RKPM2D: Open-Source Implementation of Nodally Integrated Reproducing Kernel Particle Method for Solving Partial Differential Equations

Tsung-Hui Huang[1], Haoyan Wei[1], Jiun-Shyan Chen[1], Michael Hillman[2]

[1]Department of Structural Engineering, University of California San Diego, La Jolla, CA, USA

[2]Department of Civil and Environmental Engineering, Pennsylvania State University, State College, PA, USA

Abstract: We present an open-source software RKPM2D (download link) for solving PDEs under the Reproducing Kernel Particle Method (RKPM)-based meshfree computational framework. Compared to conventional mesh-based methods, RKPM provides many attractive features, such as arbitrary order of continuity and discontinuity, relaxed tie between the quality of the discretization and the quality of approximation, simple h- and p-adaptive refinement, and ability to embed physics-based enrichment functions, among others, that make RKPM promising for solving challenging engineering problems. The aim of the present software package is to support reproducible research and serve as an efficient test platform for further development of meshfree methods. The RKPM2D software consists of a set of data structures and subroutines for discretizing two-dimensional domains of arbitrary geometry, nodal representative domain creation by Voronoi diagram partitioning, reproducing kernel shape function generation, a complete meshfree solver, and visualization tools for post-processing. In this paper, a brief overview is given that covers the key theoretical aspects of RKPM, such as the reproducing kernel approximation, weak form using Nitsche's method for boundary condition enforcement, various domain integration schemes (Gauss quadrature and stabilized nodal integration methods), as well as the fully discrete equations. In addition, the

computer implementation aspects employed in RKPM2D are discussed in detail. Benchmark problems solved by RKPM2D are presented to demonstrate the convergence, efficiency, and robustness of the RKPM implementation.

Keywords: Meshfree Method, Reproducing Kernel Particle Method, Nodal Integration, Open-Source Software, RKPM2D

1 Introduction

In recent years, the Reproducing Kernel Particle Method (RKPM) [1, 2, 3] has been recognized as an effective numerical method for solving partial differential equations (PDEs). Compared to conventional mesh-based numerical methods such as the Finite Element Method (FEM), the reproducing kernel (RK) approximation in RKPM is constructed based on a set of scattered points without any mesh connectivity, and thus the strong tie between the quality of the discretization and the quality of approximation in conventional mesh-based methods is relaxed. This "meshfree" feature makes RKPM well-suited for solving large deformation and multiphysics problems where FEM suffers from mesh distortion or mesh entanglement [1, 4, 5]. In addition, RKPM provides controllable orders of continuity and completeness, independent from one another, which enables effective solutions of PDEs involving high-order smoothness or discontinuities, and accordingly, implementation of h- and p-adaptive refinement becomes straightforward [6, 7, 8, 9, 10]. Furthermore, the wavelet-like multi-resolution properties can be obtained in the RK approximation, making it suitable for multi-resolution and multi-scale modeling [6, 7, 8]. Recently, accelerated and convergent RKPM formulations have been developed with the employment of variationally consistent and stabilized nodal integration techniques [11, 12]. With abovementioned advantages, RKPM has been successfully applied to a number of challenging engineering problems, including thin shells [13, 14], manufacturing processes [15, 16, 17], image-based biomechanics [18], geomechanics [19, 20], fracture/damage mechanics [21, 22, 23], shock dynamics [24, 25] and penetration/fragmentation phenomena [26, 27, 28], to name a few. Interested readers can refer to [2, 29] for a comprehensive review of RKPM and its applications.

While the overall programing structure of meshfree Galerkin methods using Gauss integration have been discussed in [30, 31], and efficient neighbor searching algorithms and associated data structures for meshfree methods have been developed in [32, 33, 34, 35], a public domain RKPM-based source code is in high demand. In this paper, we present an RKPM-based open-source computational software called RKPM2D that can effectively solve PDEs in a 2-D domain with an arbitrary geometry. Nitsche's method [36, 37] is adopted for imposition of essential boundary conditions in the meshfree Galerkin equations. For domain integration, the variationally consistent and stabilized nodal integration methods [11, 38], Modified Stabilized Conforming Nodal Integration (MSCNI) [39] and Naturally Stabilized Nodal Integration (NSNI) [12] are implemented in RKPM2D in addition to the conventional Gauss quadrature scheme. The program consists of a set of data structures and subroutines for two-dimensional domain discretization, nodal representative domain creation by Voronoi diagram partitioning, reproducing kernel shape function generation, a meshfree Galerkin equation solver, and visualization tools for post-processing. For demonstration purposes, linear elastostatics is chosen as the model problem, and extensions of the RKPM open-source software for solving other types of PDEs are straightforward. The RKPM2D code is implemented under a MATLAB environment [40] with pre-process, solver, and post-process functions fully integrated.

This paper is organized as follows. A brief review of the basic equations of RKPM for linear elasticity is given in Section 2, where various domain integration techniques such as Gauss integration, direct nodal integration and stabilized nodal integration are introduced. In Section 3, the computer implementation aspects are presented, including neighbor search processes, RK shape function construction, and domain integration procedures. Benchmark problems are presented in Section 4 to demonstrate the capabilities of RKPM2D. Conclusions are then given in Section 5.

2 Overview of the Reproducing Kernel Particle Method

2.1 Reproducing Kernel Approximation

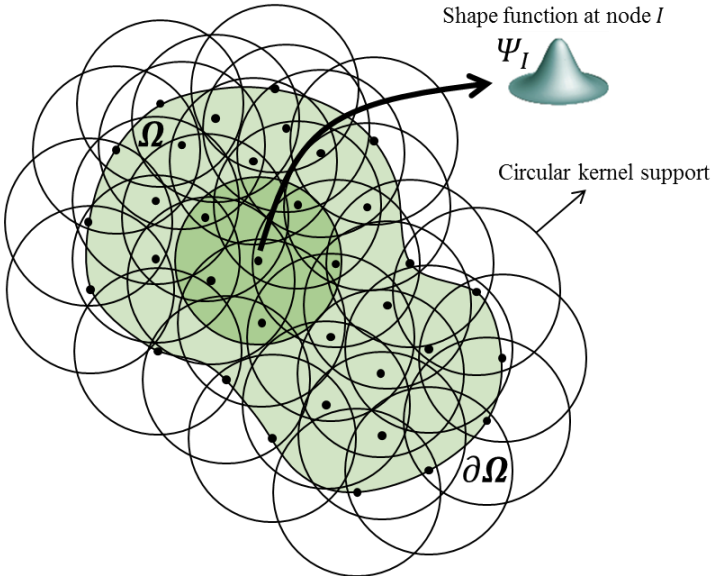


Figure 1. Illustration of a 2D RK discretization: support coverage and nodal shape function with circular kernel.

In RKPM, the numerical approximation is constructed based upon a set of scattered points or nodes [41]. The domain Ω is discretized by a set of nodes $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots \boldsymbol{x}_{NP}\}$ as shown in Figure 1, where $\boldsymbol{x}_I$ is the position vector of node I, and NP is the total number of nodes. The RK approximation of a function u is expressed as

$$u(\boldsymbol{x}) \approx u^h(\boldsymbol{x}) = \sum_{I \in G_x} \Psi_I(\boldsymbol{x}) u_I \tag{1}$$

where $\boldsymbol{x}$ is the spatial coordinates, u_I is the associated nodal coefficient to be determined, and $\Psi_I(\boldsymbol{x})$ is the reproducing kernel (RK) shape function of node I expressed as:

$$\Psi_I(\boldsymbol{x}) = \boldsymbol{H}^T(\boldsymbol{0}) \boldsymbol{M}^{-1}(\boldsymbol{x}) \boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}_I) \Phi_a(\boldsymbol{x} - \boldsymbol{x}_I) \tag{2}$$

where the basis vector $\boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}_I)$ is defined as

$$\boldsymbol{H}^T(\boldsymbol{x} - \boldsymbol{x}_I) = [1, x_1 - x_{1I}, x_2 - x_{2I}, x_3 - x_{3I}, (x_1 - x_{1I})^2, \ldots, (x_3 - x_{3I})^n \,] \tag{3}$$

and $\boldsymbol{M}(\boldsymbol{x})$ is the so-called moment matrix:

$$\boldsymbol{M}(\boldsymbol{x}) = \sum_{I \in G_{\boldsymbol{x}}} \boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}_I)\boldsymbol{H}^T(\boldsymbol{x} - \boldsymbol{x}_I)\Phi_a(\boldsymbol{x} - \boldsymbol{x}_I) \tag{4}$$

The set $G_{\boldsymbol{x}} = \{I | \Phi_a(\boldsymbol{x} - \boldsymbol{x}_I) \neq 0\}$ shown in Eq. (1) and (4) contains the nodal indexes of point $\boldsymbol{x}'s$ neighbors, and $\Phi_a(\boldsymbol{x} - \boldsymbol{x}_I)$ is the kernel function centered at $\boldsymbol{x}_I$ with compact support size a_I defined as

$$a_I = \tilde{c}h_I \tag{5}$$

In the above equation, $\tilde{c}$ is the normalized support size, and h_I is the nodal spacing associated with nodal point $\boldsymbol{x}_I$ defined as:

$$h_I = max\left(\left\|\boldsymbol{x}_J - \boldsymbol{x}_I\right\|\right), \ \ \forall \boldsymbol{x}_J \in B_I \tag{6}$$

in which the set B_I contains the four nodes that are closest to point $\boldsymbol{x}_I$ for 2D problems. The kernel function controls the smoothness of the approximation as shown in Figure 2, where the C^0 tent kernel function is compared with the following C^2 cubic B-spline kernel function:

$$\Phi_a(\boldsymbol{x} - \boldsymbol{x}_I) = \begin{cases} 2/3 - 4z_I^2 + 4z_I^3 & \text{for} \quad 0 \leq z_I \leq 1/2\,, \\ 4/3 - 4z_I + 4z_I^2 - 4/3\, z_I^3 & \text{for} \quad 1/2 \leq z_I \leq 1, \\ 0 & \text{for} \quad z_I > 1, \end{cases} \tag{7}$$

in which z_I is defined as $z_I = \frac{\|\boldsymbol{x} - \boldsymbol{x}_I\|}{a_I}$. In addition, shape functions with different normalized support sizes are plotted in Figure 3, which clearly illustrates that the locality of the approximation is controlled by the kernel support size.

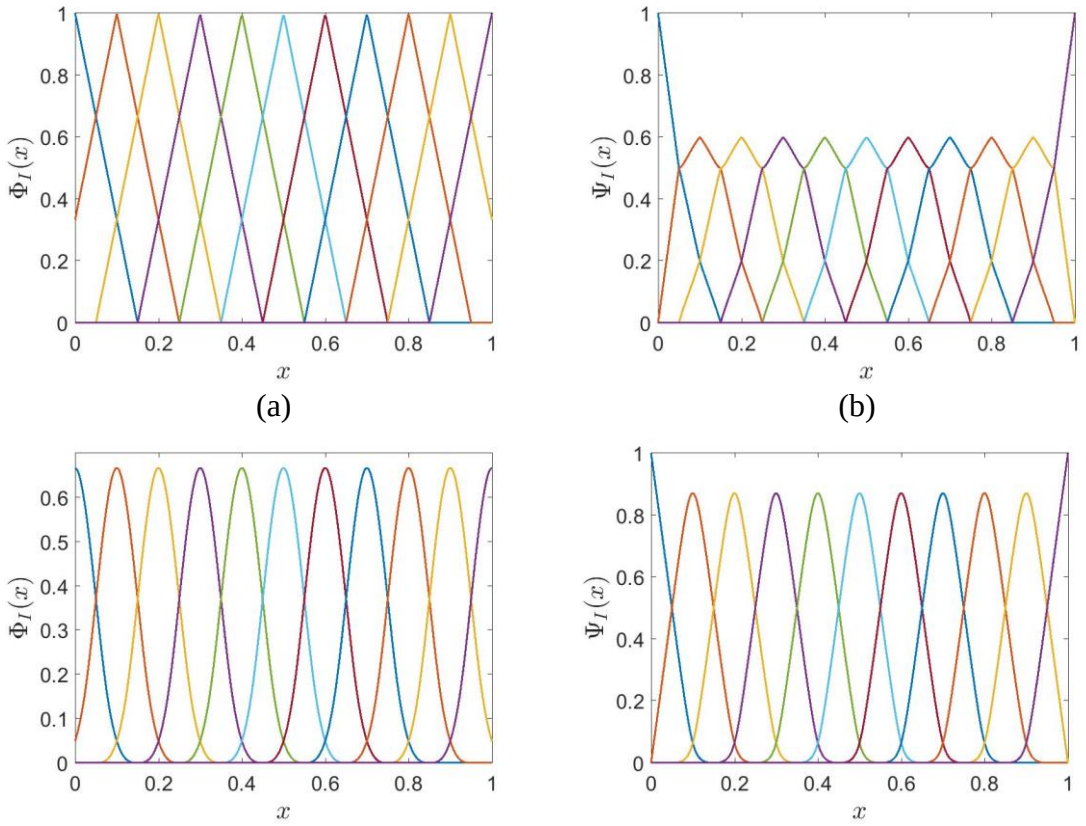


Figure 2. Kernel function and corresponding RK shape function with linear basis and normalized support size $\tilde{c} = 1.5$: (a) tent kernel and (b) corresponding RK shape function. (c) cubic B-Spline kernel and (d) corresponding RK shape function.

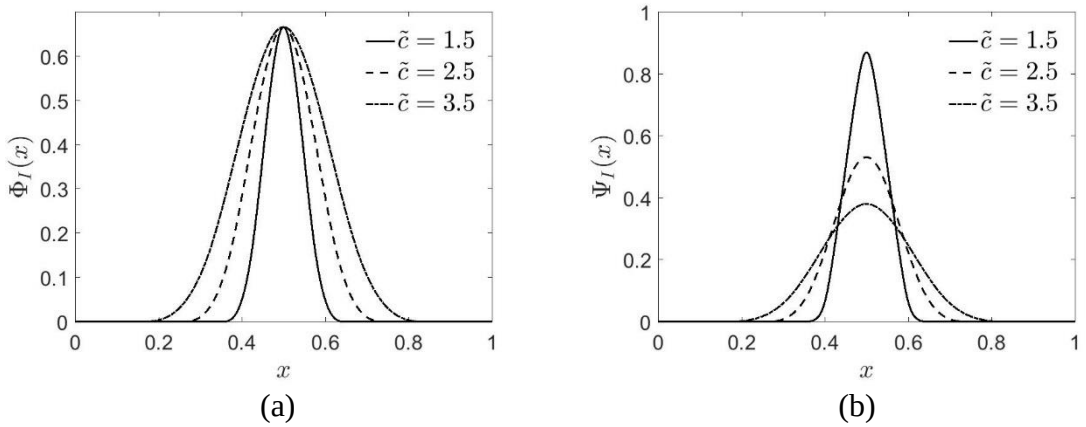
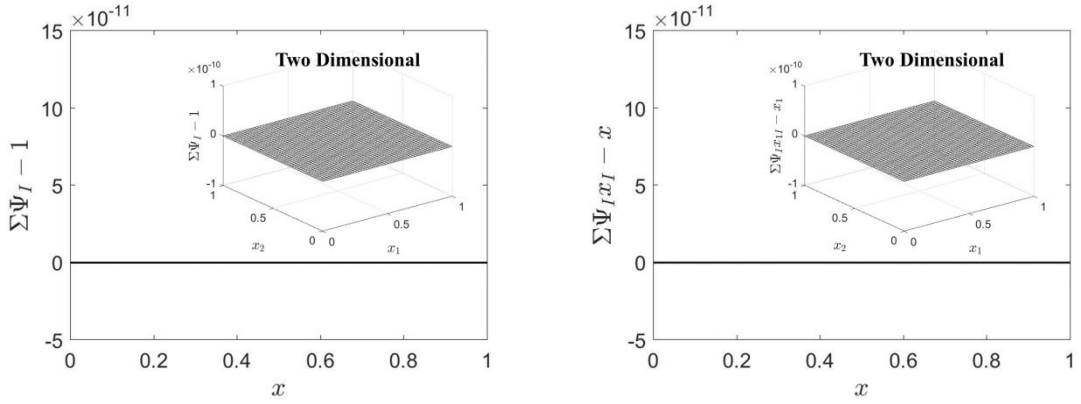


Figure 3. (a) Cubic B-spline kernel and (b) corresponding RK shape function (right) with linear basis and normalized support size $\tilde{c} = 1.5, 2.5, \text{and } 3.5$.

By construction, the RK shape functions satisfy the following n^{th} order reproducing conditions:

$$\sum_{I \in G_x} \Psi_I(\boldsymbol{x}) x_{1I}^i x_{2I}^j x_{3I}^k = x_1^i x_2^j x_3^k, \qquad 0 \le i+j+k \le n \tag{8}$$

where n is the specified order of completeness, which determines the order of consistency in the solution of PDEs. When linear basis is employed, both the zero-th and first-order reproducing conditions are satisfied as shown in Figure 4.



(a) (b)

Figure 4. (a) Zero-th and (b) first order reproducing condition verification for the RK shape function with linear basis and normalized support size $\tilde{c} = 1.5$.

2.2 Galerkin Formulation

Consider the following linear elasticity problem:

$$\begin{aligned} \sigma_{ij,j} + b_i &= 0 \quad \text{on } \Omega \\ \sigma_{ij} n_j &= t_i \quad \text{on } \partial\Omega_t \\ u_i &= g_i \quad \text{on } \partial\Omega_g \end{aligned} \tag{9}$$

where u_i is the displacement, $\sigma_{ij} = C_{ijkl}\varepsilon_{kl}$ is the Cauchy stress, C_{ijkl} is the elasticity tensor, $\varepsilon_{ij} = \left(u_{i,j} + u_{j,i}\right)/2$ is the strain, n_j is the surface normal on $\partial\Omega$, b_i is the body force, and t_i and g_i denote the prescribed traction and displacement on $\partial\Omega_t$ and $\partial\Omega_g$, respectively. Using Nitsche's method [42] for the enforcement of essential boundary conditions, the weak form of Eq. (9) can be written as follows

$$\int_{\Omega} \delta\varepsilon_{ij} C_{ijkl} \varepsilon_{kl} d\Omega$$

$$= \int_{\Omega} \delta u_i b_i d\Omega + \int_{\partial\Omega_t} \delta u_i t_i d\Gamma + \int_{\partial\Omega_g} \delta u_i \lambda_i d\Gamma \tag{10}$$

$$+ \int_{\partial\Omega_g} \delta\lambda_i (u_i - g_i) d\Gamma + \beta \int_{\partial\Omega_g} \delta u_i (u_i - g_i) d\Gamma$$

where λ_i is the Lagrange multiplier, and in Nitsche's method it is taken as the surface traction for elasticity problems, i.e., $\lambda_i = \sigma_{ij} n_j$, and $\beta = \beta_{nor} E/\bar{h}$ with β_{nor} the normalized penalty parameter, E the Young's modulus, and $\bar{h}$ the average of nodal spacing. Considering the following RK approximation for $\boldsymbol{u}$ and $\delta\boldsymbol{u}$:

$$\boldsymbol{u}^h = \sum_{I \in G_{\boldsymbol{x}}} \Psi_I(\boldsymbol{x}) \boldsymbol{u}_I \,, \quad \delta\boldsymbol{u}^h = \sum_{I \in G_{\boldsymbol{x}}} \Psi_I(\boldsymbol{x}) \delta\boldsymbol{u}_I, \tag{11}$$

Eq. (10) yields the following matrix equation:

$$\sum_J \boldsymbol{K}_{IJ} \boldsymbol{u}_J - \boldsymbol{F}_I = \boldsymbol{0}\,, \qquad \forall I \tag{12}$$

where

$$\boldsymbol{K}_{IJ} = \boldsymbol{K}_{IJ}^c + \boldsymbol{K}_{IJ}^\beta - \left(\boldsymbol{K}_{IJ}^g + \boldsymbol{K}_{IJ}^{g\,T}\right) \tag{13}$$

$$\boldsymbol{F}_I = \boldsymbol{F}_I^b + \boldsymbol{F}_I^t + \boldsymbol{F}_{IJ}^\beta - \boldsymbol{F}_I^g \tag{14}$$

in which each matrix and vector for two-dimensional elasticity are expressed as

$$\boldsymbol{K}_{IJ}^c = \int_{\Omega} \boldsymbol{B}_I^T(\boldsymbol{x}) \boldsymbol{C} \boldsymbol{B}_J(\boldsymbol{x}) d\Omega \tag{15}$$

$$\boldsymbol{K}_{IJ}^\beta = \beta \int_{\partial\Omega_g} \boldsymbol{\Psi}_I^T(\boldsymbol{x}) \boldsymbol{S} \boldsymbol{\Psi}_J(\boldsymbol{x}) d\Gamma \tag{16}$$

$$\boldsymbol{K}_{IJ}^{g} = \int_{\partial\Omega_g} \boldsymbol{B}_I^T(\boldsymbol{x})\boldsymbol{C}\boldsymbol{\eta}\boldsymbol{S}\boldsymbol{\Psi}_J(\boldsymbol{x})d\Gamma \tag{17}$$

$$\boldsymbol{F}_I^b = \int_{\Omega} \boldsymbol{\Psi}_I^T(\boldsymbol{x})\boldsymbol{b}(\boldsymbol{x})d\Omega \tag{18}$$

$$\boldsymbol{F}_I^t = \int_{\partial\Omega_t} \boldsymbol{\Psi}_I^T(\boldsymbol{x})\boldsymbol{t}(\boldsymbol{x})d\Gamma \tag{19}$$

$$\boldsymbol{F}_{IJ}^{\beta} = \beta \int_{\partial\Omega_g} \boldsymbol{\Psi}_I^T(\boldsymbol{x})\boldsymbol{S}\boldsymbol{g}d\Gamma \tag{20}$$

$$\boldsymbol{F}_I^g = \int_{\partial\Omega_g} \boldsymbol{B}_I^T(\boldsymbol{x})\boldsymbol{C}\boldsymbol{\eta}\boldsymbol{S}\boldsymbol{g}d\Gamma \tag{21}$$

$$\boldsymbol{B}_I(\boldsymbol{x}) = \begin{bmatrix} \Psi_{I,1}(\boldsymbol{x}) & 0 \\ 0 & \Psi_{I,2}(\boldsymbol{x}) \\ \Psi_{I,2}(\boldsymbol{x}) & \Psi_{I,1}(\boldsymbol{x}) \end{bmatrix}, \qquad \boldsymbol{\Psi}_I(\boldsymbol{x}) = \begin{bmatrix} \Psi_I(\boldsymbol{x}) & 0 \\ 0 & \Psi_I(\boldsymbol{x}) \end{bmatrix}, \tag{22}$$

$$\boldsymbol{\eta} = \begin{bmatrix} n_1 & 0 \\ 0 & n_2 \\ n_2 & n_1 \end{bmatrix}, \ \boldsymbol{S} = \begin{bmatrix} s_1 & 0 \\ 0 & s_2 \end{bmatrix}, \ \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \ \boldsymbol{t} = \begin{bmatrix} t_1 \\ t_2 \end{bmatrix}, \ \boldsymbol{g} = \begin{bmatrix} g_1 \\ g_2 \end{bmatrix}. \tag{23}$$

where n_i is the component of the surface unit normal on the essential boundary, and $s_i = 0$ or 1 serves as a switch for imposing each component of the boundary displacement.

