DMx_handle,i,j)
```
(72)

```
DMx=cellfun(spar_handle,DMx)
```
(73)

```
DMy=arrayfun(DMy_handle,i,j)
```
(74)

```
DMy=cellfun(spar_handle,DMy)
```
(75)

3.5.4 Explicit inverse of the moment matrix

In Sect. 3.4 the construction if the moment matrix for linear reproducing property was illustrated.

If a linear reproducing property is desired, the moment matrix can be inverted directly by symbolic manipulation, as in [10,47]. This can still be done in 3D, when the moment matrix is size 4×4.

However, if higher order reproducing properties are sought, this approach cannot be used, since the symbolic inversion of a matrix of size larger than 5×5 is unstable or even impossible to achieve practically. In Sect. 3.6.1 an alternative approach is proposed. It will be shown that the explicit inverse of the moment matrix for linear reproducing property is indeed the starting point to calculate the inverse of the moment matrix with higher reproducing capabilities.

The inversion of the moment matrix for linear reproducing properties can be done following the Cramer's rule

$$\det(\mathbf{M})=\left(\mathbf{M}_{2,2}\mathbf{M}_{3,3}-\mathbf{M}_{2,3}^2\right)\mathbf{M}_{1,1}-\mathbf{M}_{1,3}^2\mathbf{M}_{2,2} -\mathbf{M}_{1,2}^2\mathbf{M}_{3,3}+2\mathbf{M}_{1,2}\mathbf{M}_{1,3}\mathbf{M}_{2,3} \tag{76}$$

$$\mathbf{M}_{1,1}^{-1}=\frac{\mathbf{M}_{2,2}\mathbf{M}_{3,3}-\mathbf{M}_{2,3}^2}{\det(\mathbf{M})} \tag{77}$$

$$\mathbf{M}_{1,2}^{-1}=-\frac{\mathbf{M}_{1,2}\mathbf{M}_{3,3}-\mathbf{M}_{1,3}\mathbf{M}_{2,3}}{\det(\mathbf{M})} \tag{78}$$

$$\mathbf{M}_{1,3}^{-1}=\frac{\mathbf{M}_{1,2}\mathbf{M}_{2,3}-\mathbf{M}_{1,3}\mathbf{M}_{2,2}}{\det(\mathbf{M})} \tag{79}$$

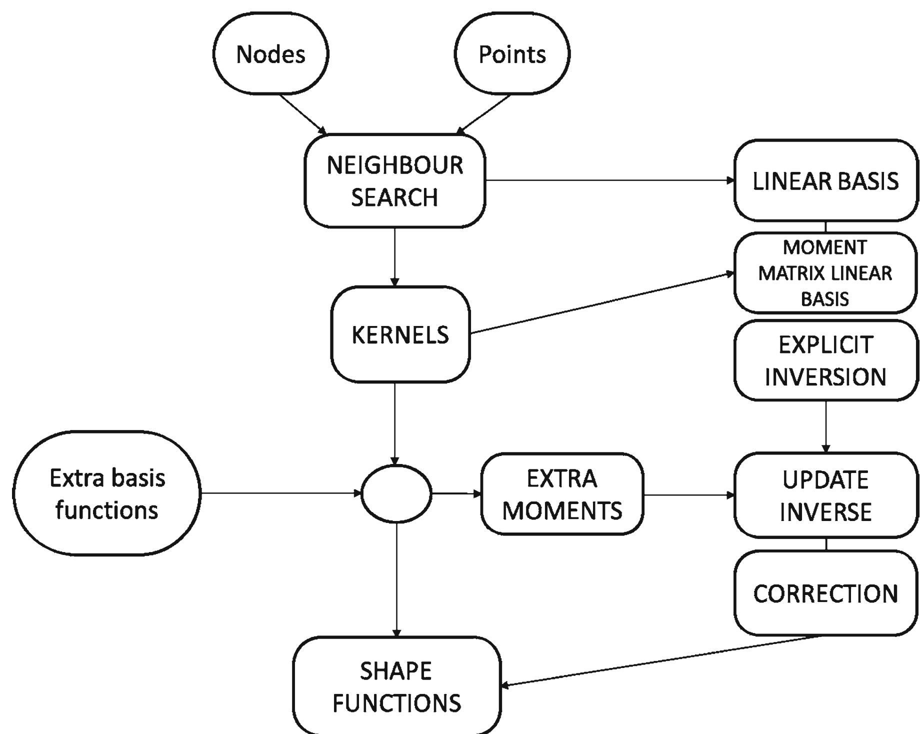


Fig. 10 Flowchart of the proposed algorithm in meshfree methods

$$\mathbf{M}^{-1}_{2,2} = \frac{\mathbf{M}_{1,1}\mathbf{M}_{3,3} - \mathbf{M}^2_{1,3}}{\det(\mathbf{M})} \tag{80}$$

$$\mathbf{M}^{-1}_{2,3} = -\frac{\mathbf{M}_{1,1}\mathbf{M}_{2,3} - \mathbf{M}_{1,2}\mathbf{M}_{1,3}}{\det(\mathbf{M})} \tag{81}$$

$$\mathbf{M}^{-1}_{3,3} = \frac{\mathbf{M}_{1,1}\mathbf{M}_{2,2} - \mathbf{M}^2_{1,2}}{\det(\mathbf{M})} \tag{82}$$

Derivatives of the inverse of the moment matrix can be calculated with Eq. 14.

3.6 Inversion of the moment matrix through partitioning

In the previous section, explicit inversion formulas for the moment matrix were illustrated, for the case of linear reproducibility. This section is focused on the following two questions:

1. how to calculate the shape functions if a higher order reproducibility is sought?
2. how to calculate the shape functions if other points are inserted?

The first question is usually known as *p-adaptivity*, whilst the second is known *h-adaptivity*. Together, they are known as *hp*-adaptivity. The *hp*-adaptivity consists in a global/local refinement of the approximation through raising the order of the polynomials in the approximating function and/or adding nodes to the discretization, with the aim of reducing the local error or capturing high gradient zones. Local *h*-refinement consists in adding nodes to limited zones of the domain, usually detected when a *measure* of the error exceeds a predefined threshold, while global *h-refinement* means refining the discretization in the whole domain. Local *p*-refinement is done by enriching the approximation with the introduction of extra-unknowns and they not need to be necessarily polynomials. Global *p*-refinement is instead obtained in meshfree methods by adding extra-functions to the polynomial basis in Eq. 2. This section will deal on global *p*-refinement.

With the aim of reducing the error, the *error* must be calculated, or, more appropriately, *estimated a posteriori*.

Detailed information on adaptivity in meshless methods can be found in [31,43]. Once these zones are individuated, then nodes or polynomials must be practically inserted into the approximation. In meshfree methods, this task is simpler than FE, since nodes can be inserted at any time in the calculation without affecting the mesh connectivity and higher order polynomials can be inserted in the basis vector $\mathbf{p}$ without additional unknowns.

This section explains how practically insert extra polynomials or nodes to the shape functions without carrying out expensive operations.

The *classic* algorithm in Fig. 3 will be modified into the one in Fig. 10.

3.6.1 Fast inversion of the moment matrix and p-adaptivity

In this section is proposed a method to efficiently invert the moment matrix whenever the order of reproducibility

is greater than one. As a result, it can be proficiently used for global p-adaptivity. A similar approach based on the imposing consistency constraints can be found in [10]. The reproducing properties are imposed through Lagrangian multipliers. The resulting method does not make use of a moment matrix, but iteratively update the weights, nonetheless leading to similar formulas. The proposed method does not employ Lagrangian multipliers but simply a partitioning of the moment matrix and it is based on the updating of its inverse. The approach described calculates *explicitly* the inverse of the moment matrix without recurring to numerical routines (Gauss elimination, LU factorization) for point-wise inversion. This is one of the most important results of this paper, since it *vectorizes* the code completely, without using any *for* loops.

Let

$$w_I = w\left(\frac{\mathbf{x}-\mathbf{x}_I}{\rho_I}\right) \quad I = 1,\ldots,N \tag{83}$$

and

$$\mathbf{p}_I^{(k)} = \mathbf{p}^{(k)}\left(\frac{\mathbf{x}-\mathbf{x}_I}{\rho}\right), \tag{84}$$

where ρ is

$$\rho = \frac{1}{N}\sum_{I=1}^{N}\rho_I. \tag{85}$$

The superscript k in Eq. 84 is the number of polynomial terms included in the basis functions $\mathbf{p}$. In fact, shape functions with such basis will be indicated with superscript k as well, i.e.

$$\phi_I^{(k)}(\mathbf{x}) = \mathbf{p}^{(k)}(0)^T\mathbf{M}_k^{-1}(\mathbf{x})\mathbf{p}_I^{(k)}w_I(\mathbf{x}), \tag{86}$$

$$\mathbf{M}_k(\mathbf{x}) = \sum_{I=1}^{N}\mathbf{p}_I^{(k)}\mathbf{p}_I^{(k)T}w_I, \tag{87}$$

where the *scaled* form has been used to avoid problems on the condition number for $\mathbf{M}_k(\mathbf{x})$. The final goal is to obtain the shape functions when an additional term is included in the basis function, without re-calculating the moment matrix and, most importantly, without performing the costly *point-by-point* matrix inversion.

This means

$$\phi_I^{(k+1)}(\mathbf{x}) = \mathbf{p}^{(k+1)}(0)^T\mathbf{M}_{k+1}^{-1}(\mathbf{x})\mathbf{p}_I^{(k+1)}w_I(\mathbf{x}). \tag{88}$$

In order to do so, the *corrective* term needs to be updated

$$C_I^{(k+1)}(\mathbf{x}) = \mathbf{p}^{(k+1)}(0)^T\mathbf{M}_{k+1}^{-1}(\mathbf{x})\mathbf{p}_I^{(k+1)}, \tag{89}$$

which in turn requires the update of the basis

$$\mathbf{p}_I^{(k+1)T} = \begin{bmatrix}\mathbf{p}_I^{(k)T} & p_I^{(e)}\end{bmatrix} \tag{90}$$

and the update of the inverse of the moment matrix

$$\mathbf{M}_{k+1}(\mathbf{x}) = \sum_{I=1}^{N}\mathbf{p}_I^{(k+1)}\mathbf{p}_I^{(k+1)T}w_I = \sum_{I=1}^{N}\begin{bmatrix}\mathbf{p}_I^{(k)}\\ p_I^{(e)}\end{bmatrix}\begin{bmatrix}\mathbf{p}_I^{(k)T} & p_I^{(e)}\end{bmatrix}w_I = \begin{bmatrix}\mathbf{M}_k(\mathbf{x}) & \mathbf{b}\\ \mathbf{b}^T & c\end{bmatrix}, \tag{91}$$

where

$$\mathbf{b}(\mathbf{x}) = \sum_{I=1}^{N}\mathbf{p}^{(k)}p^{(e)}w_I \tag{92}$$

$$c(\mathbf{x}) = \sum_{I=1}^{N}p^{(e)2}w_I. \tag{93}$$

Inversion of matrix (91) can be directly obtained from the entries of $\mathbf{M}_{k+1}^{-1}(\mathbf{x})$ without performing a computationally expensive *point by point* inversion. In fact, using Boltz's formula, $\mathbf{M}_{k+1}(\mathbf{x})$ can be inverted *block-wise* by using the following analytical inversion formula

$$\mathbf{M}_{k+1}^{-1}(\mathbf{x}) = \begin{bmatrix}\mathbf{M}_k^{-1}(\mathbf{x}) & 0\\ 0 & 0\end{bmatrix} + \frac{1}{q}\begin{bmatrix}(\mathbf{M}_k^{-1}\mathbf{b})(\mathbf{M}_k^{-1}\mathbf{b})^T & -\mathbf{M}_k^{-1}\mathbf{b}\\ -(\mathbf{M}_k^{-1}\mathbf{b})^T & 1\end{bmatrix}, \tag{94}$$

where

$$q(\mathbf{x}) = c - \mathbf{b}^T(\mathbf{M}_k^{-1}\mathbf{b}). \tag{95}$$

Equation 94 is tremendously useful since one need to perform only *element-wise* products and divisions that are particularly fast in `Matlab`.

In fact, without this formula, one has to *loop* over the far numerous Gaussian points and invert the moment matrix with a numerical routine. This is one of the major drawbacks of RKPM and MLS, since an accurate integration of the weak form of the PDE might necessitate a high order Gaussian quadrature scheme and therefore the number of Gaussian points notably grows. Instead, using Eq. 94, assuming that the entries $\mathbf{M}_k^{-1}(\mathbf{x})$ are already available, one needs only to calculate the moments $\mathbf{b}$ and c and perform only *element-wise* operations to update the moment matrix.

Boltz' formula allows a *vectorized* `Matlab` code and effectively speeds up the calculation of the shape functions.

It must be pointed out though, that Eq. 94 shows how to *update* the inverse of a matrix when an additional column (and a row) is introduced when $\mathbf{M}_k^{-1}(\mathbf{x})$ is already known. Therefore to fully exploit the advantages of the block-inversion, the entries of $\mathbf{M}_k^{-1}(\mathbf{x})$ must be obtained beforehand. Nevertheless, one of the greatest advantages of Eq. 94 is that can be applied *iteratively*, in the sense that can be applied several times until the desired order of reproduction is achieved.

For example, a linear reproduction property is at least required for the convergence of the weak form. Therefore

shape functions at least must satisfy linear reproduction conditions, which lead to a 3×3 moment matrix in two dimensions and a 4×4 moment matrix in three-dimensions. These sizes of matrices are easily invertible analytically, for example with Cramer's rule for the two-dimensional case and with symbolic inversion for 4×4, that again do not require any numerical routine for matrix inversion and can be obtained with element-wise matrix operations.

It must be remarked that symbolic inversion is *unstable* (and sometimes impossible for some symbolic softwares) for 5×5 and *impossible* to obtain for matrices of higher size. Nonetheless linear reproduction property can be the *starting point* of the inversion of a bigger size matrix. In fact once $\mathbf{M}_3^{-1}(\mathbf{x})$ is calculated for the two-dimensional case, with (94) it is possible to obtain $\mathbf{M}_4^{-1}(\mathbf{x})$. Once obtained $\mathbf{M}_4^{-1}(\mathbf{x})$ one proceeds to $\mathbf{M}_5^{-1}(\mathbf{x})$ and so on until the sought order of reproduction is achieved.

3.6.2 First order derivatives

In the assembly of the stiffness matrix of a weak form, at least first order derivatives of the shape functions are required, hence a fast computation of the first order derivatives of Eq. 88 can accelerate the assembly process.