2.3 Domain Integration

Domain integration plays an important role in accuracy, stability and convergence of meshfree methods. Unlike FEM which utilizes the element topology for integration, quadrature domains for meshfree methods can be chosen either as background cells that are independent from the point locations, or associated with the nodal representative domains. The former scheme is commonly adopted in conjunction with the Gauss

quadrature scheme and the latter is used for nodal integration schemes; both have been implemented in RKPM2D as discussed in this section.

2.3.1 Gauss Integration

When Gauss quadrature is adopted, quadrature points are generated based upon background cells [43, 44] as shown in Figure 5, where only the quadrature points inside the physical domain are considered for domain integration. Gauss points for contour integrals are generated along the natural and essential boundaries, also shown in Figure 5.

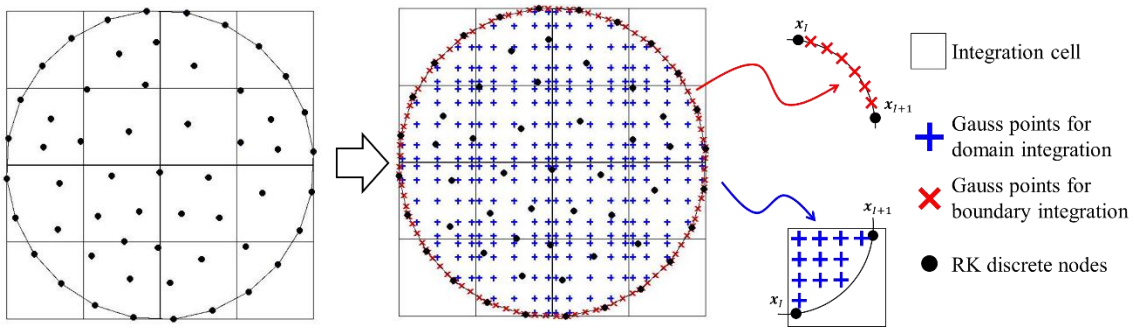


Figure 5. Meshfree points and background Gauss quadrature points in arbitrary two-dimensional domain Ω.

The domain and boundary integrations in (15) - (21) are computed as follows:

$$\int_{\Omega} \boldsymbol{P}(\boldsymbol{x}) d\Omega \approx \sum_{N=1}^{NG} \boldsymbol{P}(\boldsymbol{x}_N) W_N \tag{24}$$

$$\int_{\partial\Omega_g} \boldsymbol{Q}(\boldsymbol{x}) d\Gamma \approx \sum_{N=1}^{NGg} \boldsymbol{Q}(\hat{\boldsymbol{x}}_N) \widehat{W}_N \tag{25}$$

where $\boldsymbol{P}(\boldsymbol{x})$ and $\boldsymbol{Q}(\boldsymbol{x})$ denote integrands in the domain and boundary integrals in (15) - (21); $\boldsymbol{x}_N$, W_N and NG are the domain Gauss points, weights, and the number of domain Gauss points, respectively, and $\hat{\boldsymbol{x}}_N$, $\widehat{W}_N$ and NGg are essential boundary Gauss points, weights, and the number of essential boundary Gauss points, respectively. The same

integration rules are used for the natural boundary integration. Since RK shape functions are rational functions and their supports overlap with each other, the misalignment of Gauss integration cells and shape function supports lead to large quadrature errors unless high-order integration schemes are adopted, as shown in [11, 43]. Nonetheless, Gauss integration is chosen as a reference quadrature scheme herein.

2.3.2 Variationally Consistent Nodal Integration

The simplest nodal integration method is Direct Nodal Integration (DNI), where shape functions and their derivatives are evaluated directly at nodes.

The domain and boundary integrations in (15) - (21) are computed as follows:

$$\int_{\Omega} \boldsymbol{P}(\boldsymbol{x}) d\Omega \approx \sum_{N=1}^{NP} \boldsymbol{P}(\boldsymbol{x}_N) A_N \tag{26}$$

$$\int_{\partial\Omega_g} \boldsymbol{Q}(\boldsymbol{x}) d\Gamma \approx \sum_{N=1}^{NPg} \boldsymbol{Q}(\hat{\boldsymbol{x}}_N) L_N \tag{27}$$

where $\boldsymbol{x}_N$, A_N and NP are the RK node locations, nodal representative domain areas and the number of RK nodes, respectively, and $\hat{\boldsymbol{x}}_N$, L_N and NPg are essential boundary RK nodes, length of the nodal representative domain, and the number of RK nodes on the essential boundary, respectively. The same integration rules are used for the natural boundary integration.

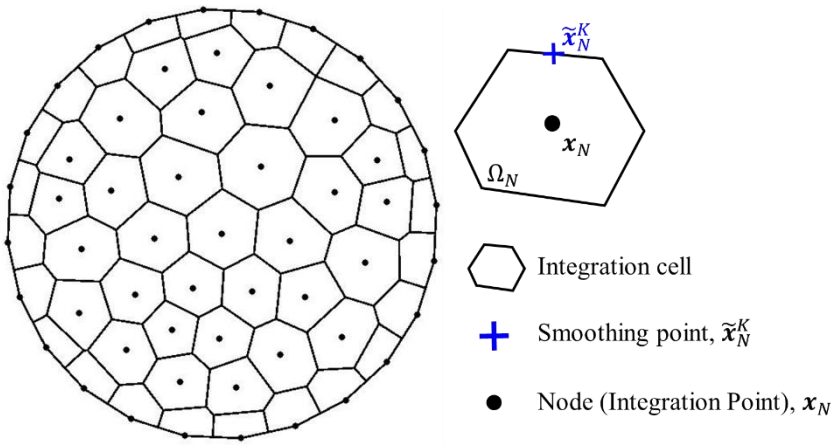


Figure 6. Voronoi cell diagram in two-dimensional domain Ω.

DNI is notorious for spurious zero-energy modes and non-convergent numerical solutions. To ensure linear variational consistency, i.e., the ability of numerical methods to pass the linear patch test, Chen et al. [38] showed that the quadrature rules need to meet the following first order integration constraint for the shape function gradient:

$$\hat{\int_{\Omega}} \Psi_{I,i} d\Omega = \hat{\int_{\partial\Omega}} \Psi_I n_i d\Gamma \tag{28}$$

In (28), ^ over the integral symbols denotes numerical integration. For nodal integration as the quadrature rule for the domain integration on the left hand side of Eq. (28), Chen et al. [38] introduced the following nodally smoothed gradient $\widetilde{\Psi}_{I,i}$ at the nodal point $\boldsymbol{x}_N$:

$$\widetilde{\Psi}_{I,i}(\boldsymbol{x}_N) = \frac{1}{A_N} \int_{\Omega_N} \Psi_{I,i}(\boldsymbol{x}) d\Omega = \frac{1}{A_N} \int_{\partial\Omega_N} \Psi_I(\boldsymbol{x}) n_i(\boldsymbol{x}) d\Gamma \tag{29}$$

where A_N denotes the area of the nodal representative domain Ω_N associated with node N, and n_i denotes the i^{th} component of the outward unit normal vector to the smoothing domain boundary as shown in Fig. 6. It was shown in [38] that integrating Eq. (28) with nodal integration with the smoothed gradient of shape function in Eq. (29), the first order integration constraint in Eq. (28) is exactly satisfied as long as the same boundary integral

quadrature rules are used for the right hand side of both Eqns. (28) and (29). As discussed in [11], in order to maintain linear consistency of the smoothed gradient of a linearly consistent shape function, a simple one-point Gauss integration rule can be used for the contour integral in Eq. (29):

$$\widetilde{\Psi}_{I,i}(\boldsymbol{x}_N) \approx \frac{1}{A_N} \sum_{K \in S_N} \Psi_I(\tilde{\boldsymbol{x}}_N^K) n_i(\tilde{\boldsymbol{x}}_N^K) L_K \tag{30}$$

where $S_N = \{K | \tilde{\boldsymbol{x}}_N^K \in \partial\Omega_N\}$ contains all center points of each boundary segment associated with node $\boldsymbol{x}_N$, and the integration weight L_K is the length of the K^{th} segment of the smoothing cell boundary. By employing smoothed shape function gradients, the stiffness matrix as well as the force vectors are re-formulated as follows:

$$\boldsymbol{K}_{IJ}^c = \int_\Omega \boldsymbol{B}_I^T(\boldsymbol{x}) \boldsymbol{C} \boldsymbol{B}_J(\boldsymbol{x}) d\Omega \approx \sum_{N=1}^{NP} \widetilde{\boldsymbol{B}}_I^T(\boldsymbol{x}_N) \boldsymbol{C} \widetilde{\boldsymbol{B}}_J(\boldsymbol{x}_N) A_N \tag{31}$$

$$\boldsymbol{K}_{IJ}^g = \int_{\partial\Omega_g} \boldsymbol{B}_I^T(\boldsymbol{x}) \boldsymbol{C} \boldsymbol{\eta} \boldsymbol{S} \boldsymbol{\Psi}_J(\boldsymbol{x}) d\Gamma \approx \sum_{N=1}^{NPg} \widetilde{\boldsymbol{B}}_I^T(\boldsymbol{x}_N) \boldsymbol{C} \boldsymbol{\eta} \boldsymbol{S} \boldsymbol{\Psi}_J(\boldsymbol{x}_N) L_N \tag{32}$$

$$\boldsymbol{F}_I^g = \int_{\partial\Omega_g} \boldsymbol{B}_I^T(\boldsymbol{x}) \boldsymbol{C} \boldsymbol{\eta} \boldsymbol{S} \boldsymbol{g} d\Gamma \approx \sum_{N=1}^{NPg} \widetilde{\boldsymbol{B}}_I^T(\boldsymbol{x}_N) \boldsymbol{C} \boldsymbol{\eta} \boldsymbol{S} \boldsymbol{g} L_N \tag{33}$$

where $\widetilde{\boldsymbol{B}}_I(\boldsymbol{x}_N)$ is defined as

$$\widetilde{\boldsymbol{B}}_I(\boldsymbol{x}_N) = \begin{bmatrix} \widetilde{\Psi}_{I,1}(\boldsymbol{x}_N) & 0 \\ 0 & \widetilde{\Psi}_{I,2}(\boldsymbol{x}_N) \\ \widetilde{\Psi}_{I,2}(\boldsymbol{x}_N) & \widetilde{\Psi}_{I,1}(\boldsymbol{x}_N) \end{bmatrix} \tag{34}$$

2.3.3 Stabilized Nodal Integration Schemes

Spurious oscillatory modes can be triggered in nodal integration methods. Therefore, additional stabilization techniques are needed to eliminate these low-energy modes, which will be described in this subsection.

2.3.3.1 Modified Stabilized Nodal Integration

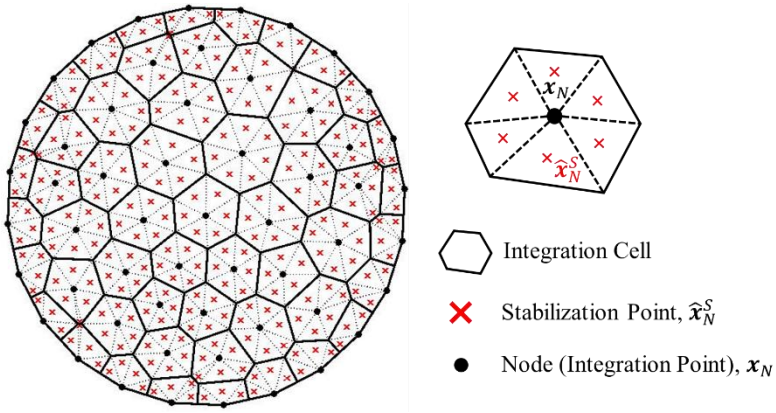


Figure 7. Illustration of nodal integration cells of the modified stabilized nodal integration.

The first stabilization technique employed here for eliminating spurious low-energy modes is called modified stabilized nodal integration [39, 45], where a least-squares type stabilization term is introduced into the stiffness matrix:

$$\boldsymbol{K}_{IJ}^{c} = \sum_{N=1}^{NP} \left(\underbrace{\widetilde{\boldsymbol{B}}_I^T(\boldsymbol{x}_N)\boldsymbol{C}\widetilde{\boldsymbol{B}}_J(\boldsymbol{x}_N)A_N}_{\text{SCNI}} + \underbrace{c_{stab}\sum_{S=1}^{NS}\left(\widetilde{\boldsymbol{B}}_I^T(\boldsymbol{x}_N) - \widetilde{\boldsymbol{B}}_I^T(\hat{\boldsymbol{x}}_N^S)\right)\boldsymbol{C}\left(\widetilde{\boldsymbol{B}}_J(\boldsymbol{x}_N) - \widetilde{\boldsymbol{B}}_J(\hat{\boldsymbol{x}}_N^S)\right)A_N^S}_{\text{stabilization}} \right) \tag{35}$$

where NS denotes the number of sub-cells associated with each nodal integration cell (as shown in Figure 7), $\hat{\boldsymbol{x}}_N^S$ denotes the centroid of the S^{th} sub-cell, $\widetilde{\boldsymbol{B}}_J(\hat{\boldsymbol{x}}_N^S)$ is the smoothed gradient evaluated by Eq. (30) and (34) for the S^{th} sub-cell and A_N^S denotes the area of the S^{th} sub-cell of the N^{th} nodal cell. Here, $0 \leq c_{stab} \leq 1$ is a stabilization parameter, which is chosen to be $c_{stab} = 1$ based on the study of Puso et al. [45] for elasticity. If the direct

gradient $\boldsymbol{B}_I(\boldsymbol{x}_N)$ and $\boldsymbol{B}_I(\hat{\boldsymbol{x}}_N^S)$ are used for nodal integration and stabilization terms in Eq. (35), the stiffness matrix is formulated as:

$$\boldsymbol{K}_{IJ}^c = \sum_{N=1}^{NP} \left(\underbrace{\boldsymbol{B}_I^T(\boldsymbol{x}_N)\boldsymbol{C}\boldsymbol{B}_J(\boldsymbol{x}_N)A_N}_{\text{DNI}} + \underbrace{c_{stab}\sum_{S=1}^{NS}\left(\boldsymbol{B}_I^T(\boldsymbol{x}_N) - \boldsymbol{B}_I^T(\hat{\boldsymbol{x}}_N^S)\right)\boldsymbol{C}\left(\boldsymbol{B}_J(\boldsymbol{x}_N) - \boldsymbol{B}_J(\hat{\boldsymbol{x}}_N^S)\right)A_N^S}_{\text{stabilization}} \right) \tag{36}$$

which is a modified stabilization for the DNI method. For comparison purposes, in the rest of this paper we will refer the modified formulations (35) based on SCNI as Modified SCNI (MSCNI), and (36) based on DNI as Modified DNI (MDNI), respectively. The least-squares type stabilization term in (35) enhances the coercivity of the discrete formulation and suppresses spurious low-energy modes in nodal integrations, but meanwhile, a number of additional shape function evaluations are required.

2.3.3.2 Naturally Stabilized Nodal Integration

The other stabilized integration technique employed here is the *naturally stabilized nodal integration* (NSNI) proposed in [12], where an implicit gradient expansion of the strain field is introduced as:

$$\boldsymbol{\varepsilon}\left(\boldsymbol{u}^h(\boldsymbol{x})\right) \approx \boldsymbol{\varepsilon}_N\left(\boldsymbol{u}^h(\boldsymbol{x}_N) + \sum_{i=1}^{d}(x_i - x_{iI})\hat{\boldsymbol{u}}_{,i}^h(\boldsymbol{x}_N)\right) \tag{37}$$

where $\hat{\boldsymbol{u}}_{,i}^h(\boldsymbol{x}_N) = \sum_{N=1}^{NP}\Psi_{iI}^{\nabla}(\boldsymbol{x}_N)\boldsymbol{u}_N$ is the implicit gradient of the displacement with Ψ_{iI}^{∇} the implicit gradient of the RK shape function [22]:

$$\Psi_{iI}^{\nabla} = \boldsymbol{H}_i^T\boldsymbol{M}^{-1}(\boldsymbol{x})\boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}_I)\Phi_a(\boldsymbol{x} - \boldsymbol{x}_I) \tag{38}$$

where $\boldsymbol{H} = [1 \quad x_1 - x_{1I} \quad x_2 - x_{2I}]^T$ and the vector $\boldsymbol{H}_i$ takes on the following values for linear basis:

$$\begin{aligned} \boldsymbol{H}_1 &= [0 \quad -1 \quad 0]^T \\ \boldsymbol{H}_2 &= [0 \quad 0 \quad -1]^T \end{aligned} \tag{39}$$

Introducing the gradient expansion terms (37) into the variational equations, the stiffness matrix is obtained as

$$\boldsymbol{K}_{IJ}^c = \sum_{N=1}^{NP} \left(\underbrace{\widetilde{\boldsymbol{B}}_I^T(\boldsymbol{x}_N)\boldsymbol{C}\widetilde{\boldsymbol{B}}_J(\boldsymbol{x}_N)A_N}_{\text{SCNI}} + \underbrace{\boldsymbol{B}_{1I}^{\nabla^T}(\boldsymbol{x}_N)\boldsymbol{C}\boldsymbol{B}_{1J}^{\nabla}(\boldsymbol{x}_N)M_{1N} + \boldsymbol{B}_{2I}^{\nabla^T}(\boldsymbol{x}_N)\boldsymbol{C}\boldsymbol{B}_{2J}^{\nabla}(\boldsymbol{x}_N)M_{2N}}_{\text{stabilization}} \right) \tag{40}$$

where $\boldsymbol{B}_{1I}^{\nabla}(\boldsymbol{x}_N)$ and $\boldsymbol{B}_{2I}^{\nabla}(\boldsymbol{x}_N)$ are defined as follows:

$$\boldsymbol{B}_{1I}^{\nabla}(\boldsymbol{x}_N) = \begin{bmatrix} \Psi_{I1,1}^{\nabla}(\boldsymbol{x}_N) & 0 \\ 0 & \Psi_{I1,2}^{\nabla}(\boldsymbol{x}_N) \\ \Psi_{I1,2}^{\nabla}(\boldsymbol{x}_N) & \Psi_{I1,1}^{\nabla}(\boldsymbol{x}_N) \end{bmatrix}, \boldsymbol{B}_{2I}^{\nabla}(\boldsymbol{x}_N) = \begin{bmatrix} \Psi_{I2,1}^{\nabla}(\boldsymbol{x}_N) & 0 \\ 0 & \Psi_{I2,2}^{\nabla}(\boldsymbol{x}_N) \\ \Psi_{I2,2}^{\nabla}(\boldsymbol{x}_N) & \Psi_{I2,1}^{\nabla}(\boldsymbol{x}_N) \end{bmatrix} \tag{41}$$

and M_{1N}, M_{2N} are the second moments of inertia in each nodal integration domain:

$$M_{1N} = \int_{\Omega_N} (x_1 - x_{1N})^2 d\Omega, \quad M_{2N} = \int_{\Omega_N} (x_2 - x_{2N})^2 d\Omega \tag{42}$$

From Eqns. (40) - (42), no subdivision of integration cells is required in the stabilization. Similar to the discussion in the previous section, this stabilization technique can also be employed in conjunction with DNI by replacing the smoothed gradients in Eq. (40) with direct gradients:

$$\boldsymbol{K}_{IJ}^c = \sum_{N=1}^{NP} \left(\underbrace{\boldsymbol{B}_I^T(\boldsymbol{x}_N)\boldsymbol{C}\boldsymbol{B}_J(\boldsymbol{x}_N)A_N}_{\text{DNI}} + \underbrace{\boldsymbol{B}_{1I}^{\nabla^T}(\boldsymbol{x}_N)\boldsymbol{C}\boldsymbol{B}_{1J}^{\nabla}(\boldsymbol{x}_N)M_{1N} + \boldsymbol{B}_{2I}^{\nabla^T}(\boldsymbol{x}_N)\boldsymbol{C}\boldsymbol{B}_{2J}^{\nabla}(\boldsymbol{x}_N)M_{2N}}_{\text{stabilization}} \right) \tag{43}$$

Accordingly, the DNI- and SCNI-based naturally stabilized formulations in Eq. (43) and in Eq. (40) are denoted as NSDNI and NSCNI, respectively, which will be examined and compared in detail in Section 4.

3 Computer Implementation Aspects

In this section, the basic data structures and main functions of RKPM2D will be explained, which cover domain discretization, RK shape functions and gradients, domain integration, assembly of stiffness matrices and force vectors, and visualization of the numerical results. Also, the key differences between RKPM and FEM programs are discussed. All the source code and sample scripts (.m files) can be downloaded online.

3.1 Overall program structure

The general flowchart of RKPM2D is given in Figure 8. Unlike in the FEM procedures where the element type is tied with mesh/nodes generation, the order of basis and smoothness is independent of the domain discretization in RKPM, while the general program functionalities (such as matrix assembly and solver) of a meshfree Galerkin method are similar to that of FEM.

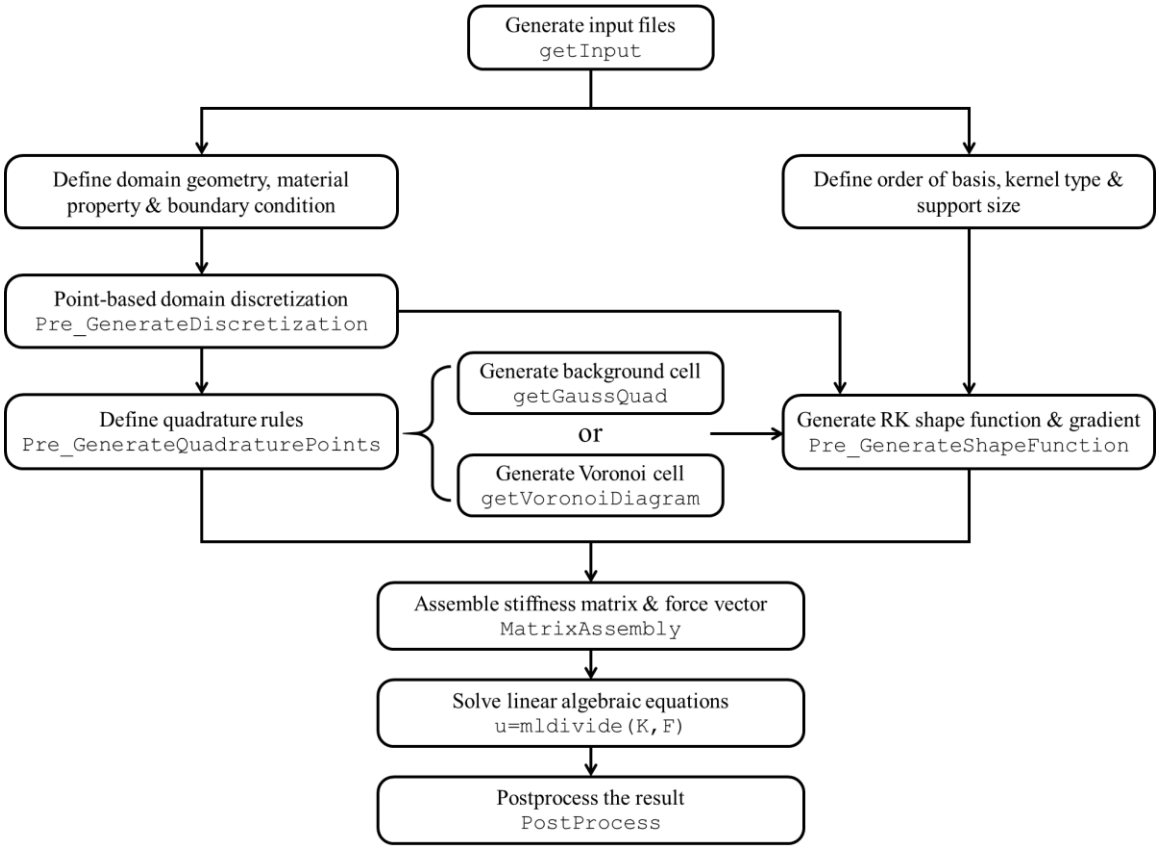


Figure 8. Flowchart of the RKPM procedure.