$$\frac{\partial \phi_I}{\partial x}^{(k+1)} = \frac{\partial C_I}{\partial x}^{(k+1)} w_I + C_I^{(k+1)} \frac{\partial w_I}{\partial x}, \tag{96}$$

where

$$\frac{\partial C_I}{\partial x}^{(k+1)} = \mathbf{p}^{(k)T}(0) \left(\frac{\partial \mathbf{M}_{k+1}^{-1}(\mathbf{x})}{\partial \mathbf{x}} \mathbf{p}_I^{(k+1)} + \mathbf{M}_{k+1}^{-1}(\mathbf{x}) \frac{\partial \mathbf{p}_I^{(k+1)}}{\partial \mathbf{x}} \right). \tag{97}$$

Derivatives of $\mathbf{M}_{k+1}^{-1}(\mathbf{x})$ can be readily obtained by derivation of Eq. 94

$$\begin{aligned} \frac{\partial \mathbf{M}_{k+1}^{-1}(\mathbf{x})}{\partial x} &= \begin{bmatrix} \frac{\partial \mathbf{M}_k^{-1}(\mathbf{x})}{\partial x} & 0 \\ 0 & 0 \end{bmatrix} \\ &- \frac{1}{q^2} \frac{\partial q}{\partial x} \begin{bmatrix} (\mathbf{M}_k^{-1}\mathbf{b})(\mathbf{M}_k^{-1}\mathbf{b})^T & -\mathbf{M}_k^{-1}\mathbf{b} \\ -(\mathbf{M}_k^{-1}\mathbf{b})^T & 1 \end{bmatrix} \\ &+ \frac{1}{q} \begin{bmatrix} \frac{\partial(\mathbf{M}_k^{-1}\mathbf{b})}{\partial x}(\mathbf{M}_k^{-1}\mathbf{b})^T + (\mathbf{M}_k^{-1}\mathbf{b})\frac{\partial(\mathbf{M}_k^{-1}\mathbf{b})^T}{\partial x} & -\frac{\partial(\mathbf{M}_k^{-1}\mathbf{b})}{\partial x} \\ -\frac{\partial(\mathbf{M}_k^{-1}\mathbf{b})^T}{\partial x} & 0 \end{bmatrix}, \end{aligned} \tag{98}$$

where

$$\frac{\partial \mathbf{M}_k^{-1}(\mathbf{x})}{\partial \mathbf{x}} = -\mathbf{M}_k^{-1}(\mathbf{x}) \frac{\partial \mathbf{M}_k(\mathbf{x})}{\partial x} \mathbf{M}_k^{-1}(\mathbf{x}), \tag{99}$$

$$\frac{\partial(\mathbf{M}_k^{-1}\mathbf{b})}{\partial x} = \frac{\partial \mathbf{M}_k^{-1}(\mathbf{x})}{\partial x}\mathbf{b} + \mathbf{M}_k^{-1}(\mathbf{x})\frac{\partial \mathbf{b}}{\partial x}, \tag{100}$$

$$\begin{aligned} \frac{\partial \mathbf{M}_k(\mathbf{x})}{\partial x} &= \sum_{I=1}^{N} \left(\frac{\partial \mathbf{p}^{(k)}}{\partial x} \mathbf{p}^{(k)T} + \mathbf{p}^{(k)} \frac{\partial \mathbf{p}^{(k)T}}{\partial x} \right) w_I \\ &\quad + \mathbf{p}^{(k)} \mathbf{p}^{(k)T} \frac{\partial w_I}{\partial x}, \end{aligned} \tag{101}$$

$$\frac{\partial q}{\partial x} = \frac{\partial c}{\partial x} - \frac{\partial \mathbf{b}^T}{\partial x}(\mathbf{M}_k^{-1}\mathbf{b}) + \mathbf{b}^T \frac{\partial(\mathbf{M}_k^{-1}\mathbf{b})}{\partial x}, \tag{102}$$

$$\frac{\partial c}{\partial x} = \sum_{I=1}^{N} 2p^{(e)} \frac{\partial p^{(e)}}{\partial x} w_I + p^{(e)2} \frac{\partial w_I}{\partial x}, \tag{103}$$

$$\begin{aligned} \frac{\partial \mathbf{b}}{\partial x} &= \sum_{I=1}^{N} \left(\frac{\partial \mathbf{p}^{(k)}}{\partial x} p^{(e)} + \mathbf{p}^{(k)} \frac{\partial p^{(e)}}{\partial x} \right) w_I \\ &\quad + \mathbf{p}^{(k)} p^{(e)} \frac{\partial w_I}{\partial x}. \end{aligned} \tag{104}$$

3.6.3 Fast inversion of the moment matrix and h-adaptivity

In this section a method to answer to the second of the questions in Sect. 3.6 is proposed.

Let us indicate with subscript N the moment matrix with N nodes

$$\mathbf{M}_N(\mathbf{x}) = \sum_{I=1}^{N} \mathbf{p}_I \mathbf{p}_I^T w_I. \tag{105}$$

If an additional node is introduced, the moment matrix inverse needs to be updated.

$$\begin{aligned} \mathbf{M}_{N+1}(\mathbf{x}) &= \sum_{I=1}^{N+1} \mathbf{p}_I \mathbf{p}_I^T w_I = \sum_{I=1}^{N} \mathbf{p}_I \mathbf{p}_I^T w_I + \mathbf{p}_{N+1} \mathbf{p}_{N+1}^T w_{N+1} \\ &= \mathbf{M}_N(\mathbf{x}) + \mathbf{p}_{N+1} \mathbf{p}_{N+1}^T w_{N+1}. \end{aligned} \tag{106}$$

Therefore it is simply a *rank-1* update:

$$\begin{aligned} \mathbf{M}_{N+1}^{-1}(\mathbf{x}) &= \mathbf{M}_N^{-1}(\mathbf{x}) \\ &\quad - \frac{\left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)\left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)^T}{r} w_{N+1}, \end{aligned} \tag{107}$$

$$r = 1 + \mathbf{p}_{N+1}^T \left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right) w_{N+1}, \tag{108}$$

therefore

$$\phi_I(\mathbf{x}) = \mathbf{p}^T(0)\mathbf{M}_{N+1}^{-1}(\mathbf{x})\mathbf{p}_I w_I(\mathbf{x}) \quad I = 1, \ldots, N+1. \tag{109}$$

Derivatives are given by

$$\begin{aligned} \frac{\partial \phi_I}{\partial x} &= \mathbf{p}^T(0) \left[\left(\frac{\partial \mathbf{M}_{N+1}^{-1}(\mathbf{x})}{\partial x} \mathbf{p}_I + \mathbf{M}_{N+1}^{-1}(\mathbf{x}) \frac{\partial \mathbf{p}_I}{\partial x} \right) w_I(\mathbf{x}) \right. \\ &\quad \left. + \mathbf{M}_{N+1}^{-1}(\mathbf{x})\mathbf{p}_I \frac{\partial w_I}{\partial x} \right], \end{aligned} \tag{110}$$

$$\frac{\partial \mathbf{M}_{N+1}^{-1}(\mathbf{x})}{\partial x} = \frac{\partial \mathbf{M}_{N}^{-1}(\mathbf{x})}{\partial x} - \frac{1}{r^2}\left[\left(\frac{\partial\left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)}{\partial x}\left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)^T + \right.\right.$$
$$\left. + \left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)\frac{\partial\left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)^T}{\partial x}\right) r$$
$$\left. - \left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)\left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)^T \frac{\partial r}{\partial x}\right] w_{N+1}$$
$$- \frac{\left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)\left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)^T}{r}\frac{\partial w_{N+1}}{\partial x}, \tag{111}$$

where

$$\frac{\partial\left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)}{\partial x} = \frac{\partial \mathbf{M}_N^{-1}(\mathbf{x})}{\partial x}\mathbf{p}_{N+1} + \mathbf{M}_N^{-1}(\mathbf{x})\frac{\partial \mathbf{p}_{N+1}}{\partial x}, \tag{112}$$

$$\frac{\partial r}{\partial x} = \left[\frac{\partial \mathbf{p}_{N+1}^T}{\partial x}\left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right) + \mathbf{p}_{N+1}^T\frac{\partial\left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)}{\partial x}\right] w_{N+1} + \mathbf{p}_{N+1}^T\left(\mathbf{M}_N^{-1}\mathbf{p}_{N+1}\right)\frac{\partial w_{N+1}}{\partial x}. \tag{113}$$

Finally, it should be remarked that the approach in formula (106) was firstly noted in [46], without explicitly suggesting the update of the *inverse* of the moment matrix.

3.7 The kernel corrective term

The term in Eq. 11 (here recalled for clarity) is called corrective because it allows to calculate the shape functions (114) from the kernel functions, restoring the reproducing properties.

$$\phi_I(\mathbf{x}) = C_I(\mathbf{x}) w\left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right), \tag{114}$$

$$C_I(\mathbf{x}) = \mathbf{p}^T(\mathbf{0})\mathbf{M}(\mathbf{x})^{-1}\mathbf{p}^T\left(\frac{\mathbf{x}_I - \mathbf{x}}{\rho}\right). \tag{115}$$

The values of the weight functions are computed through the commands (35) and stored in an array `PSI` of length n_{nz} and so do the values of `P`, stored in a cell array. The problem is the entries of the inverse of the moment matrix $\mathbf{M}^{-1}$, that are stored in a cell array `Minv` and every element of the cell is an array of length n_g. Therefore, a *naive* approach would loop over the basis function, then double loop over the nodes and over the evaluation points, calculating the correction and then applying it to the weight functions. Instead, the corrective term can be quickly calculated using the connectivity vector `ig`.

```
C = 0;
if nargout>2
    dCx = 0;
    dCy = 0;
end

for i=1:np
    for j=1:np
       if P0{i}~=0
            if i<=j
                C = C + P0{i}.*Minv{i,j}(ig).*P{j};
                if nargout>2
                    dCx = dCx + P0{i}.*DMinvx{i,j}(ig).*P{j} +...
                    + P0{i}.*Minv{i,j}(ig).*DPDX{j};
                    dCy = dCy + P0{i}.*DMinvy{i,j}(ig).*P{j} +...
                    P0{i}.*Minv{i,j}(ig).*DPDY{j};
                end
            else
                C = C + P0{i}.*Minv{j,i}(ig).*P{j};
                if nargout>2
                    dCx = dCx + P0{i}.*DMinvx{j,i}(ig).*P{j} +...
                    + P0{i}.*Minv{j,i}(ig).*DPDX{j};
                    dCy = dCy + P0{i}.*DMinvy{j,i}(ig).*P{j} +...
                    + P0{i}.*Minv{j,i}(ig).*DPDY{j};
                end
            end
       end
    end
end
```

where `ig` act as a *index* in the command `Minv{i,j}(ig)`. The variable `nargout` is the number of output arguments. If more than two arguments are requested, derivatives are calculated.

This index `ig` points to `Minv{i,j}(:)`, in the same order `PSI` and `P` are stored, allowing the generic entry of the moment matrix to become an array of length n_{nz} and multiply `PSI` and `P` in an *element-wise* manner. The same approach can be applied also to the derivatives of the corrective term, i.e. `dCx` and `dCy`. Using the same logic, the following handle can be defined

`asb_handle = @(f,g) f.*g(ig)` (116)

that will be useful for the stiffness matrix assembly in the next section. Finally, the shape functions are calculated as follows

```
PHI = C.*PSI;
if nargout>2
  DPHIx = (dCx.*PSI + C.*dPSIcsi)/mrho;
  DPHIy = (dCy.*PSI + C.*dPSIeta)/mrho;
end
```

At this stage, though, the values are stored in arrays of size n_{nz}. To reshape the arrays, the following handle can be defined

`sp_handle = @(f)sparse(ig,js,f,ng,ns).` (117)

The object (117) transforms a sparse array f defined by indexes `ig` and `js` into a sparse matrix `ng` $\times$ `ns`.

The inverse operation can be performed by the following command

```
desp_handle=@(f)f(sub2ind(size(f),ig,js)).
```
(118)

The final step for constructing the shape functions is

```
PHI = sp_handle(PHI);
if nargout>2
     DPHIx = sp_handle(DPHIx);
     DPHIy = sp_handle(DPHIy);
end
```

3.8 Assembly of the stiffness matrix

In this section the assembly of the stiffness matrix is explained in detail for the two-dimensional case, however the principle applies also to the three-dimensional structures. The important result of this section is that no loops are necessary to calculate the stiffness matrix.

Let us assume an isotropic linear elastic body Ω, (Fig. 11) under the assumption of small displacements and small gradient of displacements. The equilibrium equations and the boundary conditions are, in absence of body forces:

$$\nabla \cdot \sigma = 0 \quad \mathbf{x} \in \Omega \tag{119}$$

$$\sigma \cdot \mathbf{n} = \bar{\mathbf{t}} = \lambda \mathbf{t}_0 \quad \mathbf{x} \in \Gamma_t \tag{120}$$

$$\mathbf{u} = \bar{\mathbf{u}} \quad \mathbf{x} \in \Gamma_u \tag{121}$$

where σ is the Cauchy stress tensor, $\mathbf{n}$ is the normal unity vector of the boundary Γ_t where the traction $\lambda \mathbf{t}_0$ is prescribed and Γ_u is the boundary where the displacement $\bar{\mathbf{u}}$ is prescribed. The traction $\mathbf{t}_0$ is a unitary reference traction field with magnitude λ.

Using the displacement $\mathbf{u}$ as a test function for Eqs. 119, 120 and 121, the variational form can be written as

$$\int_{\Omega} \delta\epsilon^T \sigma \mathrm{d}\Omega - \int_{\Gamma_t} \delta\mathbf{u}^T \bar{\mathbf{t}} \mathrm{d}\Gamma_t + \alpha \int_{\Gamma_u} \delta\left(\mathbf{u} - \bar{\mathbf{u}}\right)^T \left(\mathbf{u} - \bar{\mathbf{u}}\right) \mathrm{d}\Gamma_u = 0 \tag{122}$$

where α is a penalty parameter (usually a large number) used to enforce the essential boundary conditions.

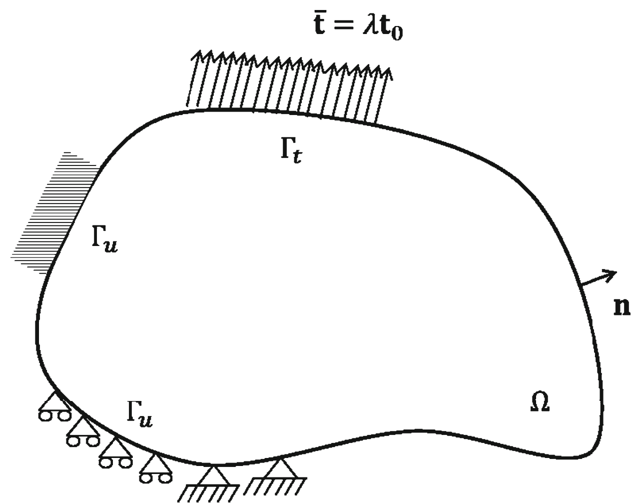


Fig. 11 Description of the body Ω

The infinitesimal strain is defined as

$$\epsilon = \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \gamma_{xy} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial u}{\partial x} \\ \dfrac{\partial v}{\partial y} \\ \dfrac{\partial u}{\partial y} + \dfrac{\partial v}{\partial x} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial}{\partial x} & \\ & \dfrac{\partial}{\partial y} \\ \dfrac{\partial}{\partial y} & \dfrac{\partial}{\partial x} \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \mathscr{L}\mathbf{u}, \tag{123}$$

where $\mathscr{L}$ is the infinitesimal strain differential operator and $\mathbf{u}$ is the displacement vector.