As shown in Figure 8, the numerical procedures for the model problem described in Eqns. (9) - (22) consists of input file generation, domain discretization, quadrature rule definition, shape function construction, matrix assembly, solver, and post-processing. The corresponding main subroutine names of RKPM2D are also shown in Figure 8.

3.2 Input file generation

To illustrate the functionality of each main subroutine shown in Figure 8, let us consider a linear elasticity problem (Eq. (9)) with a manufactured solution of

$$\boldsymbol{u}^{exact} = \begin{bmatrix} 0.1 + 0.1x_1 + 0.2x_2 \\ 0.05 + 0.15x_1 + 0.1x_2 \end{bmatrix} \tag{44}$$

which can be considered as a linear patch test, and is defined here over the 2D circular domain $\Omega \subset \mathbb{R}^2$ shown in Figure 9.

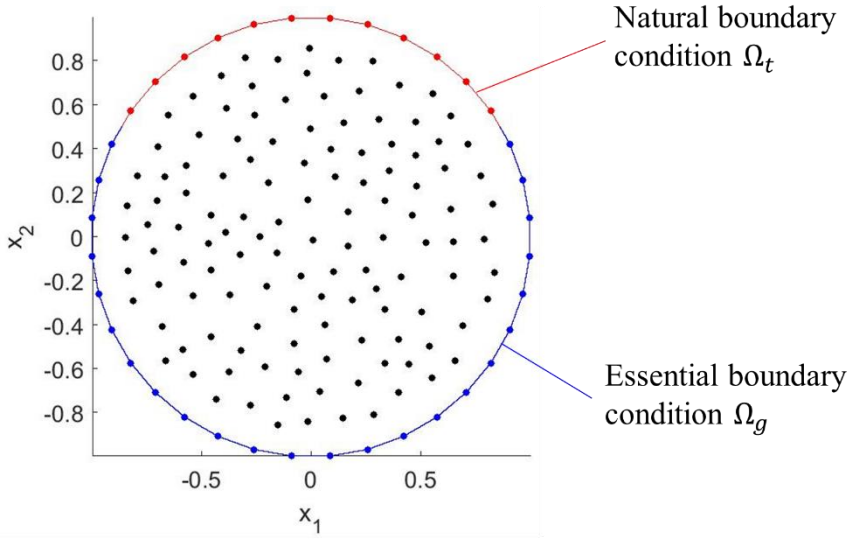


Figure 9. Domain discretization for model problem in 2D circular domain Ω.

The traction $\boldsymbol{t}$, body force $\boldsymbol{b}$, displacement $\boldsymbol{g}$ are prescribed based on the exact solution $\boldsymbol{u}^{exact}$ in Eq. (44):

$$\boldsymbol{t} = \boldsymbol{\eta}^T \boldsymbol{C} \boldsymbol{\varepsilon}^{exact}, \quad \forall \boldsymbol{x} \in \partial\Omega_t \tag{45}$$

$$\boldsymbol{g} = \boldsymbol{u}^{exact}, \quad \forall \boldsymbol{x} \in \partial\Omega_g \tag{46}$$

$$\boldsymbol{b} = \begin{bmatrix} \sigma_{11,1}^{exact} + \sigma_{12,2}^{exact} \\ \sigma_{21,1}^{exact} + \sigma_{22,2}^{exact} \end{bmatrix}, \quad \forall \boldsymbol{x} \in \Omega \tag{47}$$

where $\boldsymbol{\varepsilon}^{exact}$ and $\boldsymbol{\sigma}^{exact}$ are exact strain and stress defined as:

$$\boldsymbol{\varepsilon}^{exact} = \begin{bmatrix} \varepsilon_{11}^{exact} \\ \varepsilon_{22}^{exact} \\ 2\varepsilon_{12}^{exact} \end{bmatrix} = \begin{bmatrix} u_{1,1}^{exact} \\ u_{2,2}^{exact} \\ u_{1,2}^{exact} + u_{2,1}^{exact} \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.1 \\ 0.35 \end{bmatrix} \tag{48}$$

$$\boldsymbol{\sigma}^{exact} = \boldsymbol{C} \boldsymbol{\varepsilon}^{exact}, \qquad \boldsymbol{\eta} = \begin{bmatrix} n_1 & 0 \\ 0 & n_2 \\ n_2 & n_1 \end{bmatrix} \tag{49}$$

where $\boldsymbol{\eta}$ is the matrix of outward unit normal vector of the boundary, $\boldsymbol{C}$ is the matrix of elastic tensor with Young's modulus $E = 2.1 \times 10^{11}$ and Poisson's ratio $\nu = 0.3$. The traction $\boldsymbol{t}$ is imposed on $\partial\Omega_t$: $(x_1, x_2) \in \partial\Omega, x_2 > 0.5$, essential boundary conditions $\boldsymbol{g}$ is enforced on $\partial\Omega_g$: $(x_1, x_2) \in \partial\Omega, x_2 \leq 0.5$, and the body force is $\boldsymbol{b} = \boldsymbol{0}$ in this case. Accordingly, the input file for this problem is created by the function `getInput` with three data structures: `RK`, `Quadrature`, and `Model`, which define the RK shape functions, quadrature rules, numerical parameters (such as penalty parameters, elastic modulus, Poisson ratio, etc.), respectively. A sample input file for the abovementioned linear patch test is given in Listing 1, where `RK` contains the following fields:

- `KernelFunction`: kernel functions with different levels of continuity.
- `KernelGeometry`: the nodal support shape where "`CIR`" and "`REC`" represent circular and rectangular support, respectively.
- `NormalizedSupportSize`: normalized support size $\tilde{c}$.
- `Order`: the order of basis (constant, linear, or quadratic).

```
function [RK,Quadrature,Model] = getInput()
%% INPUT FILE
% Sample Input File for Patch Test
%% (1) Material
% Linear Elasticity
% Lame Parameters for Young's modulus and Poisson ratio
Model.E = 2.1E11; Model.nu = 0.3;
Model.Condition = 'PlaneStress'; % PlaneStress, or PlaneStrain
Model.ElasticTensor =
getElasticTensor(Model.E,Model.nu,Model.Condition);
Model.DOFu = 2;                  % two dimensional problem
%% (2) Geometry
theta = [0:pi/18:2*pi-pi/18]'; r = ones(size(theta));
[x1_vertices,x2_vertices] = pol2cart(theta,r);
% ensure the boundary segments to be counter clockwise
[x1_vertices, x2_vertices] = poly2ccw(x1_vertices, x2_vertices);
Model.xVertices = [x1_vertices, x2_vertices];
Model.DomainArea = polyarea(x1_vertices,x2_vertices);
%% (3) Boundary condition
% If an edge is not specified, natural BC with zero traction is
imposed.
Model.CriteriaEBC = @(x1,x2) find(x2<=0.5); % user input
Model.CriteriaNBC = @(x1,x2) find(x2>0.5);  % user input
% beta parameter for Nitches Method
Model.Beta_Nor = 1E2;
% For verification purpose, provide the exact displacement solution
syms x1 x2  % use x1 and x2 as x- & y- coordinates exact solution
Model.ExactSolution.u_exact =[0.1 + 0.1*x1 + 0.2*x2;
                              0.05 + 0.15*x1 + 0.1*x2;];
[Model.ExactSolution.S,...
 Model.ExactSolution.g,...
 Model.ExactSolution.t,...
 Model.ExactSolution.b] = getBoundaryConditions(Model);
 Model.ExactSolution.Exist = 1;
%% (4) Discretization Method,
% (...A) MATLAB built-in FE mesh generator: Default
Model.Discretization.Method = 'A';
Model.Discretization.Hmax = 0.1; % max nodal distance
%% (5) RK shape function parameter
RK.KernelFunction = 'SPLIN3';       % SPLIN3
RK.KernelGeometry = 'CIR';          % CIR, REC
RK.NormalizedSupportSize = 2.01;    % suggested order n + 1;
RK.Order = 'Linear';                % Constant, Linear, Quadratic
%% (6) Quadrature rule
Quadrature.Integration = 'SCNI';       % GAUSS, SCNI, DNI
Quadrature.Stabilization = 'N';        % M, N, WO
Quadrature.Option_BCintegration = 'NODAL'; % NODAL OR GAUSS
Quadrature.nGaussPoints = 6; % #Gauss Points per cell
Quadrature.nGaussCells = 10; % #GaussCells on the short side of domain
end
```

Listing 1. Input files of the linear patch test.

Note that the RK shape functions are computed at the beginning of the simulation based on the information in the structure `RK`. The variable names for kernel functions with different levels of continuity are listed in Table 1

Table 1. The abbreviation of the kernel functions used in the code

Name	Continuity	Represented Kernel Function
`HVSIDE`	C^{-1}	Heaviside
`SPLINE1`	C^{0}	Linear B-Spline (tent)
`SPLINE2`	C^{1}	Quadratic B-Spline
`SPLINE3`	C^{2}	Cubic B-Spline
`SPLINE4`	C^{3}	Quartic B-Spline
`SPLINE5`	C^{4}	Quintic B-Spline

In Listing 1, `Quadrature` contains the following fields:

- `Integration`: basic quadrature rules, where "`DNI`", "`SCNI`", and "`GAUSS`" options are provided.
- `Stabilization`: types of stabilization for nodal integration, where symbol "`N`" represents naturally stabilized nodal integration and symbol "`M`" represents modified nodal integration as described in Section 2.3.3.
- `Option_BCintegration`: quadrature rules for boundary integrals, including "`NODAL`" and "`GAUSS`" options.
- `nGaussPoints`: the number of Gauss points $N_g \in \mathbb{N}$ in each background integration cell. E.g. $N_g = 6$ denotes 6×6 Gauss points in each cell
- `nGaussCells`: the parameter determines the number of background integration cells along x_1 or x_2 direction of the problem domain, depending on the problem domain dimension.

Note that the `nGaussPoints` and `nGaussCells` are used only for Gauss integration. In addition, `Model` contains the following fields:

- `E`, `nu`: Young's modulus E, Poisson ratio ν.
- `DOFu`: the number of nodal degrees of freedom, `DOFu=2` for 2D elasticity.
- `BetaNormalized`: normalized penalty parameter.
- `xVertices`: physical coordinates of integration cells.
- `CriteriaEBC`: function handle to define the essential boundaries.
- `CriteriaNBC`: function handle to define the natural boundaries.

To implement the traction $\boldsymbol{t}$, body force $\boldsymbol{b}$, displacement $\boldsymbol{g}$ based on the exact solution $\boldsymbol{u}^{exact}$ in Eqns. (45) - (49), and the switch matrix $\boldsymbol{S}$, the symbolic computation in MATLAB is employed to obtain the expression of $\boldsymbol{t}$, $\boldsymbol{b}$, $\boldsymbol{g}$ and $\boldsymbol{S}$. As shown in Listing 2, for any given exact displacement field $\boldsymbol{u}^{exact}(\boldsymbol{x})$ in the symbolic form, $\boldsymbol{t}$, $\boldsymbol{b}$, $\boldsymbol{g}$ and $\boldsymbol{S}$ based on Eqns. (45) - (49) are obtained from the function `Pre_GenerateBoundaryConditions` with $\boldsymbol{u}^{exact}$ as an input variable, as shown in Listing 3. Then, these variables are saved as function handles under the structure `Model.ExactSolution` by `matlabFunction` command (note: a function handle is a data type in MATLAB to store an association to a function). Alternatively, if the exact solution $\boldsymbol{u}^{exact}$ is not specified, then one can define traction $\boldsymbol{t}$, body force $\boldsymbol{b}$, displacement $\boldsymbol{g}$, and switch matrix $\boldsymbol{S}$ in individual functions, as shown in Listing 4. The structure `Model.ExactSolution` contains the following fields:

- `t`: return the traction $\boldsymbol{t}$.
- `b`: return the body force vector $\boldsymbol{b}$.
- `g`: return the imposed displacement $\boldsymbol{g}$.
- `S`: return the switch matrix $\boldsymbol{S}$.

```
% give the exact solution of the displacement
syms x1 x2 % please use x1 and x2 as coordinates
Model.ExactSolution.u_exact =[0.1 + 0.1*x1 + 0.2*x2;
                              0.05 + 0.15*x1 + 0.1*x2;];
% give the expression of function handle of Switch S, essential
% boundary conditions g, traction t, and body force b
if isfield(Model,'ExactSolution') % if given analytical displacement
    [Model.ExactSolution.S,Model.ExactSolution.g,...
     Model.ExactSolution.t,Model.ExactSolution.b] = ...
    getBoundaryConditions(Model);
else % if S, g, t, b are defined in functions
    Model.ExactSolution.S = @getSebc; % function getSebc
    Model.ExactSolution.g = @getGebc; % function getGebc
    Model.ExactSolution.t = @getTraction; % function getTraction
    Model.ExactSolution.b = @getBodyForce; % function getBodyForce
end
```

Listing 2. Command lines of defining boundary conditions by providing an analytical expression of displacement $\boldsymbol{u}_{exact}$ or defining imposed traction $\boldsymbol{t}$, body force $\boldsymbol{b}$, displacement $\boldsymbol{g}$, and switch matrix $\boldsymbol{S}$.

```
function [function_S,function_g,function_traction,function_b] =
getBoundaryConditions(Model)
syms x1 x2 n1 n2
% function handle for essential boundary condition g
u = Model.ExactSolution.u_exact;
C = Model.ElasticTensor;
function_g = matlabFunction(u);
% function handle for stress
epsilon_x1 = diff(u(1),x1);
epsilon_x2 = diff(u(2),x2);
epsilon_x12 = (diff(u(1),x2)+diff(u(2),x1));
stress = C*[epsilon_x1;epsilon_x2;epsilon_x12];
% function handle for traction t
eta = [n1 0 n2; 0 n2 n1;];
traction = eta*stress;
function_traction = matlabFunction(traction,'Vars',[x1 x2 n1 n2]);
% function handle for body force b
b = [divergence([stress(1),stress(3)],[x1,x2]);
     divergence([stress(3),stress(2)],[x1,x2]);];
function_b = matlabFunction(b,'Vars',[x1 x2]);
% function handle for switch S
function_S = matlabFunction(sym(diag([1 1])),'Vars',[x1 x2]);
end
```

Listing 3. Command lines of the function that generates the exact traction $\boldsymbol{t}$, body force $\boldsymbol{b}$, imposed displacement $\boldsymbol{g}$, and switch matrix $\boldsymbol{S}$ through MATLAB symbolic operation.

```
function [ t ] = getTraction(x1,x2,n1,n2)
% Input: x1,x2: Cartesian coordinate
% Output: t: a 2 by 1 vector for the traction
E = 2.1E11; nu = 0.3;
Condition = 'PlainStress'; % PlaneStress, or PlaneStrain
C = getElasticTensor(E,nu,Condition);
Strain_exact = [0.1; 0.1; 0.35];
stress = (C*Strain_exact);
eta = [n1 0 n2; 0 n2 n1;];
t = eta*stress;
end

function [ SEBC ] = getSebc(x1,x2)
% Input:  x1,x2: coordinates in 1,2
% Output: SEBC: a 2 by 2 matrix for the switch matrix on EBC
SEBC = diag([1 1]);
End

function [ gEBC ] = getGebc(x1,x2)
% Input:   x1,x2: Cartesian coordinate
% Output:  gEBC: a 2 by 1 vector of prescribed displacement on EBC
gEBC =[0.1 + 0.1*x1 + 0.2*x2;
       0.05 + 0.15*x1 + 0.1*x2];
End

function [ b ] = getBodyForce(x1,x2)
% Input:  x1,x2: Cartesian coordinate
% Output: b: a 2 by 1 vector for the body force
b = [0; 0;];
end
```

Listing 4. Functions that define imposed traction $\boldsymbol{t}$, body force $\boldsymbol{b}$, displacement $\boldsymbol{g}$, and switch matrix $\boldsymbol{S}$.

3.3 Domain discretization

After the definition of basic model input parameters (`RK`, `Quadrature`, `Model`), the function `Pre_GenerateDiscretization(Model)` is called for domain discretization. `Pre_GenerateDiscretization` will return the structure `Discretization` which consists of the following fields:

- `xI_Boundary`: coordinates of the boundary RK nodes
- `xI_Interior`: coordinates of the interior RK nodes
- `nP`: the total number of discretized RK nodes, NP.

As shown in the command lines in Listing 5, the MATLAB built-in function `geometryFromEdges` and `generateMesh` are employed to decompose the domain

into conforming sub-domains (that can be considered as an FEM mesh), and the resulting vertices are directly employed as meshfree nodes.

```
% create PDE model object for MATLAB default mesh generator
% for more information, please refer to
%https://www.mathworks.com/help/pde/ug/pde.pdemodel.geometryfromedges.
html?s_tid=doc_ta
model = createpde;
% read in the domain vertices from Model of getInput
x1_vertices = Model.xVertices(:,1);
x2_vertices = Model.xVertices(:,2);
% Create polygon based on edge of the vertices
R1 = [3,length(x1_vertices),x1_vertices',x2_vertices']';
% gm is the geometry based on vertices,
% sf is a set formula created to define if there is any subtraction
% between geometry, eg sf = R1-C1 where C1 may be a circle
gm = [R1]; sf = 'R1';
% create geometry
ns = char('R1'); ns = ns';
% Decompose constructive geometry into minimal regions
g = decsg(gm,sf,ns);
% create geometry for Model object
geoModel = geometryFromEdges(model,g);
% generate FE mesh for Model by MATLAB
FEmesh = generateMesh(model);
% Use FE mesh nodes as RK nodes.
RKnodes = FEmesh.Nodes';
```

Listing 5. Command lines of domain discretization.

Alternatively, users of RKPM2D can also define the meshfree nodes manually or importing an FEM mesh from CAD/FEM software instead of using the MATALB built-in meshing function. Such a subroutine, `sub_ReadNeutralInputFiles`, is given in Listing 6 which reads in nodal coordinates from a Patran neutral file.

```
function [xI] = sub_ReadNeutralInputFiles(NeutralFileName)
% Read-in the CAD/FEA neutral file in Patran neutral format
% e.g., NeutralFileName = 'FE_Neutral.dat';
%
filename = fullfile(NeutralFileName);  % open the full neutral file
T = readtable(filename); % read in table format in MATLAB
C = table2cell(T);       % convert table format to cell format
% Convert cell format to double precision to obtain coordinates
nLine = length(C); % number of lines in neutral file
C_double = cellfun(@str2num,C,'UniformOutput',false);
% Read discretization information
LineOfC = C_double{2};
NNODE = LineOfC(5); % number of nodes
NELEM = LineOfC(6); % number of elements
xI = zeros(NNODE,2); % nodal coordinates initialization
disp(['From ',filename,' file: #node is ',num2str(NNODE),', #element
is ',num2str(NELEM)])
% Read the file
iL = 2; idx_node = 1;
while (iL <= nLine) % read line by line from the input files
      LineMatrix = C_double{iL};
      if ~isempty(LineMatrix) % no empty line is read
      IDCARD = LineMatrix(1);
      %% IDCARD = 01 is the coordinates list; Read coordinates
      if (IDCARD == 1) && length(LineMatrix) == 9
      iL = iL + 1;  % read next line
      xI(idx_node,1:2) = C_double{iL}(1:2);
      idx_node = idx_node + 1; % next node
      iL = iL + 1;  % next line
      end
      end % end if ~isempty
      iL = iL + 1; % next line
end % end of reading each line
end % end of the function
```

Listing 6. Command lines of reading nodal coordinates from a Patran neutral file.

3.4 Quadrature point generation

The quadrature points are generated in the function `Pre_GenerateQuadraturePoints` which support the following two domain integration schemes:

- Nodal integration, where the Voronoi cells are defined.
- Gauss integration, where the background integration cells are defined.

Standard Gauss integration cell and the corresponding coordinates and weights of Gauss points are employed for Gauss integration. As for nodal integration, the quadrature points and weights are based on Voronoi cells. The construction of Voronoi diagram is achieved by employing the function `getVoronoiDiagram`, which is modified from the open-

source code `sub_VoronoiLimit` [46]. By calling `getVoronoiDiagram`, the structure `VoronoiDiagram` which contains the following fields will be returned:

- `VerticeCoordinates`: all vertices' coordinates of Voronoi cells.
- `VoronoiCell`: indexes of vertices within each Voronoi cell.

The `VoronoiDiagram` structure defines the quadrature rules required for nodal integration as described in Eqs. (26) and (27). By looping over the Voronoi cells, the following two classes are added by `Pre_GenerateQuadraturePoints` into the structure `Quadrature` which defines the quadrature rules to compute the RK shape functions for domain and boundary integration:

- `Domain`: class that defines the variables required for domain integration.
  - `nQuad`: the total number of quadrature points.
  - `xQuad`: the coordinates of quadrature points.
  - `Weight`: quadrature weights for domain integral.
- `BC`: class that defines the variables required for boundary integration.
  - `nQuad_onBoundary`: the number of quadrature points on the boundary.
  - `xQuad_onBoundary`: coordinates of quadrature points on the boundary.
  - `Weight_onBoundary`: quadrature weights for contour integral.
  - `Normal_onBoundary`: outward unit normal vectors at quadrature points along the boundary.

3.5 RK shape function generation

In FE programing, double loops are required to evaluate the stiffness matrix and force vector, including one loop over all elements and another loop over all quadrature points within each element. In RKPM, only one loop over all quadrature points is required if a nodal integration method is used. As shown in Figure 10, at each quadrature point, the RK shape functions and their derivatives $\Psi_I(\boldsymbol{x}_N)$, $\Psi_{I,1}(\boldsymbol{x}_N)$, and $\Psi_{I,2}(\boldsymbol{x}_N)$ (and/or smooth and implicit gradients given in Eq. (29) and (38)) are evaluated in the physical domain. In contrast, FEM shape functions and derivatives constructed in the parametric domain and require an isoparametric mapping process which may introduce large errors when the elements are severely distorted.