The linear elastic stress–strain relationship is defined through the Generalized Hooke's law (with the *Voigt notation*[7])

$$\sigma = \mathbf{C}\epsilon = \begin{bmatrix} C_{11} & C_{12} & \\ C_{12} & C_{22} & \\ & & C_{33} \end{bmatrix} \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ 2\gamma_{xy} \end{bmatrix}, \tag{124}$$

where $\mathbf{C}$ is called the *stiffness tensor of the elastic moduli* and for two-dimensional orthotropic materials in *plane stress* tensional state is defined as

$$\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \\ C_{12} & C_{22} & \\ & & C_{33} \end{bmatrix} = \begin{bmatrix} E_{11}/(1-\nu_{12}\nu_{21}) & \nu_{12}E_{22}/((1-\nu_{12}\nu_{21}) & \\ \nu_{12}E_{22}/((1-\nu_{12}\nu_{21}) & E_{22}/(1-\nu_{12}\nu_{21}) & \\ & & G_{12} \end{bmatrix}, \tag{125}$$

[7] The Voigt notation is the standard mapping for tensor indexes.

where E_{ii} is the Young's modulus along axis i, G_{ij} is the shear modulus in direction j on the plane whose normal is in direction i, ν_{ij} is the Poisson's ratio that corresponds to a contraction in direction j when an extension is applied in direction i.

If *plane strain* is desired, then the following changes are necessary

$$E_{11} \leftarrow \frac{E_{11}}{1-\nu_{12}^2}, \tag{126}$$

$$E_{22} \leftarrow \frac{E_{22}}{1-\nu_{12}^2}, \tag{127}$$

$$G_{12} \leftarrow \frac{G_{12}}{1-\nu_{12}^2}, \tag{128}$$

$$\nu_{12} \leftarrow \frac{\nu_{12}}{1-\nu_{12}^2}, \tag{129}$$

$$\nu_{21} \leftarrow \frac{\nu_{21}}{1-\nu_{12}^2}. \tag{130}$$

The displacement vector $\mathbf{u}$ is approximated with the mesh-free shape functions

$$\mathbf{u}(\mathbf{x}) \approx \mathbf{u}^h(\mathbf{x}) = \begin{bmatrix} \underbrace{\Phi^T(\mathbf{x})}_{1\times n_s} & \\ & \underbrace{\Phi^T(\mathbf{x})}_{1\times n_s} \end{bmatrix} \begin{bmatrix} \underbrace{\mathbf{U}}_{n_s\times 1} \\ \underbrace{\mathbf{V}}_{n_s\times 1} \end{bmatrix}. \tag{131}$$

It is opportune to distinguish between the mathematical vector $\Phi(\mathbf{x})$ which is size $n_s \times 1$

$$\Phi^T = \left[\phi_1(\mathbf{x})\ \phi_2(\mathbf{x})\ \ldots\ \phi_{n_s}\right] \tag{132}$$

and the computer array `PHI` in Sect. 3.7, which is a sparse matrix of size $n_g \times n_s$.

Applying (123)–(131), the strain resulting from the approximation is

$$\epsilon^h(\mathbf{x}) = \begin{bmatrix} \frac{\partial\Phi^T}{\partial x} & \\ & \frac{\partial\Phi^T}{\partial y} \\ \frac{\partial\Phi^T}{\partial y} & \frac{\partial\Phi^T}{\partial x} \end{bmatrix} \begin{bmatrix} \mathbf{U} \\ \mathbf{V} \end{bmatrix} \tag{133}$$

and applying (124)–(133)

$$\sigma^h = \begin{bmatrix} C_{11} & C_{12} & \\ C_{12} & C_{22} & \\ & & C_{33} \end{bmatrix} \begin{bmatrix} \frac{\partial\Phi^T}{\partial x} & \\ & \frac{\partial\Phi^T}{\partial y} \\ \frac{\partial\Phi^T}{\partial y} & \frac{\partial\Phi^T}{\partial x} \end{bmatrix} \begin{bmatrix} \mathbf{U} \\ \mathbf{V} \end{bmatrix}$$

$$= \begin{bmatrix} C_{11}\frac{\partial\Phi^T}{\partial x} + C_{12}\frac{\partial\Phi^T}{\partial y} \\ C_{11}\frac{\partial\Phi^T}{\partial x} + C_{22}\frac{\partial\Phi^T}{\partial y} \\ C_{33}\frac{\partial\Phi^T}{\partial y} + C_{33}\frac{\partial\Phi^T}{\partial x} \end{bmatrix} \begin{bmatrix} \mathbf{U} \\ \mathbf{V} \end{bmatrix}. \tag{134}$$

The variational term for the stiffness matrix can be expressed as follows

$$\int_\Omega \delta\epsilon^T \sigma\, d\Omega = \left[\delta\mathbf{U}^T\ \delta\mathbf{V}^T\right] \int_\Omega \begin{bmatrix} \frac{\partial\Phi}{\partial x} & & \frac{\partial\Phi}{\partial y} \\ & \frac{\partial\Phi}{\partial y} & \frac{\partial\Phi}{\partial x} \end{bmatrix} \times \begin{bmatrix} C_{11}\frac{\partial\Phi^T}{\partial x} + C_{12}\frac{\partial\Phi^T}{\partial y} \\ C_{11}\frac{\partial\Phi^T}{\partial x} + C_{22}\frac{\partial\Phi^T}{\partial y} \\ C_{33}\frac{\partial\Phi^T}{\partial y} + C_{33}\frac{\partial\Phi^T}{\partial x} \end{bmatrix} d\Omega \begin{bmatrix} \mathbf{U} \\ \mathbf{V} \end{bmatrix}. \tag{135}$$

Therefore the stiffness matrix $\mathbf{K}$ can be written as

$$\mathbf{K} = \begin{bmatrix} \mathbf{K}_{11} & \mathbf{K}_{12} \\ \mathbf{K}_{12}^T & \mathbf{K}_{22} \end{bmatrix}, \tag{136}$$

where

$$\mathbf{K}_{11} = \int_\Omega C_{11}\frac{\partial\Phi}{\partial x}\frac{\partial\Phi^T}{\partial x} + C_{33}\frac{\partial\Phi}{\partial y}\frac{\partial\Phi^T}{\partial y} d\Omega, \tag{137}$$

$$\mathbf{K}_{22} = \int_\Omega C_{33}\frac{\partial\Phi}{\partial x}\frac{\partial\Phi^T}{\partial x} + C_{22}\frac{\partial\Phi}{\partial y}\frac{\partial\Phi^T}{\partial y} d\Omega, \tag{138}$$

$$\mathbf{K}_{12} = \int_\Omega C_{12}\left(\frac{\partial\Phi}{\partial x}\frac{\partial\Phi^T}{\partial y} + \left(\frac{\partial\Phi}{\partial x}\frac{\partial\Phi^T}{\partial y}\right)^T\right) d\Omega. \tag{139}$$

The stiffness matrix can be then computed whenever the following matrices are calculated

$$\mathbf{K}_{xx} = \int_\Omega \frac{\partial\Phi}{\partial x}\frac{\partial\Phi^T}{\partial x} d\Omega, \tag{140}$$

$$\mathbf{K}_{yy} = \int_\Omega \frac{\partial\Phi}{\partial y}\frac{\partial\Phi^T}{\partial y} d\Omega, \tag{141}$$

$$\mathbf{K}_{xy} = \int_\Omega \frac{\partial\Phi}{\partial x}\frac{\partial\Phi^T}{\partial y} d\Omega. \tag{142}$$

Integrals in (140), (141) and (142) are calculated using Gaussian quadrature. Naming `W` as the vector of size $n_g \times 1$

containing the Gaussian weights associated with the Gaussian points GGRID, then a brute force algorithm would be

```
for i=1:ns
  for j=1:ns
  Kxx(i,j) = W.'*(DPHIx(:,i).*DPHIy(:,j))
  end
end
```

Double loops however, really slow down the calculation. To avoid one loop, one could then think to create an intermediate matrix

```
for i=1:nset
    WX(i,:) = DPHIx(:,i)'.*W.';
end
Kxx = WX*DPHIx;
```

Another possible approach that avoids a double looping is the creation of *repetitions* of the arrays containing the weights:

```
WX = DPHIx.*repmat(W,1,ns)
```

and then

```
Kxx = WX.'*DPHIy
```

The command `repmat` creates n_s replicas of the vector `W` along the columns. Nevertheless, such approach is extremely costly in term of speed and memory, since it creates a temporary array of size $n_g \times n_s$.