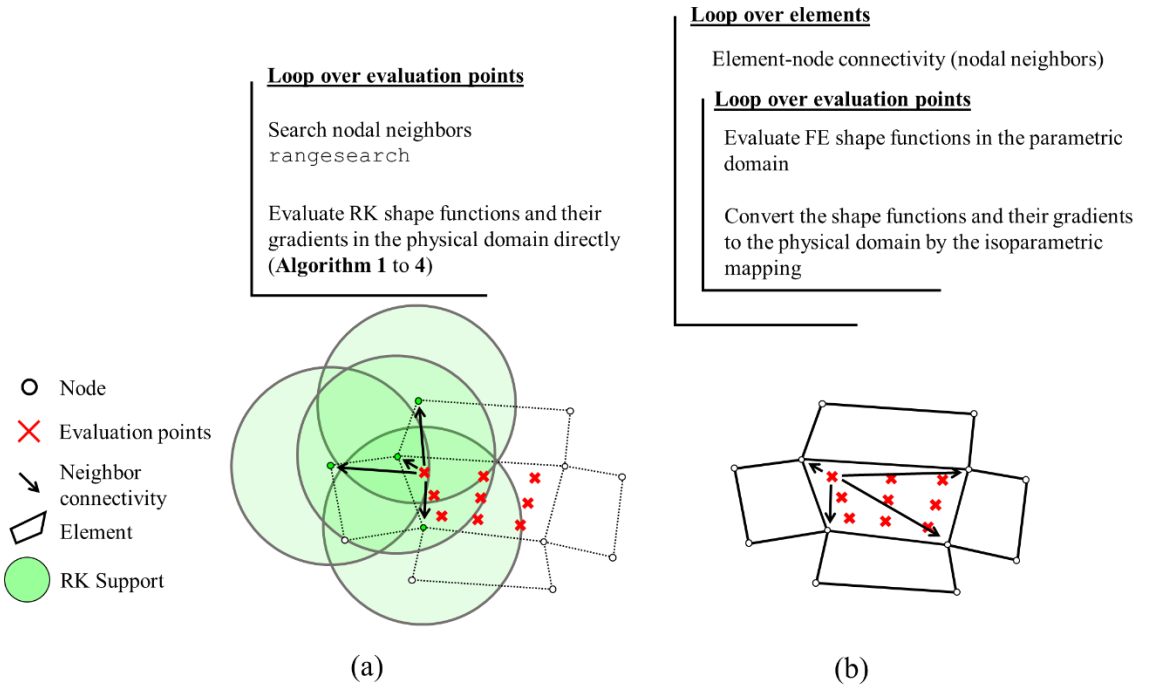


Figure 10. Flowchart of the algorithm in (a) RK and (b) FE shape function generation.

Another difference in computing RK and FE shape functions is the determination of nodal neighbors. FEM relies on the use of an element mesh to define the shape functions and thus the neighbors for a given evaluation point are simply the nodes defining the element. In contrast, in RKPM the shape functions are defined directly at nodes without the element connectivity. Consequently, a neighbor search is necessary to determine the neighbors of a given evaluation point in RKPM. The neighbor search algorithm could be CPU intensive, especially for 3D simulations, so efficient spatial search algorithms such as KD-Tree [47], DESS [48], among others [32, 33, 34, 35] have been employed. In this study, the efficient search algorithms `KDTreeSearcher` [49] and `rangesearch` function under `KDTreeSearcher` are employed. The command line for constructing the neighbor list is:

```
[NeighborList] = rangesearch(xI,x,SupportSize,'Distance','euclidean');
```

Here the inputs of `rangesearch` are explained as follows

- `xI`: coordinates of RK nodes $\boldsymbol{x}_I$.

- `x`: coordinates of evaluation points $\boldsymbol{x}$.
- `SupportSize`: the support size a_I.

Evaluation of RK shape functions with respect to RK nodes $\boldsymbol{x}_I$ at an evaluation point $\boldsymbol{x}$ is performed in `getRKShapeFunction`, for which the program structure is illustrated in **Algorithm 1** to **Algorithm 4**. Note that, in the evaluation of the derivatives of kernel functions in **Algorithm 3**, the derivative of the distance $z_{I,i} = \frac{(x_i - x_{iI})}{(z_I a_I^2)}$ becomes singular when the evaluation point $\boldsymbol{x}$ approaches node $\boldsymbol{x}_I$ (i.e., $z_I = \|\boldsymbol{x} - \boldsymbol{x}_I\| \approx 0$). This is avoided by setting $z_I = 0, \boldsymbol{\nabla} z_I = \mathbf{0}$ if $\|\boldsymbol{x} - \boldsymbol{x}_I\| < \text{eps}$, where eps is the default positive machine precision number in MATLAB, which works well for symmetric and smooth kernels. Alternatively, we can set $z_{I,1} = \frac{(x_1 - x_{1I})}{(z_I a_I^2 + \text{eps})}$, to keep the denominator of $z_{I,1}$ positive (as with $z_{I,2}$). The latter approach is implemented in RKPM2D.

Algorithm 1: RK shape functions $\Psi_I(\boldsymbol{x})$ evaluation

function $\left[\Psi_I, \Psi_{I,1}, \Psi_{I,2}\right] = getRKShapeFunction(\mathbf{RK}, \{\boldsymbol{x}_I\}_1^{NP}, \boldsymbol{x})$
  {%} find $\boldsymbol{x} \in supp(\{\boldsymbol{x}_I\}_1^{NP})$, order & kernel support size
  $n \leftarrow \mathbf{RK}.$Order; $\{a_I\}_1^{NP} \leftarrow \mathbf{RK}.\mathbf{SupportSize}$;
  Neighbor List: $G_x \leftarrow rangesearch(\{\boldsymbol{x}_I\}_1^{NP}, \boldsymbol{x}, a)$
  $\boldsymbol{M} = \mathbf{0};\ \boldsymbol{M}_{,1} = \mathbf{0};\ \boldsymbol{M}_{,2} = \mathbf{0}$ {%} Initialize the moment matrix
  for $I \in G_x$ **do**
    $\left[\boldsymbol{H}, \boldsymbol{H}_{,1}, \boldsymbol{H}_{,2}\right] = Basis(n, \boldsymbol{x}_I, \boldsymbol{x})$ {%} Eq. (3)
    $\left[\Phi_a, \Phi_{a,1}, \Phi_{a,2}\right] = Kernel(a_I, \boldsymbol{x}_I, \boldsymbol{x})$ {%} Eq. (7)
    $\left[\boldsymbol{M}, \boldsymbol{M}_{,1}, \boldsymbol{M}_{,2}\right]$ {%} Eq. (4), sum moment matrix
    $= Moment\left(\boldsymbol{M}, \boldsymbol{M}_{,1}, \boldsymbol{M}_{,2}, \boldsymbol{H}, \boldsymbol{H}_{,1}, \boldsymbol{H}_{,2}, \Phi_a, \Phi_{a,1}, \Phi_{a,2}\right)$
  end for
  $\{\Psi_I\}_1^{NP} = \boldsymbol{H}_0^T \boldsymbol{M}^{-1} \boldsymbol{H} \Phi_a$ {%} Eq. (2)
  $\boldsymbol{M}_{,1}^{-1} = -\boldsymbol{M}^{-1} \boldsymbol{M}_{,1} \boldsymbol{M}^{-1};\ \boldsymbol{M}_{,2}^{-1} = -\boldsymbol{M}^{-1} \boldsymbol{M}_{,2} \boldsymbol{M}^{-1};$
  $\left\{\Psi_{I,1}\right\}_1^{NP} = \boldsymbol{H}_0^T \boldsymbol{M}_{,1}^{-1} \boldsymbol{H} \Phi_a + \boldsymbol{H}_0^T \boldsymbol{M}^{-1} \boldsymbol{H}_{,1} \Phi_a + \boldsymbol{H}_0^T \boldsymbol{M}^{-1} \boldsymbol{H} \Phi_{a,1}$
  $\left\{\Psi_{I,2}\right\}_1^{NP} = \boldsymbol{H}_0^T \boldsymbol{M}_{,2}^{-1} \boldsymbol{H} \Phi_a + \boldsymbol{H}_0^T \boldsymbol{M}^{-1} \boldsymbol{H}_{,2} \Phi_a + \boldsymbol{H}_0^T \boldsymbol{M}^{-1} \boldsymbol{H} \Phi_{a,2}$
end function

Algorithm 2: Basis vector computation

function $\left[\boldsymbol{H}, \boldsymbol{H}_{,1}, \boldsymbol{H}_{,2}\right] = Basis(n, \boldsymbol{x}_I, \boldsymbol{x})$
  $x_1 = \boldsymbol{x}(1);\ x_2 = \boldsymbol{x}(2)$
  $x_{1I} = \boldsymbol{x}_I(1);\ x_{2I} = \boldsymbol{x}_I(2)$
  $count = 1$
  for $k = 0{:}\, n$ **do**
    for $j = 0{:}\, k$ **do**
      $i = k - j$
      $\boldsymbol{H}(count, I) = (x_1 - x_{1I})^i (x_2 - x_{2I})^j$
      $\boldsymbol{H}_{,1}(count, I) = i(x_1 - x_{1I})^{i-1} (x_2 - x_{2I})^j$
      $\boldsymbol{H}_{,2}(count, I) = j(x_1 - x_{1I})^i (x_2 - x_{2I})^{j-1}$
      $count = count + 1$
    end for
  end for
end function

Algorithm 3: Kernel function evaluation (Cubic B-spline with circular support)

function $[\Phi_a, \Phi_{a,1}, \Phi_{a,2}] = Kernel(a_I, \boldsymbol{x}_I, \boldsymbol{x})$
    {%} different types of kernels can be implemented in a similar manner
    $z_I = \frac{\|\boldsymbol{x}-\boldsymbol{x}_I\|}{a_I};\ z_{I,1} = \frac{(x_1-x_{1I})}{(z_I a_I^2)};\ z_{I,2} = \frac{(x_2-x_{2I})}{(z_I a_I^2)}$
    if $0 \le z_I < \frac{1}{2}$ **then**
        $\Phi_a = \frac{2}{3} - 4z_I^2 + 4z_I^3;\ \Phi_{a,z} = -8z_I + 12z_I^2$
    elseif ½ $\le z_I < 1$ **then**
        $\Phi_a = \frac{4}{3} - 4z_I + 4z_I^2 - \frac{4z_I^3}{3};\ \Phi_{a,z} = -4 + 8z_I - 4z_I^2$
    else
        $\Phi_a = 0;\ \Phi_{a,z} = 0$
    end if
    $\Phi_{a,1} = \Phi_{a,z} z_{,1}$
    $\Phi_{a,2} = \Phi_{a,z} z_{,2}$
end function

Algorithm 4: Moment matrix computation

function $[\boldsymbol{M}, \boldsymbol{M}_{,1}, \boldsymbol{M}_{,2}] = Moment(\boldsymbol{M}, \boldsymbol{M}_{,1}, \boldsymbol{M}_{,2}, \boldsymbol{H}, \boldsymbol{H}_{,1}, \boldsymbol{H}_{,2}, \Phi_a, \Phi_{a,1}, \Phi_{a,2})$
    $\boldsymbol{M} = \boldsymbol{M} + \boldsymbol{H}\boldsymbol{H}^T \Phi_a(\boldsymbol{x})$
    $\boldsymbol{M}_{,1} = \boldsymbol{M}_{,1} + \boldsymbol{H}_{,1}\boldsymbol{H}^T \Phi_a + \boldsymbol{H}\boldsymbol{H}_{,1}^T \Phi_a + \boldsymbol{H}\boldsymbol{H}^T \Phi_{a,1}$
    $\boldsymbol{M}_{,2} = \boldsymbol{M}_{,2} + \boldsymbol{H}_{,2}\boldsymbol{H}^T \Phi_a + \boldsymbol{H}\boldsymbol{H}_{,2}^T \Phi_a + \boldsymbol{H}\boldsymbol{H}^T \Phi_{a,2}$
end function

The evaluations of direct derivatives $\Psi_{I,1}(\boldsymbol{x}_N)$ and $\Psi_{I,2}(\boldsymbol{x}_N)$ for DNI and GI at quadrature points $\boldsymbol{x}_N$ are straightforward as the direct derivatives can be computed by **Algorithm 1** to **Algorithm 4**. In SCNI, the direct derivatives are replaced with smoothed derivatives $\widetilde{\Psi}_{I,1}(\boldsymbol{x}_N)$ and $\widetilde{\Psi}_{I,2}(\boldsymbol{x}_N)$, for which the smoothing procedure is given in **Algorithm 5**. **VoronoiCell**$\{I\}(K)$ denotes a cell structure for Voronoi cells to define the K^{th} index of cell vertices for the I^{th} Voronoi cell, and **Vertices** is a vector that defines the Cartesian coordinates of Voronoi cell vertices. The smoothing process in **Algorithm 5** is computed by the function `getSmoothedDerivative` as given in the Listing 7. For given inputs of a smoothing cell vertices' coordinates $\{\boldsymbol{x}_v\}_1^{NV}$ (denoted as "`xV`" in the Listing 7) and nodal coordinates $\boldsymbol{x}_I$ (denoted as "`xI`" in the Listing 7), the function

`getSmoothedDerivative` gives the corresponding smoothed derivatives $\widetilde{\Psi}_{I,1}(\boldsymbol{x}_N)$, $\widetilde{\Psi}_{I,2}(\boldsymbol{x}_N)$ and the smoothing domain area A_N.

Algorithm 5: SCNI smoothed derivative	
1.	**for** $N = 1{:}\,NP$ **do** {%} Loop over quadrature points (nodal points)
2.	$NV \leftarrow \mathbf{VoronoiCell}\{N\}(:)$ {%} Number of cell edge for Voronoi cell N
3.	$\{\boldsymbol{x}_v\}_1^{NV} \leftarrow \mathbf{Vertices}\big(\mathbf{VoronoiCell}\{N\}(:)\big)$ {%} Voronoi vertices coordinates
4.	$A_N \leftarrow \{\boldsymbol{x}_v\}_1^{NV}$ {%} Calculate domain area of Voronoi cell N
5.	$\boldsymbol{x}_I \leftarrow$ Discretization {%} RK nodal points
6.	$\Psi_{I,i} = \mathbf{0}$; {%} initialization for shape function evaluated at smoothing point
7.	**for** $K = 1{:}\,NV$ **do** {%} Loop over cell edges of Voronoi cell N
8.	find Voronoi cell edge quadrature points $\widetilde{\boldsymbol{x}}_N^K$ from $\{\boldsymbol{x}_v\}_K^{K+1}$
9.	find Voronoi cell edge normal n_{iK} from $\{\boldsymbol{x}_v\}_K^{K+1}$
10.	find Voronoi cell edge length L_K from $\{\boldsymbol{x}_v\}_K^{K+1}$
11.	$[\Psi_I(\widetilde{\boldsymbol{x}}_N^K)] = getRKShapeFunction(\mathbf{RK}, \{\boldsymbol{x}_I\}_1^{NP}, \widetilde{\boldsymbol{x}}_N^K)$
12.	$\Psi_{I,i} = \Psi_{I,i} + \Psi_I(\widetilde{\boldsymbol{x}}_N^K) n_{iK} L_K$
13.	**end for**
14.	$\widetilde{\Psi}_{I,i}(\boldsymbol{x}_N) = \left(\frac{1}{A_N}\right) \Psi_{I,i}$ {%} Smoothed derivative, Eq. (29)
15.	**end for**

```
function [SHPDX1_smoothed,SHPDX2_smoothed,Area_Cell] =
getSmoothedDerivative(RK,xI,xV)
% Input : xI, RK nodes; xV, vertices of the cell;
% Output: Smoothed Derivative SHPDX1/2 and cell area
% smoothing point sequence for SCNI cell
nP = length(xI); % number of node
nV = length(xV); % number of vertices
% obtain the area of each voronoi cell
Area_Cell = polyarea(xV(:,1),xV(:,2))+eps;
% initiate shape function at the smoothed points
SHP_Smoothed_local = sparse(2,nP);
for k = 0:nV-1 % loop over each edge for each cell
    % find the two ends of the edge
    if k == 0
        Vertex1 = xV(end,:);
    else
        Vertex1 = xV(k,:);
    end
    Vertex2 = xV(k+1,:);
    % Cell edge length Lk_Cell and normal Nk_Cell
    Lk_Cell = norm(Vertex2-Vertex1,2);
    xv21 = Vertex2 - Vertex1;
    xv21_Normal = xv21*[cos(pi/2) -sin(pi/2); sin(pi/2) cos(pi/2)];
    % calculate the unit normal, eps is machine precision
    Nk_Cell = xv21_Normal/(norm(xv21_Normal,2)+eps);
    if norm(Lk_Cell,2) < eps || norm(Nk_Cell,2) < eps
        Lk_Cell = eps; Nk_Cell = [eps eps];
    end
    % Smoothing Points of each segment
    xtilde_CellEdge = (Vertex1 + Vertex2)/2;
    % RK shape function evaluation at smoothing point
    [SHP_xtilde] = ...
    getRKShapeFunction(RK,xI,xtilde_CellEdge(1:2),[1,0,0]);
    SHP_Smoothed_local = SHP_Smoothed_local + ...
    (Nk_Cell'*SHP_xtilde)*Lk_Cell;
end % end of each cell boundary
% evaluate the smoothed derivative
SHPDX1_smoothed = (1/Area_Cell)*SHP_Smoothed_local(1,:);
SHPDX2_smoothed = (1/Area_Cell)*SHP_Smoothed_local(2,:);
end
```

Listing 7. Command lines for smoothing procedure in each cell.

The computed shape functions and their derivatives are stored in the structure `Quadrature` as:

- `SHP`: matrix of size `nQuad`×`nP` that stores the shape functions $\Psi_I(\boldsymbol{x}_N)$ of all nodes.
- `nQuad`: number of quadrature points for domain integration, the value $= NG$ for Gauss integration and $= NP$ for nodal integration

- `SHPDX1`: matrix of size `nQuad×nP` that stores the shape function derivatives $\Psi_{I,1}(\boldsymbol{x}_N)$ or $\widetilde{\Psi}_{I,1}(\boldsymbol{x}_N)$ of all nodes.
- `SHPDX2`: matrix of size `nQuad×nP` that stores the shape function derivatives $\Psi_{I,2}(\boldsymbol{x}_N)$ or $\widetilde{\Psi}_{I,2}(\boldsymbol{x}_N)$ of all nodes.

The shape functions can be evaluated at the boundary in a similar manner as described in **Algorithm 5**.

3.6 Stabilization Methods

The stabilization terms associated with M- and N-type stabilization method are computed under the function `Pre_GenerateShapeFunction` where RK shape functions and direct/smoothed gradients are also evaluated. For the M-type stabilization method discussed in Section 2.3.3.1, the nodal representative domain is divided into sub-cells by using the MATLAB built-in function `delaunayTriangulation`, which divides the polygon Voronoi domain into several triangular sub-cells. The procedures for generating the sub-cells and associated evaluation points for the additional stabilization terms are shown in Listing 8. By looping over each sub-cell, the shape function gradients with associated integration weights (i.e., the area of each sub-cell) are computed. At each sub-cell, the function `getSmoothedDerivative` in Listing 7 is employed to evaluate the smoothed gradient to construct the M-type stabilization terms in (Eq. (35)). An alternative way is to evaluate direct shape function gradient (Eq. (36)) by **Algorithm 1** to **Algorithm 4** at the centroid of each sub-cell for the M-type stabilization, which is adopted in Section 4 for DNI-based methods.

```
% M-type Stabilization by subdivision of integration cell
xVerices_MSCNI = unique([xV; xQuad(idx_nQuad,:)],'rows');
% use delauny triangle as subcell
DT_MSCNI = delaunayTriangulation(xVerices_MSCNI(:,1),...
                                 xVerices_MSCNI(:,2));
[N_subcell,~] = size(xNs_MSCNI);
for s = 1 : N_subcell % loop over the sub cell
% vertices coordinates of the sub cell
xV_SubCell = DT_MSCNI.Points(DT_MSCNI.ConnectivityList(s,:),:);
% evaluate area of the sub cell
AREA_Is{idx_nQuad}(s,1) = polyarea(xV_SubCell(:,1),xV_SubCell(:,2));
end
```

Listing 8. Command lines for sub-division of gradient smoothing cells for the M-type stabilization.

Once all stabilization terms are computed, they are stored under the structure `Quadrature.MtypeStabilization` with the following fields:

- `nS`: number of sub-cells NS for the I^{th} Voronoi cell.
- `SHPDX1_Is`: cell structure contains matrix of size `nS×nP` that stores the shape function derivatives $\Psi_{I,1}(\hat{\boldsymbol{x}}_N^S)$ or $\widetilde{\Psi}_{I,1}(\hat{\boldsymbol{x}}_N^S)$ in the I^{th} Voronoi cell
- `SHPDX2_Is`: cell structure contains matrix of size `nS×nP` that stores the shape function derivatives $\Psi_{I,2}(\hat{\boldsymbol{x}}_N^S)$ or $\widetilde{\Psi}_{I,2}(\hat{\boldsymbol{x}}_N^S)$ in the I^{th} Voronoi cell
- `AREA_Is`: cell structure that stores area of the s^{th} sub-cell associated with the I^{th} Voronoi cell

Unlike the M-type stabilization method, the subdivision of the Voronoi diagrams is not required in the N-type stabilization method discussed in Section 2.3.3.2. The second-order shape function derivatives are achieved by taking direct derivatives of the first-order implicit gradients, which can be easily achieved by replacing the $\boldsymbol{H}_0^T$ in **Algorithm 1** with $\boldsymbol{H}_1^T$ and $\boldsymbol{H}_2^T$ shown in Eq. (39). The corresponding output $\Psi_{I,1}(\boldsymbol{x})$ and $\Psi_{I,2}(\boldsymbol{x})$ would become $\Psi_{I1,1}^{\nabla}$ and $\Psi_{I1,2}^{\nabla}$ when $\boldsymbol{H}_1^T$ is used (or, $\Psi_{I2,1}^{\nabla}$ and $\Psi_{I2,2}^{\nabla}$ when $\boldsymbol{H}_2^T$ is used). The second moments of inertia M_{1N}, M_{2N} in each nodal integration domain are computed straightforwardly by the formula in [50] from the Voronoi vertices' coordinates as shown in Listing 9.

```
% NSNI stabilization for calculating moment inertia
for idx_nQuad = 1:nQuad % loop over quadrature points
% xQuad(idx_nQuad,:) is quadrature points (nodal points)
% Evaluation of second order gradient of shape function
IG = 1; % where 'IG' is the switch to turn on/off implicit gradient
[~,~,~,SHPDX1X1(idx_nQuad,:),SHPDX2X2(idx_nQuad,:),SHPDX1X2(idx_nQuad,
:)] = getRKShapeFunction(RK,xI,xQuad(idx_nQuad,:),IG);
% normalized the vertices coordinates by mean coordinates
Vertices1 = xV - ones(nV,1)*mean(xV);
Vertices2 = circshift(xV - ones(nV,1)*mean(xV),[-1 0]);
% using formula to calculate inertia for cell by vertices
M1N(idx_nQuad) =
(1/12)*abs(sum((Vertices1(:,2).^2+Vertices1(:,2).*Vertices2(:,2)+Verti
ces2(:,2).^2).*(Vertices1(:,1).*Vertices2(:,2) -
Vertices2(:,1).*Vertices1(:,2))));
M2N(idx_nQuad) =
(1/12)*abs(sum((Vertices1(:,1).^2+Vertices1(:,1).*Vertices2(:,1)+Verti
ces2(:,1).^2).*(Vertices1(:,1).*Vertices2(:,2) -
Vertices2(:,1).*Vertices1(:,2))));
% Parallel axis theorem for shift it back to global coordinates
dx1 = sqrt((xQuad(idx_nQuad,1)-mean(xV(:,1))).^2);
dx2 = sqrt((xQuad(idx_nQuad,2)-mean(xV(:,2))).^2);
M1N(idx_nQuad) = M1N(idx_nQuad) + Area_VoronoiCell(idx_nQuad)*dx1^2;
M2N(idx_nQuad) = M2N(idx_nQuad) + Area_VoronoiCell(idx_nQuad)*dx2^2;
end
```

Listing 9. Command lines for evaluating $\Psi^{\nabla}_{I1,1}, \Psi^{\nabla}_{I1,2}, \Psi^{\nabla}_{I2,2}\ M_{1N}, M_{2N}$ in N-type stabilization for each node.