Instead, *indexing* can be used with the handle in Eq. 116. With the command `desp_handle` (118)

$$\texttt{DPHIx} = \texttt{desp_handle(DPHIx)} \tag{143}$$

the derivatives of the shape functions are reshaped into mono-dimensional arrays.

$$\texttt{WX} = \texttt{asb_handle(DPHIx,W)} \tag{144}$$

Then reshaping `DPHIx` and `WX` into bi-dimensional arrays with (117)

$$\texttt{DPHIx} = \texttt{sp_handle(DPHIx)} \tag{145}$$

$$\texttt{WX} = \texttt{sp_handle(WX)} \tag{146}$$

Finally Eq. 140 is computed

$$\texttt{KXX} = \texttt{WX.'*DPHIx} \tag{147}$$

The same approach can be repeated for Eqs. 142 and 141.

4 Results

In order to show the effectiveness of the improvements, in this section it will be presented a comparison between the *classic* and the *new* methods based on the computational run-times.

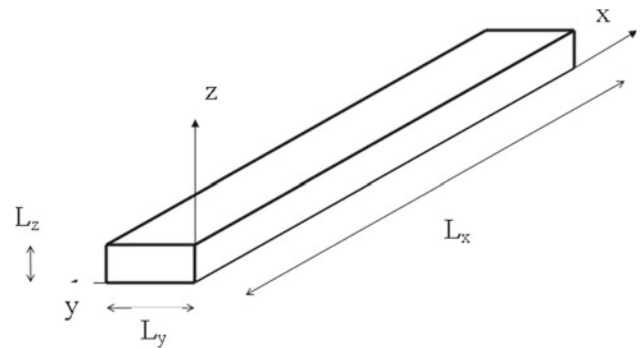


Fig. 12 Three-dimensional case: geometry

Table 2 Elastic properties for the benchmark case

E_{11}	$E_{22} = E_{33}$	G_{12}	G_{23}	$\nu_{12} = \nu_{13}$	ν_{23}
122.7 GPa	10.1 GPa	5.5 GPa	3.7 GPa	0.25	0.45

Table 3 Geometry for the benchmark case

L_x	L_y	L_z
185 mm	25 mm	2.5 mm

By *classic* it is intended the method based on a *point-wise* inversion of the moment matrix (32), i.e. the moment matrix is calculated and then the inversion is made by looping over the Gaussian points, create a temporary matrix, invert it with a standard numerical routine (LU factorization or Gaussian elimination) and then pass it back to the cell array `Minv`. Instead, the *new* method is the one explained in Sect. 3.6. For both methods, the routines used for the connectivity, the correction and the assembly are the same.

To assure a fair comparison, both methods have been executed on the same computer, with a 3 GHz processor Intel ®Pentium ®with 1 GB of RAM. Nevertheless, performances may vary and computational times reduce on different computers. Hence, the computational times reported here should be intended as a relative measure, not as an absolute measure of the code performances.

As a benchmark case, a three-dimensional model of a bar has been considered as in Fig. 12, with elastic properties and geometry resumed in respectively in Tables 2 and 3.

The quadrature cells considered are cubic as in Fig. 13 and a full quadratic basis (superscript 2) has been included as in Eq. 148, thus the moment matrix has size 10×10.

$$\mathbf{p}^{(2)T}(\mathbf{x}) = \left[1\ x\ y\ z\ x^2\ y^2\ z^2\ xy\ yz\ xz\right]. \tag{148}$$

Tables 4, 5, 6, 7, 8 and 9 illustrates the computational run-times for the specimen for same number of nodes ($11 \times 6 \times 3 = 132$), but different number of Gaussian points. The dilatation parameter ρ_I is variable with the node and it is chosen

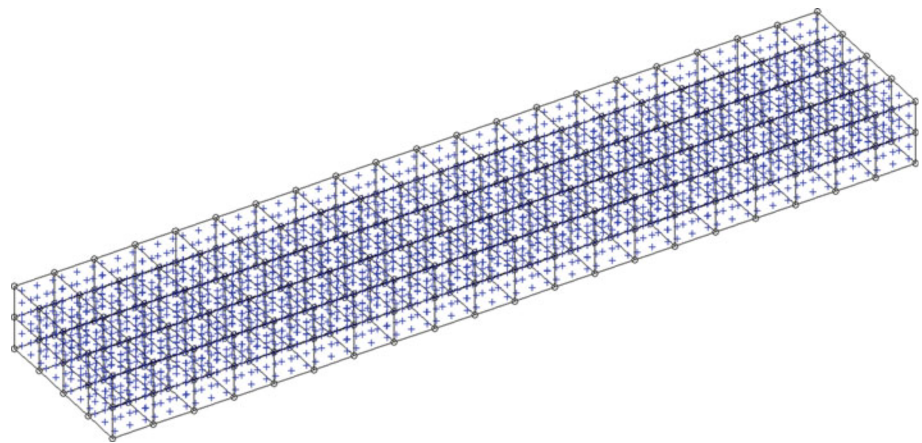


Fig. 13 *Continuous line* quadrature cells; *circles* nodes; *blue crosses* Gaussian points. (Color figure online)

Table 4 Computational run-times for the connectivity

Gaussian points	Classic (s)	New (s)
800	0.1614	0.1269
2,700	0.4329	0.4500
6,400	0.8698	0.8728
12,500	1.5980	1.5694
21,600	3.0833	3.2427
34,300	4.8564	4.2063
51,200	7.4106	7.2022

Table 5 Computational run-times for the correction term

Gaussian points	Classic (s)	New (s)
800	0.2220	0.3374
2,700	2.6361	2.5686
6,400	4.7568	4.7648
12,500	7.1372	7.7983
21,600	12.5379	18.1758
34,300	25.3300	24.7946
51,200	34.6201	33.2414

Table 6 Computational run-times for the assembly

Gaussian points	Classic (s)	New (s)
800	0.3424	0.3908
2,700	1.0952	1.0500
6,400	2.9279	3.1260
12,500	7.2566	5.9309
21,600	14.7860	11.1589
34,300	22.0235	21.5802
51,200	29.3857	31.1248

as 2.4 $max(h_I)$ with h_I the maximum length of the edges concurring in that node.