Once all stabilization terms are computed, they are stored under the structure `Quadrature.NtypeStabilization` with the following fields:

- `SHPDX1X1`: matrix of size `nP×nP` that stores the shape function second derivatives $\Psi^{\nabla}_{I1,1}$ of all nodes.
- `SHPDX1X2`: matrix of size `nP×nP` that stores the shape function second derivatives $\Psi^{\nabla}_{I1,2}$ of all nodes.
- `SHPDX2X2`: matrix of size `nP×nP` that stores the shape function second derivatives $\Psi^{\nabla}_{I2,2}$ of all nodes.
- `M`: second moments of inertia M_{1N}, M_{2N} in each nodal integration domain.

3.7 Matrix Evaluation and Assembly

Once RK shape functions are computed, the function `MatrixAssmebly` is called to evaluate and assemble the stiffness matrix and force vector. With the utilization of MATLAB built-in sparse matrix data structure [51], the sparse storage scheme is employed for matrix evaluation and assembly in order to reduce the memory requirements. Other sparse storage schemes such as banded [52], skyline [53], compress row [54], that avoid operations on zero elements can also be adopted. The program structure of `MatrixAssmebly` is given in **Algorithm 6**.

Algorithm 6: Stiffness matrix and force vector assembly

function $\left[\boldsymbol{K}_{IJ}, \boldsymbol{F}_I\right] = MatrixAssembly(\textbf{Quadrature}, \textbf{Model})$
  {%} initialization of stiffness matrix and force vector
  $\boldsymbol{K}_{IJ}^{c}, \boldsymbol{K}_{IJ}^{\beta}, \boldsymbol{K}_{IJ}^{g} = \boldsymbol{0}$; $\boldsymbol{F}_I^{b}, \boldsymbol{F}_I^{t}, \boldsymbol{F}_I^{\beta}, \boldsymbol{F}_{IJ}^{g} = \boldsymbol{0}$;
  $\boldsymbol{C}, \beta \leftarrow \textbf{Model}$; {%} elastic tensor and penalty parameter
  {%} assemble the stiffness matrix and body force vector
  $nQuad = \textbf{Quadrature}.\textbf{Domain}.$nQuad {%} # of quadrature points
  for $N = 1{:}\,nQuad$ **do**
    $\boldsymbol{x}_N, W_N, \boldsymbol{B}_I, \boldsymbol{\Psi}_I \leftarrow \textbf{Quadrature}.\textbf{Domain}$
    $\boldsymbol{K}_{IJ}^{c} = \boldsymbol{K}_{IJ}^{c} + \boldsymbol{B}_I^T \boldsymbol{C} \boldsymbol{B}_J W_N$ {%} tangent stiffness by Eq. (24)/(26)/(31)
    $\boldsymbol{K}_{IJ}^{c} = \boldsymbol{K}_{IJ}^{c} + \boldsymbol{K}_{IJ}^{stab}$
    {%} $\boldsymbol{K}_{IJ}^{stab}$: additional stabilization in Eq. (35)/(36), (40)/(43)
    $\boldsymbol{b} \leftarrow getBodyForce(\boldsymbol{x}_N)$ {%} obtain body force $\boldsymbol{b}$
    $\boldsymbol{F}_I^{b} = \boldsymbol{F}_I^{b} + \boldsymbol{\Psi}_I \boldsymbol{b} W_N$ {%} body force by Eq. (24)/(26)
  end for
  {%} boundary integration for natural and essential boundary conditions
  $nQuad_{BC} = \textbf{Quadrature}.\textbf{BC}.\textbf{nQuad}_{\textbf{onBoundary}}$
  for $N = 1{:}\,nQuad_{BC}$ **do**
    $\boldsymbol{x}_N, W_N, \boldsymbol{\eta}, \boldsymbol{B}_I, \boldsymbol{\Psi}_I \leftarrow \textbf{Quadrature}.\textbf{BC}$
    if $\boldsymbol{x}_N \in Essential\ Boundary\ Conditions$ **then**
      $\boldsymbol{g} \leftarrow getGEBC(\boldsymbol{x}_N)$ {%} obtain prescribed displacement $\boldsymbol{g}$
      $\boldsymbol{S} \leftarrow getSEBC(\boldsymbol{x}_N)$ {%} obtain switch matrix $\boldsymbol{S}$
      {%} Eqns. (16) and (17) by (25)/(27)/(32)(33)
      $\boldsymbol{K}_{IJ}^{\beta} = \boldsymbol{K}_{IJ}^{\beta} + \beta \boldsymbol{\Psi}_I \boldsymbol{S} \boldsymbol{\Psi}_J W_N$
      $\boldsymbol{K}_{IJ}^{g} = \boldsymbol{K}_{IJ}^{g} + \boldsymbol{B}_I^T \boldsymbol{C} \boldsymbol{\eta} \boldsymbol{S} \boldsymbol{\Psi}_J W_N$
      $\boldsymbol{F}_I^{\beta} = \boldsymbol{F}_I^{\beta} + \beta \boldsymbol{\Psi}_I \boldsymbol{S} \boldsymbol{g} W_N$
      $\boldsymbol{F}_I^{g} = \boldsymbol{F}_I^{g} + \boldsymbol{B}_I^T \boldsymbol{C} \boldsymbol{\eta} \boldsymbol{S} \boldsymbol{g} W_N$
    elseif $\boldsymbol{x}_N \in Natural\ Boundary\ Conditions$ **then**
      $\boldsymbol{t} \leftarrow getTraction(\boldsymbol{x}_N)$ {%} obtain surface traction $\boldsymbol{t}$
      $\boldsymbol{F}_I^{t} = \boldsymbol{F}_I^{t} + \boldsymbol{\Psi}_I \boldsymbol{t} W_N$
    end if
  end for
  $\boldsymbol{K}_{IJ} = \boldsymbol{K}_{IJ}^{c} + \boldsymbol{K}_{IJ}^{\beta} - \left(\boldsymbol{K}_{IJ}^{g} + \boldsymbol{K}_{IJ}^{g\,T}\right)$; $\boldsymbol{F}_I = \boldsymbol{F}_I^{b} + \boldsymbol{F}_I^{t} + \boldsymbol{F}_I^{\beta} - \boldsymbol{F}_{IJ}^{g}$;
end function

As shown in the **Algorithm 6**, the assembly of the stiffness matrix $\boldsymbol{K}_{IJ}^{c}$ and body force vector $\boldsymbol{F}_{I}^{b}$ are performed, and the MATLAB command lines are given in Listing 10.

```
DOFu = Model.DOFu; % load degrees of freedom
% B matrix and Psi matrix initiation
B = sparse(3,nP*DOFu) ;
PSI = sparse(2,nP*DOFu) ;
% Load shape function from Quadrature
SHP = Quadrature.SHP;
SHPDX1 = Quadrature.SHPDX1;
SHPDX2 = Quadrature.SHPDX2;
for idx_quad = 1:nQuad % loop over quadrature points
 B(:,:) = 0; PSI(:,:) = 0;
 % Load B and Psi matrix with shape function and their derivative
 B(1,1:2:end) = SHPDX1(idx_nQuad,:);
 B(2,2:2:end) = SHPDX2(idx_nQuad,:);
 B(3,1:2:end) = SHPDX2(idx_nQuad,:);
 B(3,2:2:end) = SHPDX1(idx_nQuad,:);
 PSI(1,1:2:end) = SHP(idx_nQuad,:);
 PSI(2,2:2:end) = SHP(idx_nQuad,:);
 % Stiffness matrix assembly
 KIJ_c = KIJ_c + (B')*(C)*B*(Weight(idx_nQuad,1));
 % Obtain the body force at quadrature point's coordinates
 b = f_b(xQuad(idx_nQuad,1),xQuad(idx_nQuad,2));
 % Body force vector assembly
 FI_b = FI_b + (PSI')*b*(Weight(idx_nQuad,1));
end
```

Listing 10. Command lines of stiffness matrix $\boldsymbol{K}_{IJ}^{c}$ and body force vector $\boldsymbol{F}_{I}^{b}$ assembly for elasticity problems.

The command lines for the assembly of the traction force $\boldsymbol{F}_{I}^{t}$ as well as the Nitsche's term, $\boldsymbol{K}_{IJ}^{\beta}, \boldsymbol{K}_{IJ}^{g}, \boldsymbol{F}_{I}^{\beta}, \boldsymbol{F}_{IJ}^{g}$, in **Algorithm 6** are given in Listing 11.

```
nP = Quadrature.nP; % number of nodal points
DOFu = Model.DOFu; % degrees of freedom of u
% Load Model, Young's modulus
E = Model.E;
% Load function handle for evaluating S,g,t
f_S = Model.ExactSolution.S;
f_g = Model.ExactSolution.g;
f_t = Model.ExactSolution.t;
% Beta Parameter for Nitsche's Method
beta = Model.Beta_Nor*E/sqrt((Model.DomainArea/nP)); % Penalty Number
% Load information required for boundary integration
nQuad_BC = Quadrature.BC.nQuad_onBoundary;
xQuad_BC = Quadrature.BC.xQuad_onBoundary;
Weight_BC = Quadrature.BC.Weight_onBoundary;
Normal_BC = Quadrature.BC.Normal_onBoundary;
SHP_onBC = Quadrature.BC.SHP_BC;
SHPDX1_onBC = Quadrature.BC.SHPDX1_BC;
SHPDX2_onBC = Quadrature.BC.SHPDX2_BC;
for idx_nQuad = 1:nQuad_BC % for loop of quadrature points at boundary
 % normal at quadrature points
 n1 = Normal_BC(idx_nQuad,1);
 n2 = Normal_BC(idx_nQuad,2);
 % PSI matrix at quadrature points
 PSI(:,:) = 0;
 PSI(1,1:2:end) = SHP_onBC(idx_nQuad,:);
 PSI(2,2:2:end) = SHP_onBC(idx_nQuad,:);
 % switch to different boundary conditions
 switch Quadrature.BC.EBCtype{idx_nQuad}
  case {'NBC'}
  % surface traction
  t = f_t(xQuad_BC(idx_nQuad,1),xQuad_BC(idx_nQuad,2),n1,n2);
  FI_t = FI_t + PSI'*t*Weight_BC(idx_nQuad);
  case {'EBC'}
  % Surface normal Eta
  ETA = [n1 0 n2; 0 n2 n1]';
  % Load B Matrix
  B(:,:) = 0;
  B(1,1:2:end) = SHPDX1_onBC(idx_nQuad,:);
  B(2,2:2:end) = SHPDX2_onBC(idx_nQuad,:);
  B(3,1:2:end) = SHPDX2_onBC(idx_nQuad,:);
  B(3,2:2:end) = SHPDX1_onBC(idx_nQuad,:);
  % Essential boundary condition g
  g = f_g(xQuad_BC(idx_nQuad,1),xQuad_BC(idx_nQuad,2));
  % Switch s
  S = f_S(xQuad_BC(idx_nQuad,1),xQuad_BC(idx_nQuad,2));
  % Nitche's term
  KIJ_g = KIJ_g + (PSI'*S'*ETA*C*B)*Weight_BC(idx_nQuad);
  FI_g = FI_g + (B'*C*ETA*S*g)*Weight_BC(idx_nQuad);
  KIJ_beta = KIJ_beta + beta*(PSI'*S*PSI)*Weight_BC(idx_nQuad);
  FI_beta = FI_beta + beta*(PSI'*S*g)*Weight_BC(idx_nQuad);
 end % end swtich different boundary conditions
end % end for loop of quadrature points at boundary
```

Listing 11. Command lines of assembly of the traction force and the Nitsche's term for elasticity problems.

As seen from **Algorithm 6**, Listing 10 and Listing 11, the function `MatrixAssembly` only involves shape functions and quadrature rules under the data structure `Quadrature`, whereas the material parameters and evaluation of traction $\boldsymbol{t}$, body force $\boldsymbol{b}$, essential boundary conditions $\boldsymbol{g}$ and switch $\boldsymbol{S}$ are obtained under the data structure `Model`. After the assembly, the resultant system of linear equations $\sum_J \boldsymbol{K}_{IJ}\boldsymbol{u}_J = \boldsymbol{F}_I$ can be solved by either direct methods such as LU factorization, Gauss elimination or non-stationary iterative methods such as PCG, GMRES, and BICGSTAB [55]. Many of these methods are available in public domain or open-source software and can be readily integrated or linked into this RKPM2D. Here, the MATLAB function `mldivide` is adopted, which can take advantage of matrix symmetries and automatically assign an appropriate matrix solver.

Remark: Marginal changes in inputs from `Quadrature` and `Model` and slight modifications in Listing 10 and Listing 11 can be performed to convert the code to solve a different type of equation. For demonstration, an example of modifying the routine in **Algorithm 6** to solve a diffusion equation is given in the appendix.

3.8 Post-processing

The final step of the program is to compute and visualize the computed displacement, strain and stress fields with the use of the routine `PostProcess`. The following fields are computed and returned:

- `uhI`: matrix of size $NP \times 2$ that defines the displacements at RK nodes $\boldsymbol{u}^h(\boldsymbol{x}_I)$.
- `Strain`: double vector of size $NP \times 3$ that defines the strain at RK nodes $\boldsymbol{\varepsilon}(\boldsymbol{x}_I) = [\varepsilon_{11}(\boldsymbol{x}_I), \varepsilon_{22}(\boldsymbol{x}_I), 2\varepsilon_{12}(\boldsymbol{x}_I)]$.
- `Stress`: double vector of size $NP \times 3$ that defines the stress at RK nodes $\boldsymbol{\sigma}(\boldsymbol{x}_I) = [\sigma_{11}(\boldsymbol{x}_I), \sigma_{22}(\boldsymbol{x}_I), 2\sigma_{12}(\boldsymbol{x}_I)]$.

For simplicity, the continuous fields of displacement, strain, and stress are plotted based on Delaunay triangulation of the whole domain by using the MATALB built-in functions `delaunayTriangulation` and `trisurf` as shown in Listing 12.

```
% Using delaunay Triangulation to post processing the scattered data
% Index_BC is the index number of node on the boundary
tri = delaunayTriangulation(xI(:,1),xI(:,2) ,...
                   [Discretization.Index_BC',...
                    circshift(Discretization.Index_BC',[-1 0])]);
% plot u1 displacement
figure, trisurf(tri(tri.isInterior(),:), xI(:,1),xI(:,2),uhI(:,1));
```

Listing 12. Parts of the subroutine for Delaunay triangulation

4 Numerical Examples

In this section, benchmark numerical examples are analyzed to examine the performance of the open-source code RKPM2D. The reproducing kernel approximation with linear basis and cubic B-spline kernel is adopted, for which circular support with a normalized support size $\tilde{c} = 2.0$ is employed. The normalized penalty parameter for Nitsche's method is chosen to be $\beta_{nor} = 100$. In the examples, the following domain integration methods are analyzed and compared:

(1). GI: Gauss Integration in Eqns. (24) - (25), where different integration orders are considered.
(2). DNI: Direct Nodal Integration in Eqns. (26) - (27)
(3). MDNI: Modified DNI formulation in Eq. (36)
(4). NDNI: Naturally Stabilized DNI formulation in Eq. (43)
(5). SCNI: Stabilized Conforming Nodal Integration in Eqns. (31) - (33)
(6). MSCNI: Modified SCNI formulation in Eq. (35)
(7). NSCNI: Naturally Stabilized SCNI in Eq. (40)

To access the accuracy of different numerical schemes, the displacement and energy norms are defined as follows:

$$\|\boldsymbol{u}-\boldsymbol{u}^h\|_{L_2} = \sqrt{\frac{\int_\Omega \left[\left(\boldsymbol{u}^h(\boldsymbol{x})-\boldsymbol{u}^{exact}(\boldsymbol{x})\right)^T\left(\boldsymbol{u}^h(\boldsymbol{x})-\boldsymbol{u}^{exact}(\boldsymbol{x})\right)\right]d\Omega}{\int_\Omega \left[\boldsymbol{u}^{exact}(\boldsymbol{x})^T\boldsymbol{u}^{exact}(\boldsymbol{x})\right]d\Omega}}, \tag{50}$$

$$\|\boldsymbol{u}-\boldsymbol{u}^h\|_{E} = \sqrt{\frac{\int_\Omega \left[\left(\boldsymbol{\varepsilon}^h(\boldsymbol{x})-\boldsymbol{\varepsilon}^{exact}(\boldsymbol{x})\right)^T\left(\boldsymbol{\sigma}^h(\boldsymbol{x})-\boldsymbol{\sigma}^{exact}(\boldsymbol{x})\right)\right]d\Omega}{\int_\Omega \left[\boldsymbol{\varepsilon}^{exact}(\boldsymbol{x})^T\boldsymbol{\sigma}^{exact}(\boldsymbol{x})\right]d\Omega}} \tag{51}$$

in which the superscript 'h' and 'exact' denote the numerical and exact solutions, respectively.

For domain integration involved in Eqns. (50) and (51), a high-order Gauss integration is employed

$$\|\boldsymbol{u}-\boldsymbol{u}^h\|_{L_2} \approx \sqrt{\frac{\sum_{N=1}^{NGg}\left(\boldsymbol{u}_N^h-\boldsymbol{u}_N^{exact}\right)^T\left(\boldsymbol{u}_N^h-\boldsymbol{u}_N^{exact}\right)W_N}{\sum_{N=1}^{NGg}\boldsymbol{u}_N^{exact^T}\boldsymbol{u}_N^{exact}W_N}}, \tag{52}$$

$$\|\boldsymbol{u}-\boldsymbol{u}^h\|_E \approx \sqrt{\frac{\sum_{N=1}^{NGg}\left(\boldsymbol{\varepsilon}_N^h-\boldsymbol{\varepsilon}_N^{exact}\right)^T\left(\boldsymbol{\sigma}_N^h-\boldsymbol{\sigma}_N^{exact}\right)W_N}{\sum_{N=1}^{NGg}\boldsymbol{\varepsilon}_N^{exact^T}\boldsymbol{\sigma}_N^{exact}W_N}} \tag{53}$$

where $\boldsymbol{u}_N^h=\boldsymbol{u}^h(\boldsymbol{x}_N)$, $\boldsymbol{u}_N^{exact}=\boldsymbol{u}^{exact}(\boldsymbol{x}_N)$, NGg is the total number of Gauss points and W_N is the weight for the Gauss points. The number of integration cells for error evaluation is set equal to the total number of nodes, and 10×10 Gauss points per integration cell is used, so the total number of quadrature points for error norm calculation is $NGg=100\times NP$ in Eqns. (52) and (53).

4.1 Patch test

In the first example, the linear patch test is analyzed to verify the accuracy of RKPM2D using linear basis in the RK approximation. The elasticity equation in (9) is considered with the exact solution defined as a linear polynomial function:

$$\boldsymbol{u}^{exact}=\begin{bmatrix}0.1+0.1x_1+0.2x_2\\0.05+0.15x_1+0.1x_2\end{bmatrix} \tag{54}$$

Accordingly, the traction $\boldsymbol{t}=\boldsymbol{\eta}^T\boldsymbol{C}\boldsymbol{\varepsilon}^{exact}$ is imposed on $\partial\Omega_t\colon(x_1,x_2)\in\partial\Omega, x_2>0.5$, where $\boldsymbol{\eta}$ is the collection of outward unit normal vector of the boundary surface, $\boldsymbol{C}$ is the matrix of elastic moduli with Young's modulus $E=2.1\times 10^{11}$ and Poisson's ratio $\nu=0.3$, $\boldsymbol{\varepsilon}^{exact}$ is the exact strain with $\boldsymbol{\varepsilon}^{exact}=[0.1,\ 0.1,\ 0.35]^T$; $\boldsymbol{g}=\boldsymbol{u}^{exact}$ is enforced on $\partial\Omega_g\colon(x_1,x_2)\in\partial\Omega, x_2\leq 0.5$, and the body force is set to $\boldsymbol{b}=\boldsymbol{0}$. Four geometries are considered, and their nodal discretizations are shown in Figure 11.

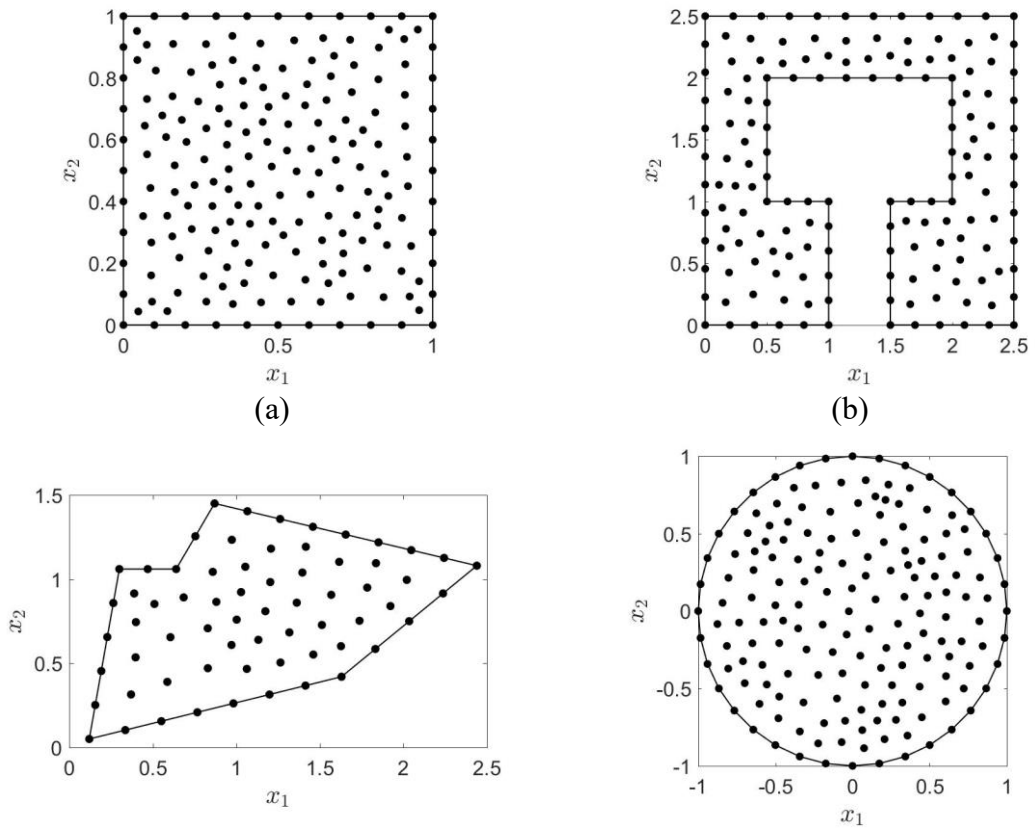


Figure 11. (a) Geometry 1, (b) geometry 2, (c) geometry 3, and (d) geometry 4 considered in the patch test.

When Gauss Integration (GI) is employed, uniform background integration cells with 1×1, 2×2, 4×4 or 6×6 Gauss points per cell is considered. The displacement and energy error norms (Eqns. (52) and (53)) using GI, DNI and SCNI are given in Table 2 and Table 3, respectively. As can be seen, while low-order GI diverges, high-order GI (6×6 integration points per cell) can achieve the L_2 errors around 10^{-4} but is computationally expensive, as will be shown later. For nodal integrations, DNI leads to very large errors, whereas SCNI exactly passes the patch test for all geometries.