The run-time has been divided mainly in time for the creation of the *shape functions* and time for the *assembly* of the stiffness matrix. The run-times for the shape functions comprise also the derivatives of the shape functions. The run-time for the shape functions has been subdivided in 4 contributions:

Table 7 Computational run-times for the moment matrix

Gaussian points	Classic (s)	New (s)
800	1.8273	0.5012
2,700	11.4439	1.8473
6,400	22.4772	3.9266
12,500	49.4992	8.1483
21,600	75.7696	14.1422
34,300	135.0388	21.9535
51,200	170.9730	35.3794

Table 8 Computational run-times for the inversion of the moment matrix

Gaussian points	Classic (s)	New (s)
800	1.5319	1.8981
2,700	12.6961	9.5511
6,400	58.8198	21.1465
12,500	190.0229	39.9969
21,600	555.5533	72.0766
34,300	1344.9519	117.3779
51,200	3002.2513	162.7680

Table 9 Computational run-times for the shape functions

Gaussian points	Classic (s)	New (s)	New/classic (%)
800	3.7471	2.8693	76.57
2,700	27.2389	14.4471	53.04
6,400	86.9840	30.7691	35.37
12,500	248.3761	57.6299	23.20
21,600	647.1185	107.8531	16.67
34,300	1510.5029	168.6082	11.16
51,200	3215.7002	239.0169	7.43

1. *connectivity*: because the size of data sets ($n_g \times n_s$) is relatively small, brute force was used in both cases;
2. *moment matrix*, i.e. the time necessary to compute the cell array M as in Sect. 3.4;
3. *inversion of the moment matrix*, i.e. the time for inverting the moment matrix;
4. *correction term*, i.e. the time for calculating the correction to the kernel as in Sect. 3.7.

It can be observed that, as expected, the major differences are not in the connectivity (Table 4), in the correction term (Table 5) and in the assembly (Table 6). In fact, for these purposes, the same routines have been used in both methods. Instead, there is a great difference for the computation times related to the moment matrix (Table 7), even though the same procedure of Sect. 3.4 has been used for both cases. The reason for such discrepancy is that in the *classic* method, a symmetric 10×10 moment matrix need to be constructed straight after the connectivity phase, whereas with the new method (the partitioning inversion method) only a symmetric 4×4 moment matrix is necessary. The symmetric 4×4 moment matrix for the partitioning method can be inverted symbolically and represents the starting point of the iterative procedure described in Sect. 3.6. The remaining terms in the new method are constructed at the same time of the inversion, allowing a huge saving in terms of computational time, around 80% average for all the cases.

This saving is not only in terms of speed but also in terms of storage memory. In fact, the number of entries of the moment matrix stored in the classic method is $10 \times 11/2 = 55$, whereas in the partitioning method is $4 \times 5/2 = 10$. Therefore, an estimation of the saving in terms of speed is $(55 - 10)/55 \times 100 = 81.18\%$.

Moreover, for the storage memory, there is another aspect to consider, which is not evident from the computational times. After the inversion, the memory occupied by the moment matrix (4×4) can be cleared since it is no longer necessary. The update is executed directly on the *inverse* of the moment matrix, through previous calculation of the moments $\mathbf{b}$ (Eq. 92) and c (Eq. 93). After the update is executed, the moments $\mathbf{b}$ and c required for the update of the inverse can be deleted as well. On the contrary, for the classic method, all the terms must be retained for the *point-wise* inversion, and deleted only after the inversion is concluded. The inversion through partitioning of the moment matrix is particularly advantageous, as shown in Table 8. Moreover, Figs. 14 and 15 show another interesting side: the dataset of the computational times and the number of Gaussian points can be fit by a quadratic curve for the classic method and by a *linear* curve for the proposed method.

The proposed method, either for the inversion of the moment matrix and for the construction of the shape functions, has a computational cost $\mathscr{O}(n)$ with respect to the number of Gaussian points, compared to the $\mathscr{O}(n^2)$ of the classic one. The saving in computational time is tremendous: in fact, for the classic method (Table 9), for 51,200 Gaussian points, almost an hour (54 min) is needed to calculate the shape functions and their first derivatives, while with the proposed method only 4 min.

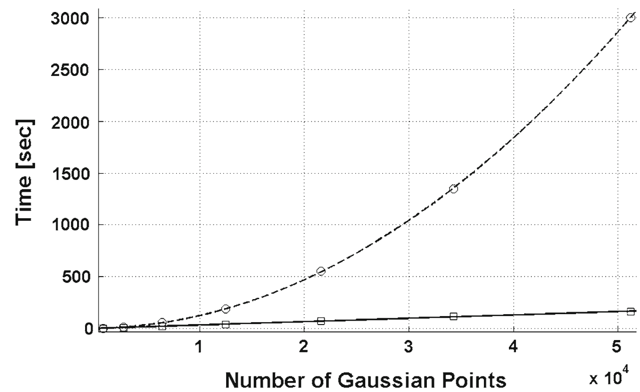


Fig. 14 Computational run-times for the inversion of the moment matrix: *circles* classic method; *squares* new method; *dashed line* quadratic fit for the classic method; *continuous line* linear fit for the new method

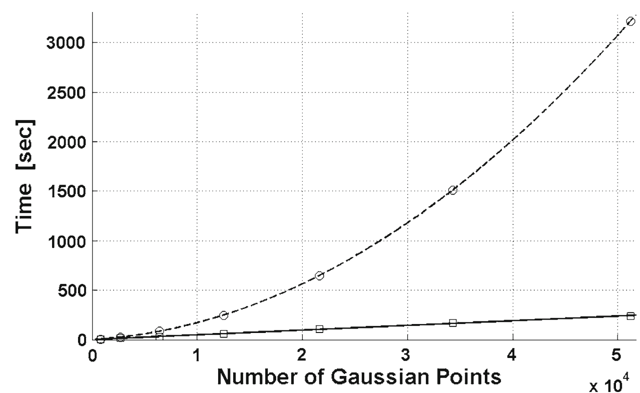


Fig. 15 Computational run-times for the shape functions: *circles* classic method; *squares* new method; *dashed line* quadratic fit for the classic method; *continuous line* linear fit for the new method

5 Conclusions

This paper reported a number of techniques able to reduce the computational costs of the construction of the shape functions in a MLS/RKPM. Firstly, the bottlenecks of the method were identified, namely the neighbour search, the inversion of the moment matrix and the assembly of the stiffness matrix.

The burden of the neighbour search can be alleviated through the use of the *kd*-tree algorithm, that also creates the *node-Gaussian point* connectivity that facilitates the assembly of the stiffness matrix. Another important result is the inversion of the moment matrix through partition, which

eliminates the need of point-wise inversion and *vectorizes* one of the most computationally expensive in meshfree methods.

It has been shown that, for a fixed number of degrees of freedom, the computational cost goes linearly with the number of Gaussian points. Gaussian points are needed to numerically evaluate the integrals in a weak form of the equations of equilibrium. A good compromise between accuracy and costs is given by a Gaussian quadrature of order 3, which means 9 Gaussian points per triangular element in two-dimensions and 27 Gaussian points per tetrahedral element in three-dimensions.