Table 2. L_2 error norm $\|\boldsymbol{u}-\boldsymbol{u}^h\|_{L_2}$ in linear patch tests

Quadrature Scheme	Geometry 1	Geometry 2	Geometry 3	Geometry 4
GI (1×1)	1.2E-01	1.1E+02	1.4E+02	5.1E+04
GI (2×2)	2.4E-03	9.5E-02	2.5E-02	2.5E-02
GI (4×4)	1.4E-04	4.2E-02	4.0E-03	5.2E-03
GI (6×6)	2.6E-05	2.4E-02	3.7E-03	1.6E-03
DNI	4.3E-02	2.8E-02	1.5E-02	2.0E-01
MDNI	1.2E-02	1.1E-01	1.2E-01	2.0E-02
NDNI	7.2E-03	1.5E-02	9.7E-03	1.8E-02
SCNI	1.2E-14	4.4E-14	2.9E-15	7.1E-15
MSCNI	2.6E-15	2.1E-14	3.1E-15	2.5E-15
NSCNI	3.2E-15	1.1E-14	3.4E-15	2.8E-15

Table 3. Energy error norm $\|\boldsymbol{u}-\boldsymbol{u}^h\|_E$ in linear patch tests

Quadrature Scheme	Geometry 1	Geometry 2	Geometry 3	Geometry 4
GI (1×1)	3.8E+00	3.1E+03	5.8E+02	4.5E+05
GI (2×2)	2.6E-02	3.3E-01	6.01E-01	1.4E-01
GI (4×4)	1.6E-03	1.1E-01	4.0E-02	5.4E-02
GI (6×6)	3.0E-04	1.1E-01	3.5E-02	1.0E-02
DNI	2.1E-01	1.5E-01	1.4E-01	2.8E-01
MDNI	7.2E-02	3.2E-01	3.8E-01	7.0E-02
NDNI	4.5E-02	6.4E-02	8.8E-02	4.4E-02
SCNI	1.1E-13	2.8E-13	8.1E-15	2.3E-14
MSCNI	1.8E-14	6.2E-14	9.5E-15	8.6E-15
NSCNI	1.0E-13	1.1E-13	1.3E-14	1.9E-14

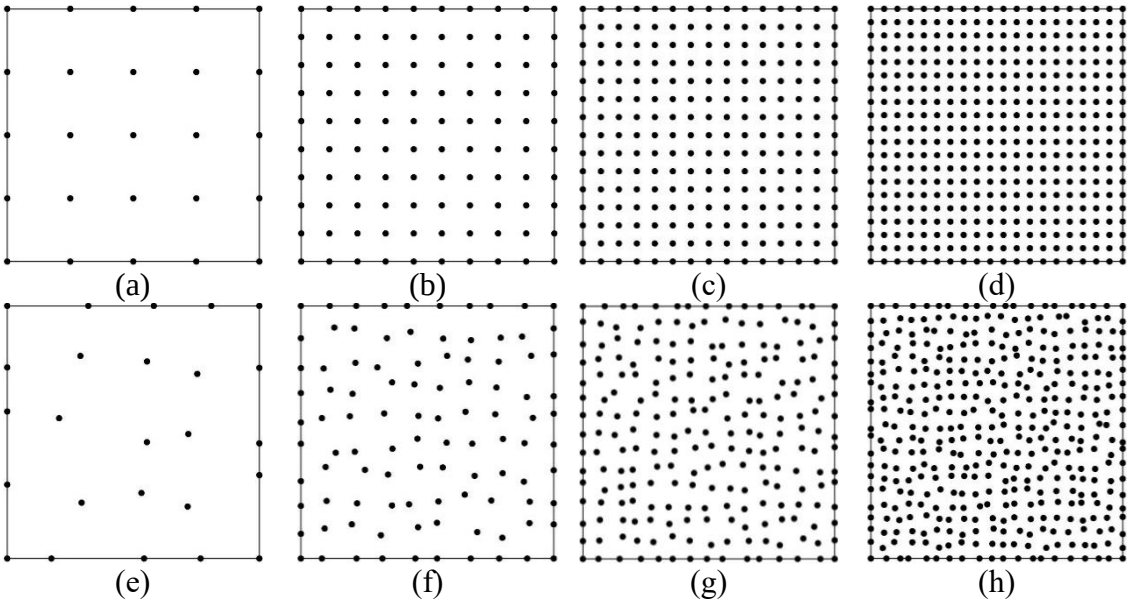


Figure 12. Uniform and non-uniform discretization of a square domain with (a) 5×5, (b) 10×10, (c) 15×15, and (d) 20×20 nodes, where the non-uniform discretizations from (e) to (h) are consisted of randomized nodal distribuitons that corespond to the uniform nodal distribuitons from (a) to (d), respectively.

Next, the solution accuracy with model discretization refinement under both uniform and non-uniform discretizations is examined. As shown in Figure 12, non-uniform discretizations are generated by introducing random numbers between $\pm 0.5h$ into the uniform discretizations, where h is the nodal distance in the uniform discretizations of a unit square. For GI, 4×4, 9×9, 14×14, 19×19 background integration cells are constructed based on uniform discretizations with 5×5, 10×10, 15×15, 20×20 nodes, respectively. As shown from Table 4 to Table 7, SCNI-based methods pass the patch test independent to the type of discretizations, whereas DNI-based methods and GI yield large errors. For instance, the displacement error norm is around 10^{-3} in DNI and 10^{-6} in GI 6×6 under uniform discretizations in Table 4, and under non-uniform discretizations in Table 6 the error norms increase up to 10^{-2} and around $10^{-5} \sim 10^{-6}$ in DNI and GI 6×6, respectively. Also, as shown in Table 5, the energy norm in uniform discretizations is around 10^{-2} in DNI and 10^{-5} in GI 6×6 and the error further increases in the non-uniform cases as shown in Table 7. On the other hand, the error norms in SCNI are always less than 10^{-12}. Furthermore, for nodal integrations with additional stabilizations (N-type and M-type), all SCNI-based formulations (MSCNI and NSCNI) pass the linear patch test,

while all DNI-based formulations fail even with the employment of additional stabilization techniques due to the violation of variational consistency conditions. To restore the variational consistency for DNI-based methods, the VCI approach [11] can be used, but is considered out of the scope of this paper since conforming smoothing (SCNI-based methods) can be employed to satisfy the conditions.

Table 4. L_2 error norm $\|\boldsymbol{u}-\boldsymbol{u}^h\|_{L_2}$ in linear patch tests (uniform discretization)

Quadrature Scheme	Refinement Level			
	5×5	10×10	15×15	20×20
GI (1×1)	8.9E-02	6.3E-02	3.4E-02	2.1E-02
GI (2×2)	1.1E-03	3.6E-04	1.9E-04	1.2E-04
GI (4×4)	2.3E-05	1.2E-05	5.9E-06	3.6E-06
GI (6×6)	1.2E-05	7.6E-06	4.3E-06	2.8E-06
DNI	1.7E-02	4.9E-03	2.3E-03	1.6E-03
MDNI	1.3E-02	3.1E-03	1.5E-03	9.5E-04
NDNI	2.0E-01	7.2E-02	3.7E-02	2.4E-02
SCNI	5.0E-15	5.0E-15	6.0E-15	5.7E-15
MSCNI	5.3E-15	3.1E-15	3.1E-15	6.5E-15
NSCNI	6.5E-15	3.8E-15	2.6E-15	4.7E-15

Table 5. Energy error norm $\|\boldsymbol{u}-\boldsymbol{u}^h\|_E$ in linear patch tests (uniform discretization)

Quadrature Scheme	Refinement Level			
	5×5	10×10	15×15	20×20
GI (1×1)	2.2E+00	4.1E+00	3.9E+00	3.2E+00
GI (2×2)	6.2E-03	5.9E-03	4.8E-03	4.2E-03
GI (4×4)	1.9E-04	1.3E-04	9.0E-05	7.2E-05
GI (6×6)	9.7E-05	7.5E-05	6.2E-05	5.3E-05
DNI	1.7E-01	1.1E-01	7.1E-02	6.8E-02
MDNI	1.1E-01	5.8E-02	4.1E-02	3.3E-02
NDNI	7.3E-01	3.8E-01	2.9E-01	2.5E-01
SCNI	4.2E-14	8.2E-14	1.1E-13	1.8E-13
MSCNI	3.2E-14	6.9E-14	6.7E-14	1.4E-13
NSCNI	2.0E-14	6.5E-14	6.8E-14	1.5E-13

Table 6. L_2 error norm $\|\boldsymbol{u}-\boldsymbol{u}^h\|_{L_2}$ in linear patch tests (non-uniform discretization)

Quadrature Scheme	Refinement Level			
	5×5	10×10	15×15	20×20
GI (1×1)	1.3E-01	1.6E-01	3.2E-02	1.2E-02
GI (2×2)	1.3E-03	1.1E-03	1.3E-03	9.5E-04
GI (4×4)	7.9E-05	5.4E-05	3.0E-05	2.1E-05
GI (6×6)	1.4E-05	9.9E-06	5.7E-06	4.1E-06
DNI	2.6E-02	1.9E-02	3.9E-02	3.1E-02
MDNI	1.7E-02	1.0E-02	1.5E-02	1.2E-02
NDNI	9.1E-02	3.2E-02	4.6E-02	2.9E-02
SCNI	9.3E-15	5.2E-15	1.0E-14	6.3E-15
MSCNI	9.7E-15	2.8E-15	5.1E-15	4.7E-15
NSCNI	8.7E-15	3.1E-15	6.7E-15	4.1E-15

Table 7. Energy error norm $\|\boldsymbol{u}-\boldsymbol{u}^h\|_E$ in linear patch tests (non-uniform discretization)

Quadrature Scheme	Refinement Level			
	5×5	10×10	15×15	20×20
GI (1×1)	2.8E+00	9.1E+00	1.9E+00	1.2E+00
GI (2×2)	1.4E-02	2.4E-02	4.0E-02	3.9E-02
GI (4×4)	9.3E-04	1.6E-03	1.3E-03	1.1E-03
GI (6×6)	1.4E-04	2.5E-04	2.6E-04	2.4E-04
DNI	2.6E-01	3.0E-01	5.2E-01	5.2E-01
MDNI	1.5E-01	1.4E-01	1.9E-01	1.6E-01
NDNI	4.5E-01	2.6E-01	2.9E-01	2.6E-01
SCNI	8.5E-14	1.3E-13	3.1E-13	3.6E-13
MSCNI	5.2E-14	4.9E-14	1.2E-13	1.4E-13
NSCNI	2.5E-14	4.8E-14	7.1E-14	1.1E-13

4.2 Cantilever beam problem

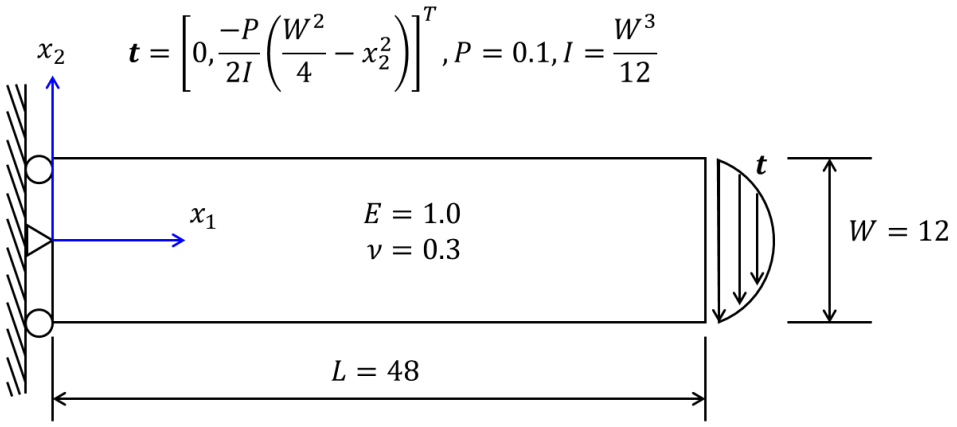


Figure 13. Problem setting of a cantilever beam under a shear force.

A cantilever beam problem shown in Figure 13 is considered next. The exact displacement solution to this problem is:

$$u_1^{exact} = \frac{Px_2}{6EI}\left[(6L - 3x_2)x_1 + (2+\nu)\left(x_2^2 - \frac{W^2}{4}\right)\right],$$
$$u_2^{exact} = \frac{-P}{6EI}\left[3\nu x_2^2(L - x_1) + (4+5\nu)\frac{W^2 x_1}{4} + (3L - x_1)x_1^2\right] \quad (55)$$

in which the Young's modulus $E = 1$ and Poisson ratio $\nu = 0.3$ is selected, and the geometry is shown in Figure 13. The exact displacement solution is prescribed on the left wall as the essential boundary condition, i.e., $\boldsymbol{g} = \boldsymbol{u}^{exact}$, and the corresponding traction is imposed on the right-side surface as the natural boundary condition.

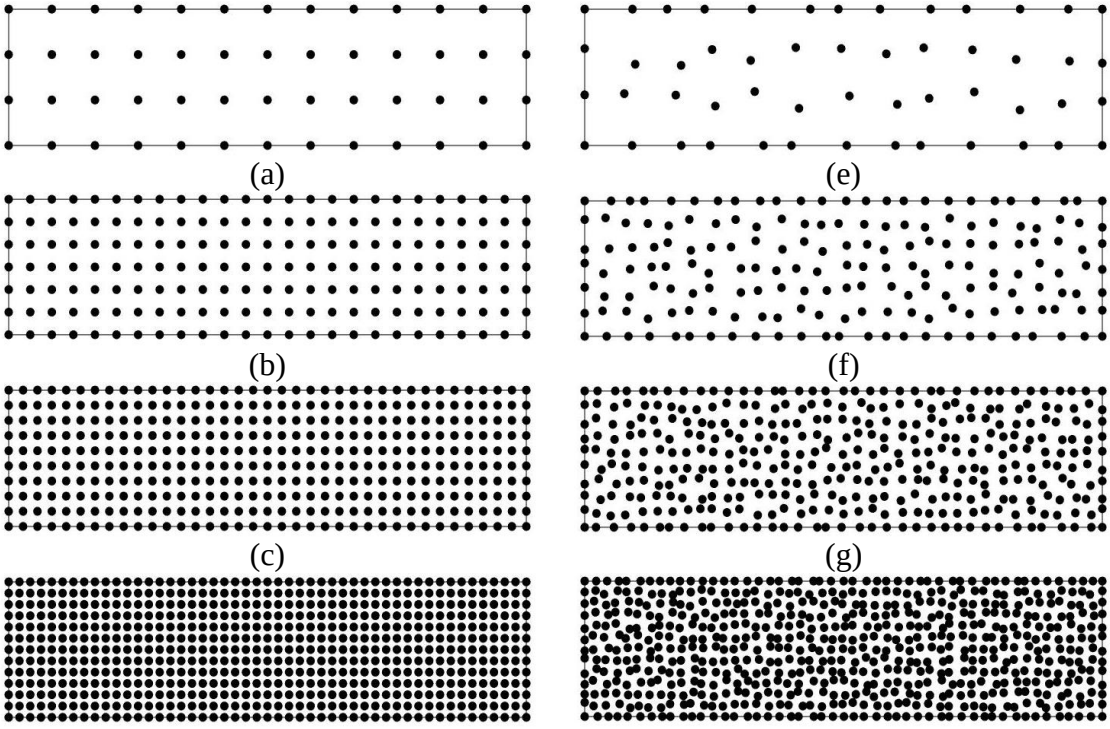
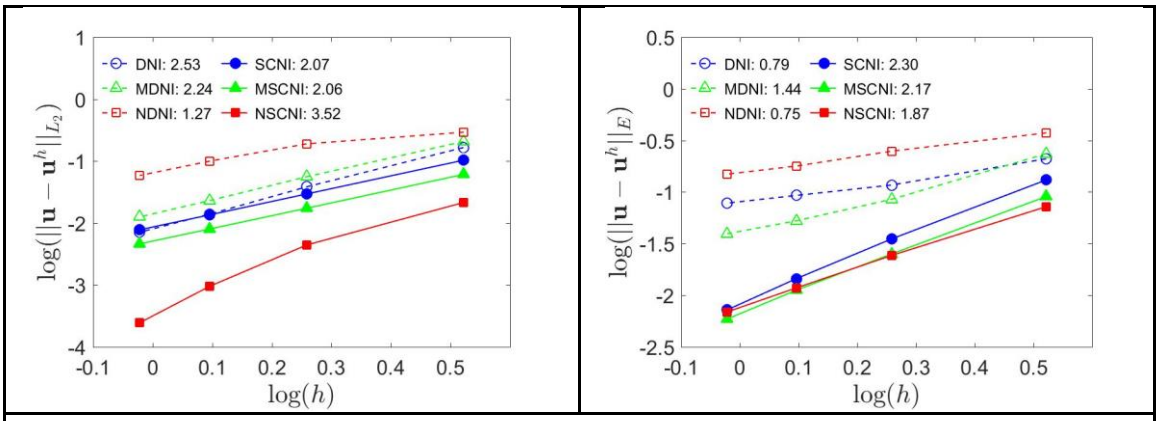
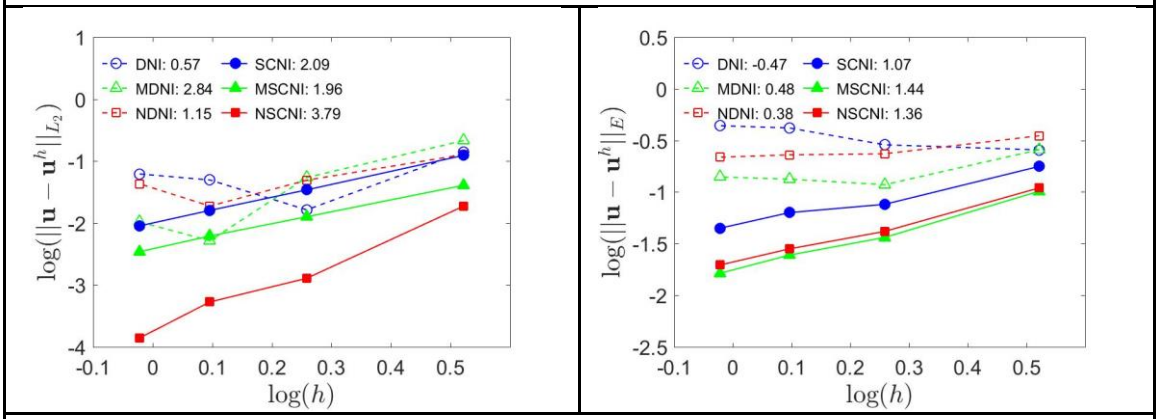


Figure 14. Uniform and non-uniform discretization for the accuracy and efficiency study in the cantilever beam problem. (a) 13×4, (b) 25×7, (c) 37×10, and (d) 49×13 nodes, where the non-uniform discretizations from (e) to (h) are consisted of randomized nodal distribuitons that corespond to the uniform nodal distribuitons from (a) to (d), respectively

Uniform and non-uniform nodal discretizations of the beam are shown in Figure 14. For GI, equally spaced 12×3, 24×6, 36×9, and 48×12 background Gauss integration cells are adopted, which correspond to four different levels of refinement. The convergence of DNI and SCNI are compared in Figure 15, which shows that MSCNI and NSCNI have the optimal convergence rate compared to SCNI and other DNI-based methods. In the cases of non-uniform discretizations, the DNI-based schemes yield poor convergence.



(a) Uniform discretization



(b) Non-uniform discretization

Figure 15. Accuracy of DNI- and SCNI-based nodal integration methods for the cantilever beam problem.

The stabilized nodal integrations MSCNI and NSCNI are compared with various order of Gauss integration as shown in Figure 16. It can be seen that only high order Gauss integration, GI 4×4 and GI 6×6, converges under uniform and non-uniform nodal distributions in both the L_2 norm and energy norm. For the stabilized nodal integration methods, MSCNI is found to be less accurate in displacement error norm but is as accurate as GI 6×6 in the energy norm. On the other hand, NSCNI shows the same level of accuracy in both displacement and energy norms as GI 6×6 under non-uniform nodal distributions. Figure 17 shows the stress distribution of DNI, NSCNI and MSCNI under non-uniform discretizations. As can be seen, DNI leads to severe oscillations, while NSCNI and MSCNI demonstrate accurate solutions compared to the exact solution.

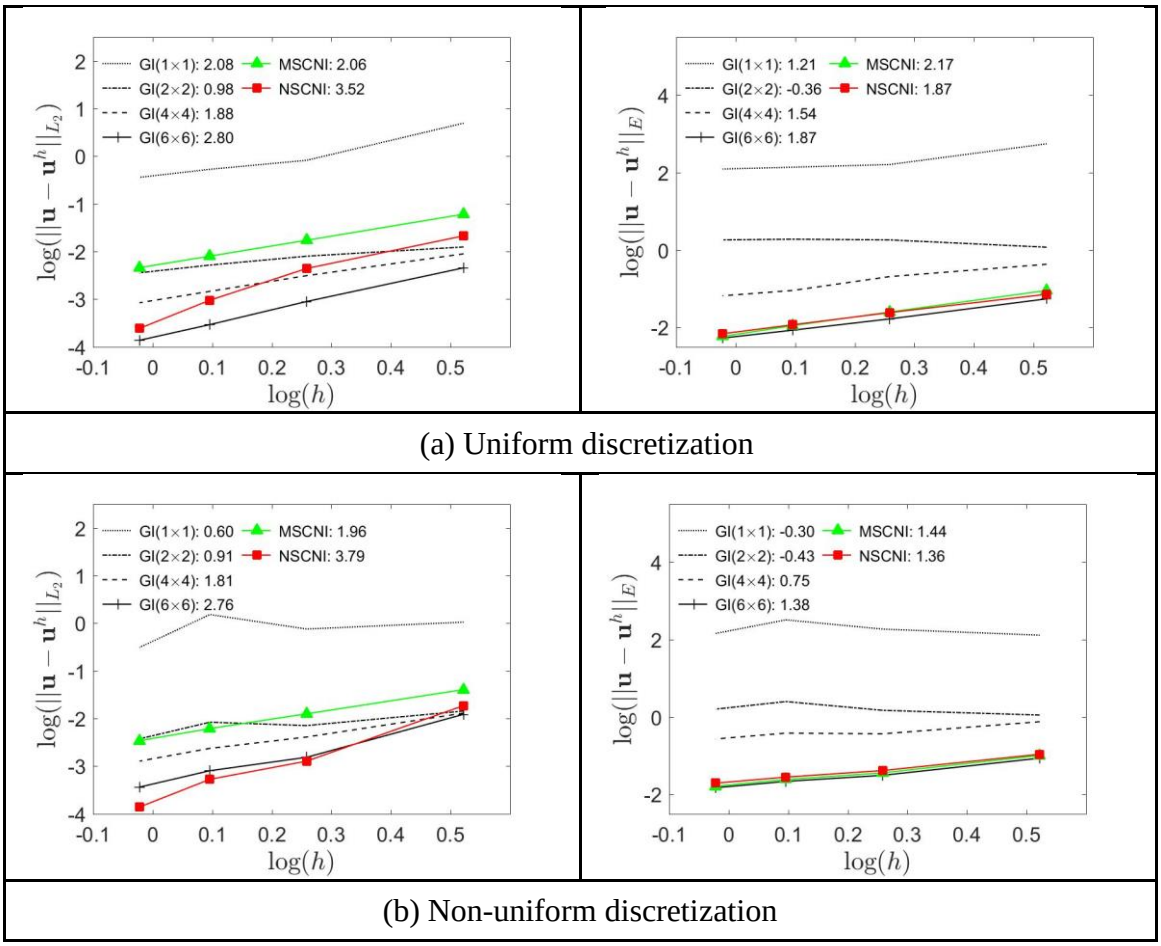


Figure 16. Accuracy of MSCNI, NSCNI, and Gauss integration (GI) for the cantilever beam problem.

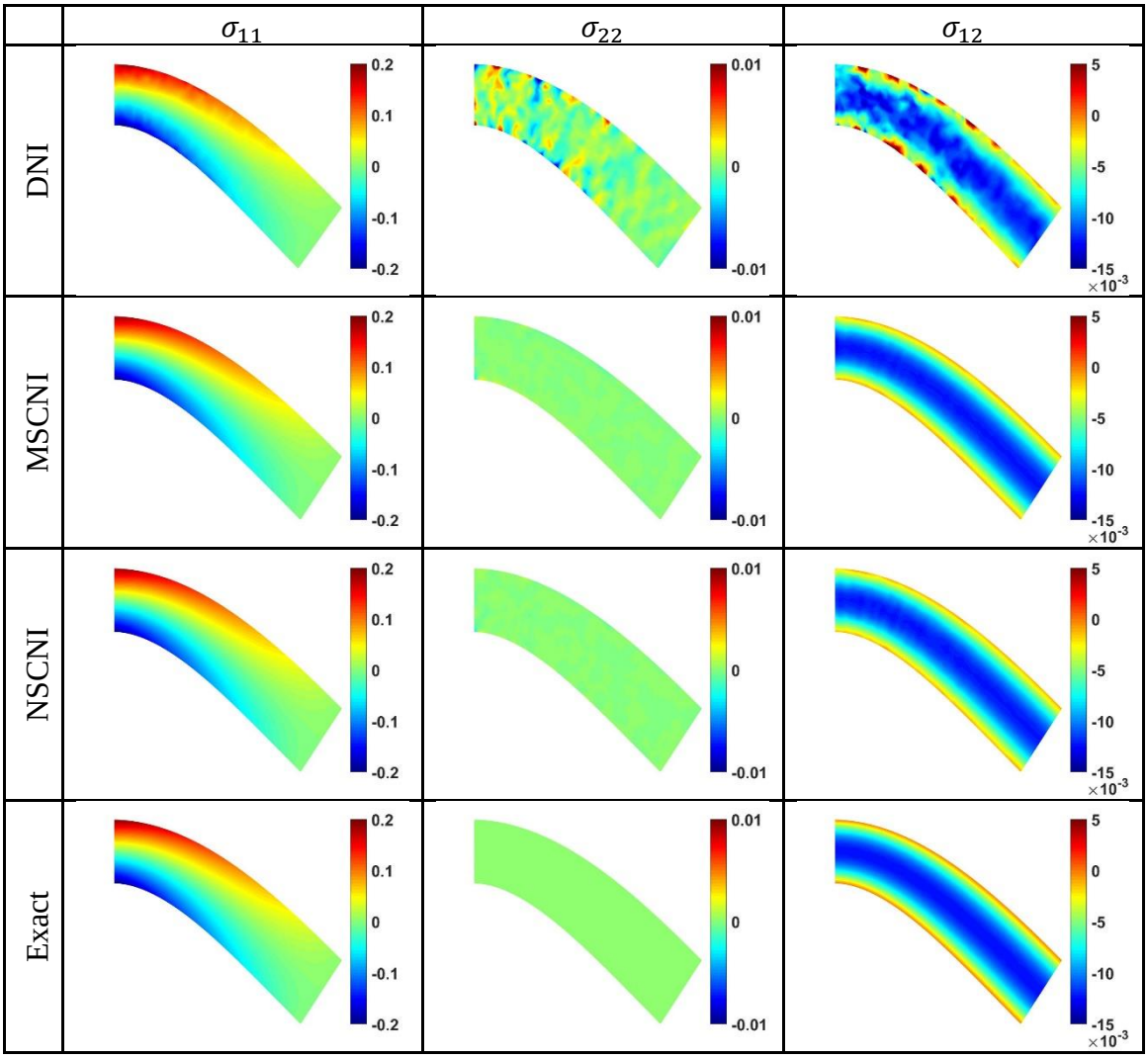
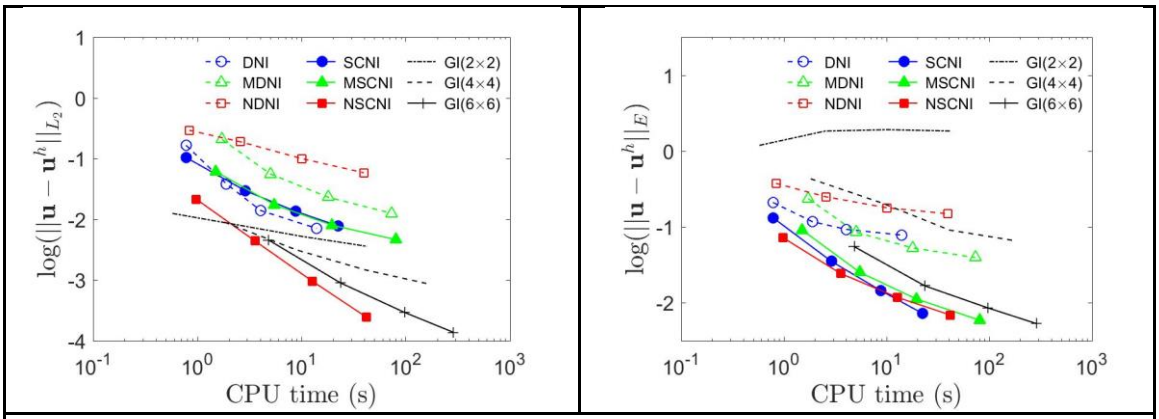
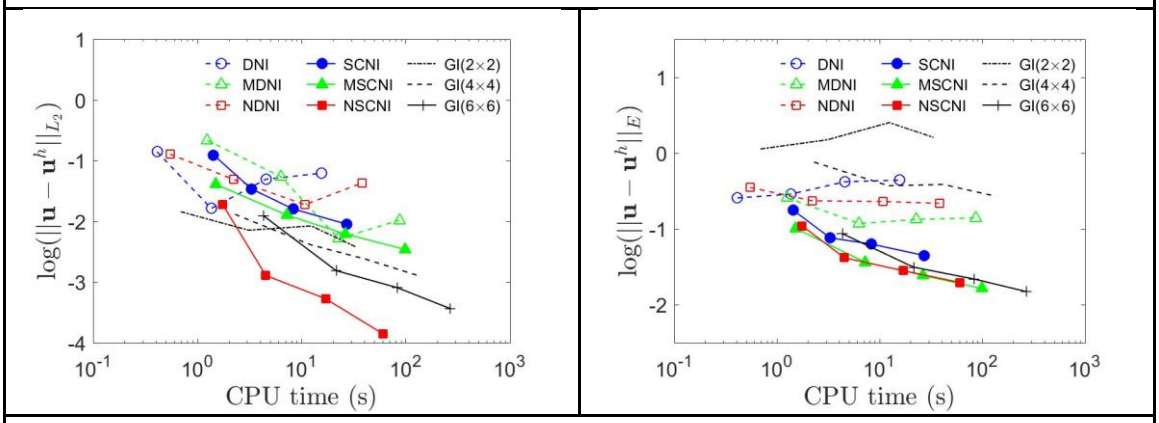


Figure 17. Stress fields of the cantilever beam problem under non-uniform discretization in the case of Figure 14 (h).

In Figure 18, the efficiency of nodal integration methods is compared to that of various order of Gauss integration. For both uniform and non-uniform cases, the computational cost of the nodal integration method is similar to that of GI 2×2. Among the nodal integration methods, NSCNI is found to achieve the same accuracy as the high-order GI with much lower computational cost, and in addition, NSCNI is more effective than SCNI and MSCNI with both accuracy and CPU time considered.



(a) Uniform discretization



(b) Non-uniform discretization

Figure 18. Computational efficiency of various order Gauss integrations (GI), DNI-based methods, and SCNI-based methods for the cantilever beam problem.

4.3 Plate with a hole problem under highly distorted discretizations

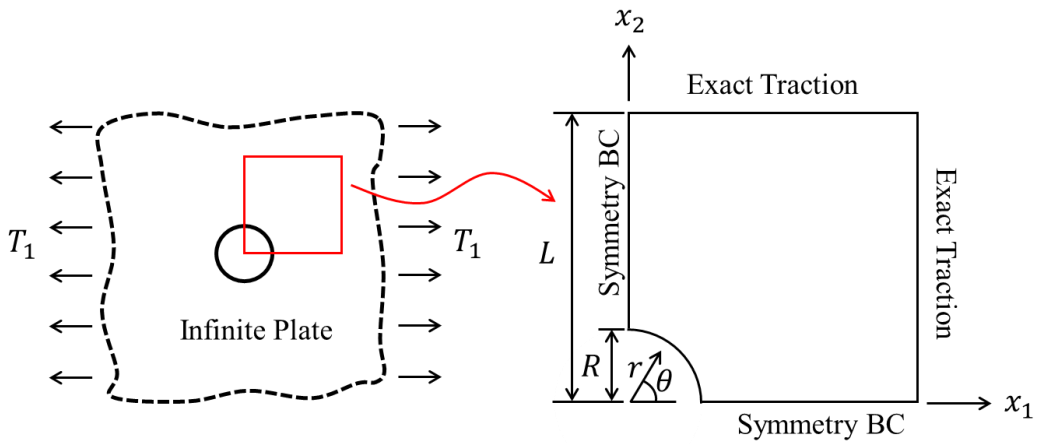


Figure 19. Problem settings for plate with a hole under far-field traction.

The problem of a plate with a circular hole is considered to assess the performance of RKPM2D under highly distorted discretizations. The plate is subjected to far-field traction T_1 in the x_1 direction, and due to symmetry, only a quarter of the plate is modeled as illustrated in Figure 19, where the length $L = 4$ and inner radius $R = 1$ are chosen. There is no body force, i.e., $\boldsymbol{b} = \boldsymbol{0}$, and the traction imposed on the finite domain boundary is defined as $\boldsymbol{t} = \boldsymbol{n} \cdot \boldsymbol{\sigma}^{exact}$, where $\boldsymbol{n}$ is the surface normal on the boundary, $\boldsymbol{\sigma}^{exact}$ is the exact stress field given below as well as the exact displacement field $\boldsymbol{u}^{exact}$:

$$u_1^{exact} = \frac{T_1 r}{8\mu}\left(\frac{r(\kappa+1)\cos\theta}{R} + \frac{2R\big((1+\kappa)\cos\theta + \cos 3\theta\big)}{r} - \frac{2R^3\cos 3\theta}{r^3}\right)$$
$$u_2^{exact} = \frac{T_1 r}{8\mu}\left(\frac{r(\kappa-3)\sin\theta}{R} + \frac{2R\big((1-\kappa)\sin\theta + \sin 3\theta\big)}{r} - \frac{2R^3\sin 3\theta}{r^3}\right) \tag{56}$$

$$\sigma_{11}^{exact} = T_1 - \frac{T_1 R^2}{r^2}\left(\frac{3}{2}(cos\,2\theta + cos\,4\theta)\right) + \frac{3T_1 R^4}{2r^4}(cos\,4\theta),$$
$$\sigma_{22}^{exact} = -\frac{T_1 R^2}{r^2}\left(\frac{1}{2}(cos\,2\theta - cos\,4\theta)\right) - \frac{3T_1 R^4}{2r^4}(cos\,4\theta), \tag{57}$$
$$\sigma_{12}^{exact} = -\frac{T_1 R^2}{r^2}\left(\frac{1}{2}(sin\,2\theta + sin\,4\theta)\right) + \frac{3T_1 R^4}{2r^4}(sin\,4\theta)$$

where (r, θ) denote polar coordinates, $T_1 = 10$, $\mu = \frac{E}{2(1+\nu)}$, $\kappa = \frac{3-\nu}{1+\nu}$, in which Young's modulus and Poisson ratio are taken as $E = 2.1 \times 10^{11}$ and $\nu = 0.3$, respectively.

To examine the influence of non-uniform discretizations on the performance of RKPM, domain discretizations with highly distorted nodal distributions are generated using Shestakov's algorithm [56]. In the discretization process, given the coordinates $\boldsymbol{x}_{1\sim4}$ of four corner nodes of a quadrilateral domain, a new node is created with its position $\boldsymbol{x}$ determined by Eq. (58):

$$\boldsymbol{x} = \alpha_d\beta_d\boldsymbol{x}_1 + (1-\alpha_d)\beta_d\boldsymbol{x}_2 + (1-\alpha_d)(1-\beta_d)\boldsymbol{x}_3 + \alpha_d(1-\beta_d)\boldsymbol{x}_4 \tag{58}$$

where

$$\begin{aligned} \alpha_d &= \gamma_d + (1-2\gamma_d)rand_1 \\ \beta_d &= \gamma_d + (1-2\gamma_d)rand_2 \end{aligned} \tag{59}$$

In this algorithm, the following control parameters need to be defined:

- $n_c \in \mathbb{N}$: a constant parameter that controls the discretization refinement level;
- $0 < \gamma_d \leq 0.5$: a constant parameter that controls the discretization distortion level;
- $rand_1, rand_2 \in [0,1]$: random numbers that randomly perturb the nodal positions.

After generating the new nodal point $\boldsymbol{x}$ by Eq. (58), the quadrilateral domain can be divided into four quadrilateral sub-domains, and within each sub-domain, the above procedure can be repeatedly applied to create more nodal points and sub-domains until the total number of nodes equal to $(2^{n_c}+1) \times (2^{n_c}+1)$, i.e., when the desired refinement level controlled by n_c is achieved.

Note that, when the distortion parameter is chosen as $\gamma_d = 0.5$, a uniform nodal discretization is obtained, whereas decreasing γ_d will result in a more distorted nodal distribution, for which poorly shaped elements are yielded if FEM is used. On the other hand, varying the random numbers $rand_1$ and $rand_2$ can lead to different nodal distributions, but the associated distortion levels will remain the same as long as the distortion parameter γ_d is unchanged.

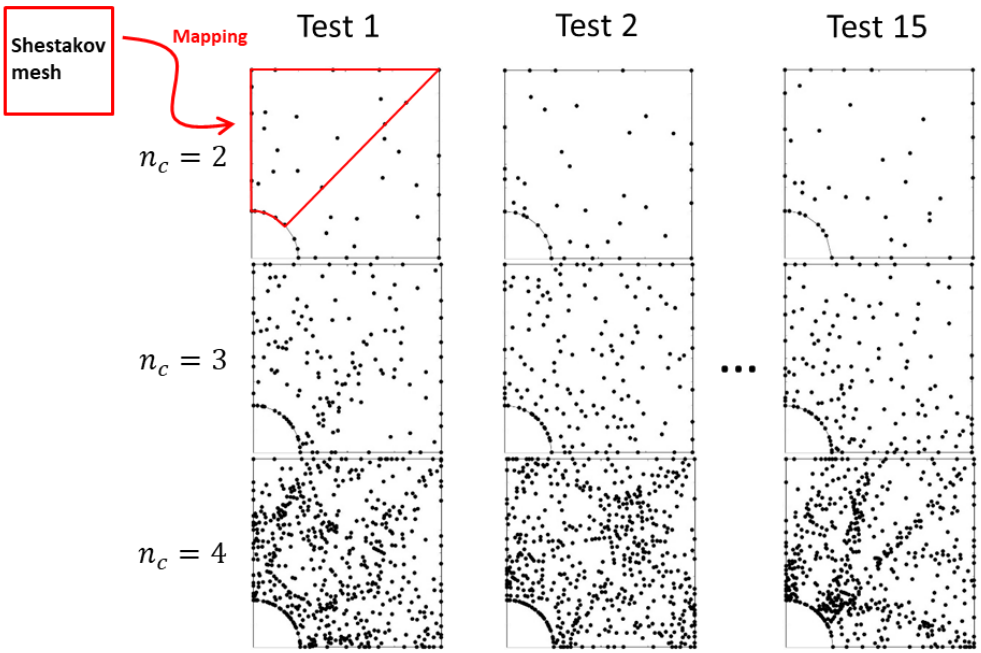


Figure 20. Highly distorted nodal discretizations at three refinement levels for the plate, where 15 randomly perturbed discretizations (from test 1 to test 15) at each refinement level are generated

In the present study, the plate domain is firstly divided into two sub-domains as illustrated in red color in Figure 20, and each sub-domain is discretized with $(2^{n_c}+1)\times(2^{n_c}+1)$ nodes using the abovementioned Shestakov's algorithm. Nodes located on the domain boundaries are generated in a similar fashion. Interested readers can refer to [56] for more details of the Shestakov's algorithm. Three levels of discretization refinement are adopted with the refinement parameter $n_c = 2, 3, 4$, respectively. Further, at each refinement level, 15 tests are performed with a different set of random numbers ($rand_1, rand_2$ in Eq. (59)) for each test, as illustrated in Figure 20. It is noteworthy to mention that, since a constant distortion parameter $\gamma_d = 0.1$ is adopted, the distortion levels of all 45 discretizations are considered identical and highly distorted.

To study the convergence properties of RKPM, we define the nodal spacing as $\tilde{h} = \sqrt{A/NP}$, where A is the area of the whole domain and NP is the total number of nodes. In order to evaluate the numerical accuracy, we compute the mean values of L_2 and energy norms for the 15 discretizations employed at each refinement level, and plot them in Figure

21 for comparison of different numerical schemes. The results show that DNI and high-order GI do not converge under the highly distorted discretizations, where $2^{n_c} \times 2^{n_c}$ uniformly distributed background integration cells are adopted for the GI scheme. On the other hand, all SCNI-based nodal integration schemes show satisfactory performance. In particular, both NSCNI and MSCNI achieve optimal convergence and good accuracy.

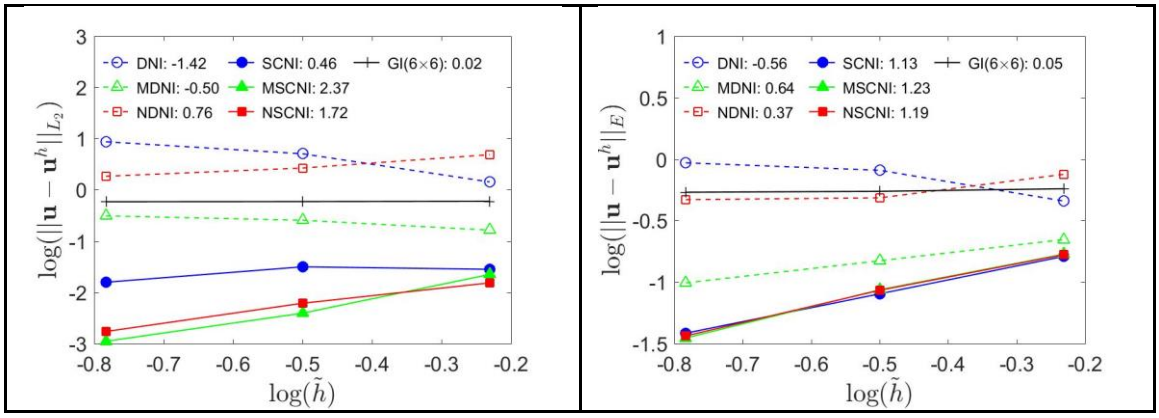


Figure 21. Convergence under highly distorted discretizations for the plate problem, where the mean error norms of 15 randomly perturbed discretizations at each refinement level are plotted

To assess the sensitivity of each numerical scheme on randomly perturbed nodal positions, the following standard deviations in numerical error norms are defined:

$$\text{Standard deviation in } L_2\text{: } S_{L_2} = \sqrt{\frac{\sum_{i=1}^{NT}\left(e_i^{L_2} - \bar{e}^{L_2}\right)^2}{NT-1}}, \tag{60}$$

$$\text{Standard deviation in energy: } S_E = \sqrt{\frac{\sum_{i=1}^{NT}(e_i^{E} - \bar{e}^{E})^2}{NT-1}}, \tag{61}$$

where $NT = 15$ denotes the total number of test cases at each refinement level, $e_i^{L_2} = \|\boldsymbol{u} - \boldsymbol{u}^h\|_{L_2}$ and $e_i^E = \|\boldsymbol{u} - \boldsymbol{u}^h\|_E$ denote the L_2 and energy error norms for the i-th test, $\bar{e}^{L_2}$ and $\bar{e}^{eng}$ are the mean values of the L_2 and energy error norms, respectively. Table 8 and Table 9 give the calculated standard deviations in L_2 and energy error norms, which show that SCNI-based schemes yield similar deviations with the high-order GI, whereas DNI-based methods result in very large deviations. The results clearly indicate

that DNI-based methods are very sensitive to the nodal positions, whereas SCNI-based methods are more robust and perform well even when the highly distorted nodal discretizations are perturbed.

Table 8, Standard deviation S_{L_2} of L_2 error norms under highly distorted discretizations

Quadrature Scheme	Refinement level $n_c = 2$	$n_c = 3$	$n_c = 4$
GI (6×6)	2.8E-03	2.1E-03	7.0E-04
DNI	1.7E+00	2.0E+00	4.1E+00
MDNI	7.0E-02	4.8E-02	2.7E-02
NDNI	2.3E+00	2.4E+00	6.5E-01
SCNI	9.5E-03	8.5E-02	7.5E-03
MSCNI	8.6E-03	8.2E-02	1.4E-03
NSCNI	6.5E-03	7.8E-03	5.2E-04

Table 9, Standard deviation S_E of energy error norms under highly distorted discretiztions

Quadrature Scheme	Refinement level $n_c = 2$	$n_c = 3$	$n_c = 4$
GI (6×6)	3.8E-03	2.2E-03	7.6E-03
DNI	1.3E-01	2.6E-01	2.6E+00
MDNI	2.9E-02	6.3E-02	1.1E-02
NDNI	1.9E-01	1.7E-01	8.4E-02
SCNI	1.1E-02	7.7E-02	9.1E-03
MSCNI	1.1E-02	6.9E-02	5.5E-03
NSCNI	1.2E-02	6.6E-02	5.3E-03

Finally, the computational efficiency of high-order GI and different nodal integrations are compared in Figure 22, where the mean values of error norms at each refinement level are plotted. Compared to DNI-based methods and the high-order GI, both MSCNI and NSCNI show superior performance, and NSCNI achieves the best efficiency among all schemes under the highly distorted nodal discretizations.