This means that the *bottle neck* is represented by the number of Gaussian points. In this sense, a leap ahead would be then represented by further researching stabilization of nodal integration techniques: a truly-meshfree method that do not require a background mesh for the purposes of the integration, will then eliminate the costs associated with the use of Gaussian points.

References

1. Barbieri E, Meo M (2009) Evaluation of the integral terms in reproducing kernel methods. Comput Methods Appl Mech Eng 198(33–36):2485–2507
2. Belytschko T, Fleming M (1999) Smoothing, enrichment and contact in the element-free Galerkin method. Comput Struct 71(2):173–195
3. Belytschko T, Tabbara M (1996) Dynamic fracture using element-free Galerkin methods. Int J Numer Methods Eng 39(6):923–938
4. Belytschko T, Gu L, Lu Y (1994) Fracture and crack growth by element-free Galerkin methods. Model Simul Mater Sci Eng 2:519–534
5. Belytschko T, Lu Y, Gu L (1994) Element-free Galerkin methods. Int J Numer Methods Eng 37(2):229–256
6. Belytschko T, Lu Y, Gu L (1995) Crack propagation by element-free Galerkin methods. Eng Fract Mech 51(2):295–315
7. Belytschko T, Lu Y, Gu L, Tabbara M (1995) Element-free Galerkin methods for static and dynamic fracture. Int J Solids Struct 32(17):2547–2570
8. Belytschko T, Krongauz Y, Fleming M, Organ D, Liu WS (1996) Smoothing and accelerated computations in the element free Galerkin method. J Comput Appl Math 74(1):111–126
9. Belytschko T, Krongauz Y, Organ D, Fleming M, Krysl P (1996) Meshless methods: an overview and recent developments. Comput Methods Appl Mech Eng 139(1–4):3–47
10. Breitkopf P, Rassineux A, Touzot G, Villon P (2000) Explicit form and efficient computation of MLS shape functions and their derivatives. Int J Numer Methods Eng 48(451):466
11. Cartwright C, Oliveira S, Stewart D (2001) A parallel quadtree algorithm for efficient assembly of stiffness matrices in meshfree Galerkin methods. In: Proceedings of the 15th international parallel and distributed processing symposium, pp 1194–1198
12. Chen J, Pan C, Wu C, Liu W (1996) Reproducing kernel particle methods for large deformation analysis of non-linear structures. Comput Methods Appl Mech Eng 139(1–4):195–227
13. Chen J, Pan C, Wu C (1997) Large deformation analysis of rubber based on a reproducing kernel particle method. Comput Mech 19(3):211–227
14. Collier N, Simkins D (2009) The quasi-uniformity condition for reproducing kernel element method meshes. Comput Mech 44(3):333–342
15. Cormen T, Leiserson C, Rivest R, Stein C (2001) Introduction to algorithms. MIT Press, Cambridge
16. Dolbow J, Belytschko T (1998) An introduction to programming the meshless element free Galerkin method. Arch Comput Methods Eng 5(3):207–241
17. Duarte C (1995) A review of some meshless methods to solve partial differential equations. TICAM Report 95-06
18. Fasshauer G (2007) Meshfree approximation methods with Matlab. World Scientific Publishing Co. Inc., River Edge
19. Finkel R, Bentley J (1974) Quad trees a data structure for retrieval on composite keys. Acta Inform 4(1):1–9
20. Fries T, Matthies H (2004) Classification and overview of meshfree methods. Informatikbericht Nr. 2003-3, Scientific Computing University
21. Griebel M, Schweitzer M (2002) A particle-partition of unity method–part II: efficient cover construction and reliable integration. SIAM J Sci Comput 23(5):1655–1682
22. Idelsohn S, Oñate E (2006) To mesh or not to mesh. That is the question. Comput Methods Appl Mech Eng 195(37–40):4681–4696
23. Klaas O, Shephard M (2000) Automatic generation of octree-based three-dimensional discretizations for partition of unity methods. Comput Mech 25(2):296–304
24. Krysl P, Belytschko T (2001) ESFLIB: a library to compute the element free Galerkin shape functions. Comput Methods Appl Mech Eng 190(15–17):2181–2206
25. Lancaster P, Salkauskas K (1981) Surfaces generated by moving least squares methods. Math Comput 37(155):141–158
26. Li S, Liu W (1996) Moving least-square reproducing kernel method part II: Fourier analysis. Comput Methods Appl Mech Eng 139(1–4):159–193
27. Li S, Liu W (2004) Meshfree particle methods. Springer, Berlin
28. Libersky L, Petschek A (1990) Smooth particle hydrodynamics with strength of materials. In: Proceedings of the next free-Lagrange conference, June 1990, Jackson Lake Lodge, Moran, Wyoming
29. Libersky L, Petschek A, Carney T, Hipp J, Allahdadi F (1993) High strain Lagrangian hydrodynamics: a three-dimensional SPH code for dynamic material response. J Comput Phys 109(1):67–75
30. Liu G (2003) Mesh free methods: moving beyond the finite element method. CRC Press, Boca Raton
31. Liu G, Tu Z (2002) An adaptive procedure based on background cells for meshless methods. Comput Methods Appl Mech Eng 191(17–18):1923–1943
32. Liu W, Jun S, Zhang Y (1995) Reproducing kernel particle methods. Int J Numer Methods Fluids 20(8–9):1081–1106
33. Liu W, Chen Y, Uras R, Chang C (1996) Generalized multiple scale reproducing kernel particle methods. Comput Methods Appl Mech Eng 139(1–4):91–157
34. Liu W, Hao W, Chen Y, Jun S, Gosz J (1997) Multiresolution reproducing kernel particle methods. Comput Mech 20(4):295–309
35. Liu W, Li S, Belytschko T (1997) Moving least-square reproducing kernel methods. (I) Methodology and convergence. Comput Methods Appl Mech Eng 143(1–2):113–154
36. Lucy L (1977) A numerical approach to the testing of the fission hypothesis. Astron J 82(12):1013–1024
37. Macri M, De S, Shephard M (2003) Hierarchical tree-based discretization for the method of finite spheres. Comput Struct 81(8–11):789–803
38. Monaghan J (1982) Why particle methods work. SIAM J Sci Stat Comput 3:422

39. Monaghan J (1988) An introduction to SPH. Comput Phys Commun 48(1):89–96
40. Monaghan J (1992) Smoothed particle hydrodynamics. Annu Rev Astron Astrophys 30(1):543–574
41. Moore A (1991) A tutorial on kd-trees. University of Cambridge Computer Laboratory. Technical Report No. 209. Extract from PhD thesis. http://www.autonlab.org/autonweb/14665.html
42. Nguyen V, Rabczuk T, Bordas S, Duflot M (2008) Meshless methods: a review and computer implementation aspects. Math Comput Simul 79:763–813
43. Rabczuk T, Belytschko T (2005) Adaptivity for structured meshfree particle methods in 2D and 3D. Int J Numer Methods Eng 63(11):1559–1582
44. Shaofan L, Liu W (2002) Meshfree and particle methods and their applications. Appl Mech Rev 55:1–34
45. Tabarraei A, Sukumar N (2005) Adaptive computations on conforming quadtree meshes. Finite Elements Anal Des 41(7–8):686–702
46. You Y, Chen J, Lu H (2003) Filters, reproducing kernel, and adaptive meshfree method. Comput Mech 31(3):316–326
47. Zhou J, Wang X, Zhang Z, Zhang L (2005) Explicit 3-D RKPM shape functions in terms of kernel function moments for accelerated computation. Comput Methods Appl Mech Eng 194(9–11):1027–1035