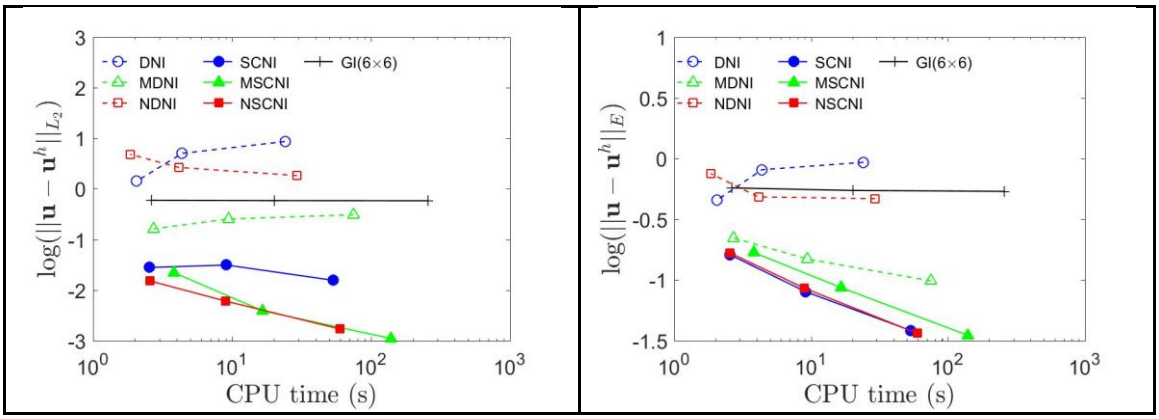


Figure 22. Efficiency under highly distorted discretizations for the plate problem, where the mean error norms of 15 randomly perturbed discretizations at each refinement level are plotted

5 Conclusion

We have developed an open-source software called RKPM2D for meshfree analysis. The program is developed based on the Reproducing Kernel Particle Method (RKPM), and consists of a set of data structures and routines for discretizing two-dimensional domains of arbitrary shape, nodal representative domain creation by Voronoi diagram partitioning, reproducing kernel shape function generation, domain integration using Gauss integration and various stabilized nodal integration methods, meshfree Galerkin matrix assembly and solver, and visualization tools for post-processing. Benchmark problems are solved to analyze and verify the convergence, efficiency, and robustness properties of stabilized and nodally integrated RKPM implemented into RKPM2D, under both uniform and highly non-uniform nodal distributions.

The open source code can serve as an entry point for researchers who are interested in the computer implementation of RKPM, and other related meshfree methods, and the code can also be adopted as a rapid prototyping and testing tool for further development of advanced meshfree algorithms. Although the linear elasticity problem is chosen as a model problem, the flexibility of the code allows the extension to solve different PDEs for various scientific and engineering problems.

Acknowledgments

Appendix

In this section, a diffusion problem is used as example to illustrate how to modify RKPM2D for the solution of different types of PDEs. A diffusion equation is considered as follows:

$$\begin{aligned} \left(D_{ij}u_{,j}\right)_{,i} + b = 0 \quad & \text{on } \Omega \\ D_{ij}u_{,j}n_i = t \quad & \text{on } \partial\Omega_t \\ u = g \quad & \text{on } \partial\Omega_g \end{aligned} \tag{62}$$

where u is a scalar field, D_{ij} is the diffusivity, b is the source term, t and g are the prescribed boundary flux and boundary values of u on $\partial\Omega_t$ and $\partial\Omega_g$, respectively. By introducing the RK approximation in Eq. (11), (62) can be recast into the following matrix equations for isotropic scalar diffusivity:

$$\sum_J K_{IJ}u_J - F_I = 0 \tag{63}$$

where

$$K_{IJ} = K_{IJ}^d + K_{IJ}^\beta - \left(K_{IJ}^g + K_{IJ}^{g^T}\right) \tag{64}$$

$$F_I = F_I^b + F_I^t + F_{IJ}^\beta - F_I^g \tag{65}$$

in which the matrices and vectors in nodal integration are expressed as

$$K_{IJ}^d = \int_\Omega \boldsymbol{B}_I^T(\boldsymbol{x})\boldsymbol{D}\boldsymbol{B}_J(\boldsymbol{x})d\Omega \approx \sum_{N=1}^{NP} \boldsymbol{B}_I^T(\boldsymbol{x}_N)\boldsymbol{D}\boldsymbol{B}_J(\boldsymbol{x}_N)A_N \tag{66}$$

$$F_I^b = \int_\Omega \Psi_I^T(\boldsymbol{x})b(\boldsymbol{x})d\Omega \approx \sum_{N=1}^{NP} \Psi_I^T(\boldsymbol{x}_N)b(\boldsymbol{x}_N)A_N \tag{67}$$

$$F_I^t = \int_{\partial\Omega_t} \Psi_I^T(\boldsymbol{x}) t(\boldsymbol{x}) d\Gamma \approx \sum_{N=1}^{NPt} \Psi_I^T(\boldsymbol{x}_N) t(\boldsymbol{x}_N) L_N \tag{68}$$

$$K_{IJ}^{\beta} = \beta \int_{\partial\Omega_g} \Psi_I^T(\boldsymbol{x}) S \Psi_J(\boldsymbol{x}) d\Gamma \approx \beta \sum_{N=1}^{NPg} \Psi_I^T(\boldsymbol{x}_N) S \Psi_J(\boldsymbol{x}_N) L_N \tag{69}$$

$$K_{IJ}^{g} = \int_{\partial\Omega_g} \boldsymbol{B}_I^T(\boldsymbol{x}) \boldsymbol{D}\boldsymbol{\eta} S \Psi_J(\boldsymbol{x}) d\Gamma \approx \sum_{N=1}^{NPg} \boldsymbol{B}_I^T(\boldsymbol{x}_N) \boldsymbol{D}\boldsymbol{\eta} S \Psi_J(\boldsymbol{x}_N) L_N \tag{70}$$

$$F_{IJ}^{\beta} = \beta \int_{\partial\Omega_g} \Psi_I^T(\boldsymbol{x}) S g d\Gamma \approx \beta \sum_{N=1}^{NPg} \Psi_I^T(\boldsymbol{x}_N) S g L_N \tag{71}$$

$$F_I^{g} = \int_{\partial\Omega_g} \boldsymbol{B}_I^T(\boldsymbol{x}) \boldsymbol{D}\boldsymbol{\eta} S g d\Gamma \approx \sum_{N=1}^{NPg} \boldsymbol{B}_I^T(\boldsymbol{x}_N) \boldsymbol{D}\boldsymbol{\eta} S g L_N \tag{72}$$

$$\boldsymbol{B}_I(\boldsymbol{x}_N) = \begin{bmatrix} \Psi_{I,1}(\boldsymbol{x}_N) \\ \Psi_{I,2}(\boldsymbol{x}_N) \end{bmatrix}, \ \boldsymbol{D} = \begin{bmatrix} d & 0 \\ 0 & d \end{bmatrix}, \ \boldsymbol{\eta} = \begin{bmatrix} n_1 \\ n_2 \end{bmatrix}, \ S = 1. \tag{73}$$

where $\boldsymbol{D}$ is the diffusivity tensor, d is the diffusion coefficient and $\boldsymbol{\eta}$ is a collection of components of the surface unit normal on the boundary, and $S = 1$ is set for the convenience of keeping a unified coding structure in RKPM2D. Let us consider a diffusion problem (Eq. (62)) with a manufactured solution:

$$u^{exact} = 0.1 + 0.1x_1 + 0.2x_2 \tag{74}$$

in a circular domain $\Omega \subset \mathbb{R}^2$ shown in Figure 9 from Section 3.2. The flux $t = 0.1n_1 + 0.2n_2$ is imposed on $\partial\Omega_t$: $(x_1, x_2) \in \partial\Omega, x_2 > 0.5$ where n_1, and n_2 are the normal vector components. $g = u^{exact}$ is enforced on $\partial\Omega_g$: $(x_1, x_2) \in \partial\Omega, x_2 \leq 0.5$, and the body source is $b = 0$.

The input file for this problem is generated in the function `getInput`. Compared to Listing 1, the following changes need to be made:

- Remove `Model.nu` and `Model.Condition`, as Poisson ratio and plane-stress/strain condition are not required for diffusion problem.

- Replace `Model.E` with `Model.d` (i.e., change the definition of Young's modulus E to be the diffusion coefficient d).
- Replace `Model.ElasticTensor` with `Model.DiffusiveTensor` (i.e., change the definition of elastic tensor $\boldsymbol{C}$ to be the diffusive tensor $\boldsymbol{D}$).
- Set `Model.DiffusiveTensor = diag([Model.d,Model.d])` to define the diffusive tensor $\boldsymbol{D} = \begin{bmatrix} d & 0 \\ 0 & d \end{bmatrix}$.
- Set `Model.DOFu = 1` to change the nodal degrees of freedom `DOFu` from 2 to 1.
- Set `u_exact = 0.1+0.1*x1+0.2*x2` to define the exact solution u^{exact}.

In addition, we also need to modify the subroutine `getBoundaryConditions` to generate the exact boundary flux $t = \boldsymbol{\eta}^T \boldsymbol{D} \boldsymbol{\nabla} u^{exact}$, source term $b = \boldsymbol{\nabla} \cdot (\boldsymbol{D} \boldsymbol{\nabla} u^{exact})$, essential boundary conditions $g = u^{exact}$, and switch matrix $S = 1$ based on a given expression of the exact solution u^{exact} in a symbolic form, as shown in Listing 13.

```
function [function_S,function_g,function_traction,function_b] =
getBoundaryConditions(Model)
syms x1 x2 n1 n2
% function handle for essential boundary condition g
u = Model.ExactSolution.u_exact;
D = Model.DiffusiveTensor;
function_g = matlabFunction(u);
% function handle for diff(u)
dudx1 = diff(u,x1);
dudx2 = diff(u,x2);
flux = D*[dudx1; dudx2;];
% function handle for surface flux (traction)
eta = [n1; n2;];
surf_flux = eta'*flux;
function_traction = matlabFunction(surf_flux,'Vars',[x1 x2 n1 n2]);
% function handle for source b
b = [diff(flux(1),x1)+ diff(flux(2),x2)];
function_b = matlabFunction(b,'Vars',[x1 x2]);
% function handle for switch S
function_S = matlabFunction(sym(1),'Vars',[x1 x2]);
end
```

Listing 13. Command lines of function to generate exact heat flux t, heat sources b, imposed temperature g, and switch matrix S for diffusion problem.

Due to the change of dimensionality in $\boldsymbol{B}$ and $\boldsymbol{\Psi}$ matrix, modifications are made to `MatrixAssmebly` (Listing 10) as follows

- Set `d = Model.d` to define the diffusivity coefficient.

- Set `D = Model.DiffusiveTensor` to define the diffusivity from input files.
- Replace `E` with `d` (i.e., replace the Young's modulus `E` with diffusion coefficient `d`).
- Replace `C` with `D` (i.e., replace elastic tensor `C` with diffusive tensor `D`).
- Set `B = sparse(2,nP*DOFu).`
- Set `PSI = sparse(1,nP*DOFu).`
- Modify the allocation of the `B` and `PSI` from shape function `SHP` and derivative `SHPDX1`, `SHPDX2` as:
  - `PSI = SHP(idx_nQuad,:);`
  - `B(1,:)= SHPDX1(idx_nQuad,:);`
  - `B(2,:) = SHPDX2(idx_nQuad,:);`
- Set `ETA = [n1; n2;]` to define the surface normal $\boldsymbol{\eta}$.

With the abovementioned modifications, RKPM2D is converted to a code for solving a diffusion problem. By comparing the original code for the elasticity problem with the modified code for diffusion problem, one can see that a minimal code modification is required to. This capability of easy code extension is a unique feature of RKPM2D.

References

[1] J.-S. Chen, C. Pan, C.-T. Wu and W. K. Liu, "Reproducing kernel particle methods for large deformation analysis of non-linear structures," *Computer Methods in Applied Mechanics and Engineering,* vol. 139, no. 1-4, pp. 195-227, 1996.

[2] J.-S. Chen, M. Hillman and S.-W. Chi, "Meshfree methods: progress made after 20 years," *Journal of Engineering Mechanics,* vol. 143, no. 4, p. 04017001, 2017.

[3] W. K. Liu, S. Jun and Y. F. Zhang, "Reproducing kernel particle methods," *International Journal for Numerical Methods in Fluids,* vol. 20, no. 8-9, pp. 1081-1106, 1995.

[4] W. K. Liu, S. Hao, T. Belytschko, S. Li and C. T. Chang, "Multiple scale meshfree methods for damage fracture and localization," *Computational materials science,* vol. 16, no. 1-4, pp. 197-205, 1999.

[5] S. Li, W. Hao and W. K. Liu, "Mesh-free simulations of shear banding in large deformation," *International Journal of solids and structures,* vol. 37, no. 48-50, pp. 7185-7206, 2000.

[6] W. K. Liu, S. Jun, D. T. Sihling, Y. Chen and W. Hao, "Multiresolution reproducing kernel particle method for computational fluid dynamics," *International Journal for Numerical Methods in Fluids,* vol. 24, no. 12, pp. 1391--1415, 1997.

[7] W. K. Liu and Y. Chen, "Wavelet and multiple scale reproducing kernel methods," *International Journal for Numerical Methods in Fluids,* vol. 21, no. 10, pp. 901-931, 1995.

[8] W. K. Liu, Y. Chen, R. A. Uras and C. T. Chang, "Generalized multiple scale reproducing kernel particle methods," *Computer Methods in Applied Mechanics and Engineering,* vol. 139, no. 1-4, pp. 91-157, 1996.

[9] M. O. Ruter and J.-S. Chen, "An enhanced-strain error estimator for Galerkin meshfree methods based on stabilized conforming nodal integration," *Computers & Mathematics with Applications,* vol. 74, no. 9, pp. 2144-2171, 2017.

[10] Y. You, J.-S. Chen and H. Lu, "Filters, reproducing kernel, and adaptive meshfree method," *Computational Mechanics,* vol. 31, no. 3-4, pp. 316-326, 2003.

[11] J.-S. Chen, M. Hillman and M. Ruter, "An arbitrary order variationally consistent integration for Galerkin meshfree methods," *International Journal for Numerical Methods in Engineering,* vol. 95, no. 5, pp. 387-418, 2013.

[12] M. Hillman and J.-S. Chen, "An accelerated, convergent, and stable nodal integration in Galerkin meshfree methods for linear and nonlinear mechanics," *International Journal for Numerical Methods in Engineering,* vol. 107, no. 7, pp. 603-630, 2016.

[13] J.-S. Chen and D. Wang, "A constrained reproducing kernel particle formulation for shear deformable shell in Cartesian coordinates," *International Journal for Numerical Methods in Engineering,* vol. 68, no. 2, pp. 151-172, 2006.

[14] N. H. Kim, K. K. Choi, J.-S. Chen and M. E. Botkin, "Meshfree analysis and design sensitivity analysis for shell structures," *International Journal for Numerical Methods in Engineering,* vol. 53, no. 9, pp. 2087-2116, 2002.

[15] J.-S. Chen, C. Pan, C. Roque and H.-P. Wang, "A Lagrangian reproducing kernel particle method for metal forming analysis," *Computational Mechanics,* vol. 22, no. 3, pp. 289-307, 1998.

[16] J.-S. Chen, H.-P. Wang, S. Yoon and Y. You, "Some recent improvements in meshfree methods for incompressible finite elasticity boundary value problems with contact," *Computational Mechanics,* vol. 25, no. 2-3, pp. 137-156, 2000.

[17] H.-P. Wang, C.-T. Wu and J.-S. Chen, "A reproducing kernel smooth contact formulation for metal forming simulations," *Computational Mechanics,* vol. 54, no. 1, pp. 151-169, 2014.

[18] J.-S. Chen, R. R. Basava, Y. Zhang, R. Csapo, V. Malis, U. Sinha, J. Hodgson and S. Sinha, "Pixel-based meshfree modelling of skeletal muscles," *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization,* vol. 4, no. 2, pp. 73-85, 2016.

[19] H. Wei, J.-S. Chen and M. Hillman, "A stabilized nodally integrated meshfree formulation for fully coupled hydro-mechanical analysis of fluid-saturated porous media," *Computers & Fluids,* vol. 141, pp. 105-115, 2016.

[20] H. Wei, J.-S. Chen and F. Beckwith, "A naturally stabilized semi-Lagrangian meshfree formulation for multiphase porous media with application to landslide modeling," *Journal of Engineering Mechanics,* vol. under review, 2018.

[21] J.-S. Chen, C.-T. Wu and T. Belytschko, "Regularization of material instabilities by meshfree approximations with intrinsic length scales," *International Journal for Numerical Methods in Engineering,* vol. 47, no. 7, pp. 1303-1322, 2000.

[22] J.-S. Chen, X. Zhang and T. Belytschko, "An implicit gradient model by a reproducing kernel strain regularization in strain localization problems," *Computer Methods in Applied Mechanics and Engineering,* vol. 193, no. 27-29, pp. 2827-2844, 2004.

[23] H. Wei and J.-S. Chen, "A damage particle method for smeared modeling of brittle fracture," *International Journal for Multiscale Computational Engineering,* vol. 16, no. 4, 2018.

[24] M. J. Roth, J.-S. Chen, T. R. Slawson and K. T. Danielson, "Stable and flux-conserved meshfree formulation to model shocks," *Computational Mechanics,* vol. 57, no. 5, pp. 773-792, 2016.

[25] M. J. Roth, J.-S. Chen, K. T. Danielson and T. R. Slawson, "Hydrodynamic meshfree method for high-rate solid dynamics using a Rankine--Hugoniot enhancement in a Riemann-SCNI framework," *International Journal for Numerical Methods in Engineering,* vol. 108, no. 12, pp. 1525-1549, 2016.

[26] P.-C. Guan, S.-W. Chi, J.-S. Chen, T. Slawson and M. J. Roth, "Semi-Lagrangian reproducing kernel particle method for fragment-impact problems," *International Journal of Impact Engineering,* vol. 38, no. 12, pp. 1033-1047, 2011.

[27] J. A. Sherburn, M. J. Roth, J. Chen and M. Hillman, "Meshfree modeling of concrete slab perforation using a reproducing kernel particle impact and penetration

formulation," *International Journal of Impact Engineering,* vol. 86, pp. 96-110, 2015.

[28] S.-W. Chi, C.-H. Lee, J.-S. Chen and P.-C. Guan, "A level set enhanced natural kernel contact algorithm for impact and penetration modeling," *International Journal for Numerical Methods in Engineering,* vol. 102, no. 34, pp. 839-866, 2015.

[29] J.-S. Chen, W. K. Liu, M. Hillman, S.-W. Chi, Y. Lian and M. Bessa, "Reproducing Kernel Particle Method for Solving Partial Differential Equations," *Encyclopedia of Computational Mechanics, Second Edition,* pp. 1-44, 2017.

[30] E. Hardee, K.-H. Chang, S. Yoon, M. Kaneko, I. Grindeanu and J.-S. Chen, "A Structural Nonlinear Analysis Workspace (SNAW) based on meshless methods," *Advances in Engineering Software,* vol. 30, no. 30, pp. 153-175, 1999.

[31] Y.-M. Hsieh and M.-S. Pan, "ESFM: An essential software framework for meshfree methods," *Advances in Engineering Software,* vol. 76, pp. 133-147, 2014.

[32] E. Barbieri and M. Meo, "A fast object-oriented Matlab implementation of the Reproducing Kernel Particle Method," *Computational Mechanics,* vol. 49, no. 5, pp. 581-602, 2012.

[33] C. Cartwright, S. Oliveira and D. E. Stewart, "Parallel support set searches for meshfree methods," *SIAM Journal on Scientific Computing,* vol. 28, no. 4, pp. 1318-1334, 2006.

[34] G. F. Parreira, A. R. Fonseca, A. C. Lisboa, E. J. Silva and R. C. Mesquita, "Efficient algorithms and data structures for element-free Galerkin method," *IEEE transactions on magnetics,* vol. 42, no. 4, pp. 659-662, 2006.

[35] J. Olliff, B. Alford and D. C. Simkins, "Efficient searching in meshfree methods," *Computational Mechanics,* pp. 1-23, 2018.

[36] J. Nitsche, "Uber ein Variationsprinzip zur Losung von Dirichlet-Problemen bei Verwendung von Teilraumen, die keinen Randbedingungen unterworfen sind.," *Abh. Math. Sem. Univ. Hamburg,* vol. 36, pp. 9-15, 1970-1971.

[37] S. Fernandez-Mendez and A. Huerta, "Imposing essential boundary conditions in mesh-free methods," *Computer Methods in Applied Mechanics and Engineering,* vol. 193, no. 12-14, pp. 1257-1275, 2004.

[38] J.-S. Chen, C.-T. Wu, S. Yoon and Y. You, "A stabilized conforming nodal integration for Galerkin mesh-free methods," *International Journal for Numerical Methods in Engineering,* vol. 50, no. 2, pp. 435-466, 2001.

[39] J.-S. Chen, W. Hu, M. Puso, Y. Wu and X. Zhang, "Strain smoothing for stabilization and regularization of Galerkin meshfree methods," in *Meshfree Methods for Partial Differential Equations III*, Springer, 2007, pp. 57-75.

[40] MathWorks, *MATLAB,* Natick, Massachusetts: The MathWorks Inc., 1984.

[41] S. Li and W. K. Liu, Meshfree particle methods, Berlin: Springer Science & Business Media, 2007.

[42] J.-S. Chen and H.-P. Wang, "New boundary condition treatments in meshfree computation of contact problems," *Computer Methods in Applied Mechanics and Engineering,* vol. 187, no. 3-4, pp. 441-468, 2000.

[43] J. Dolbow and T. Belytschko, "Numerical integration of the Galerkin weak form in meshfree methods," *Computational Mechanics,* vol. 23, no. 3, pp. 219-230, 1999.

[44] Y. Lu, T. Belytschko and L. Gu, "A new implementation of the element free Galerkin method," *Computer Methods in Applied Mechanics and Engineering,* vol. 113, no. 3-4, pp. 397-414, 1994.

[45] M. A. Puso, E. Zywicz and J. Chen, "A new stabilized nodal integration approach," in *Meshfree Methods for Partial Differential Equations III*, Berlin, Springer, 2007, pp. 207-217.

[46] J. Sievers, "Constrain the vertices of a Voronoi decomposition to the domain of the input data," MathWorks, 31th October 2016. [Online]. Available: http://se.mathworks.com/matlabcentral/fileexchange/34428-voronoilimit. [Accessed 1st September 2018].

[47] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM,* vol. 18, no. 9, pp. 509-517, 1975.

[48] E. Perkins and J. R. Williams, "A fast contact detection algorithm insensitive to object sizes," *Engineering Computations,* vol. 18, no. 1/2, pp. 48-62, 2001.

[49] J. H. Friedman, J. L. Bentley and R. A. Finkel, "An algorithm for finding best matches in logarithmic time," *ACM Trans. Math. Software,* vol. 3, no. SLAC-PUB-1549-REV. 2, pp. 209-226, 1976.

[50] C. Steger, "On the calculation of arbitrary moments of polygons," Munchen University, Tech. Rep. FGBV-96-05, Munchen, Germany, 1996.

[51] J. R. Gilbert, C. Moler and R. Schreiber, "Sparse matrices in MATLAB: design and implementation," *SIAM Journal on Matrix Analysis and Applications,* vol. 13, no. 1, pp. 333-356, 1992.

[52] A. George and J. W. Liu, "Computer solution of large sparse positive definite system," in *Prentice Hall Professional Technical Reference*, Englewood Cliffs. NJ, 1981.

[53] A. Jennings, "A compact storage scheme for the solution of symmetric linear simultaneous equations," *The Computer Journal,* vol. 9, no. 3, pp. 281-285, 1966.

[54] A. H. Sherman, "On the efficient solution of sparse systems of linear and nonlinear equations," *Yale University,* 1975.

[55] Y. Saad, Iterative methods for sparse linear systems, Minneapolis, MN: SIAM, 2003.

[56] A. Shestakov, D. Kershaw and G. Zimmerman, "Test problems in radiative transfer calculations," *Nuclear science and engineering,* vol. 105, no. 1, pp. 88-104, 1990.