Medusa: A C++ Library for Solving PDEs Using Strong Form Mesh-free Methods

JURE SLAK*, Jožef Stefan Institute, Slovenia and Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
GREGOR KOSEC*, Jožef Stefan Institute, Slovenia

Medusa, a novel library for implementation of non-particle strong form mesh-free methods, such as GFDM or RBF-FD, is described. We identify and present common parts and patterns among many such methods reported in the literature, such as node positioning, stencil selection, and stencil weight computation. Many different algorithms exist for each part and the possible combinations offer a plethora of possibilities for improvements of solution procedures that are far from fully understood. As a consequence there are still many unanswered questions in the mesh-free community resulting in vivid ongoing research in the field. Medusa implements the core mesh-free elements as independent blocks, which offers users great flexibility in experimenting with the method they are developing, as well as easily comparing it with other existing methods. The article describes the chosen abstractions and their usage, illustrates aspects of the philosophy and design, offers some executions time benchmarks and demonstrates the application of the library on cases from linear elasticity and fluid flow in irregular 2D and 3D domains.

Q1 CCS Concepts: • **Mathematics of computing** → *Solvers*; Mathematical software performance; **Discretization**; **Numerical differentiation**; *Computations on matrices*; **Partial differential equations**;

Additional Key Words and Phrases: Strong form mesh-free methods, meshless methods, PDE, RBF-FD, object-oriented programming

ACM Reference format:
Jure Slak and Gregor Kosec. 2021. Medusa: A C++ Library for Solving PDEs Using Strong Form Mesh-free Methods. *ACM Trans. Math. Softw.* 47, 3, Article 28 (April 2021), 25 pages.
https://doi.org/10.1145/3450966

1 INTRODUCTION

Mesh-free (also called meshless) methods for solving **partial differential equations (PDEs)** arose in 1970s and are still an active topic of research in applied mathematics today. In mesh-free methods the computational domain is represented by a cloud of points instead of a mesh of elements, as is typical for mesh-based methods. The weak form mesh-free methods are most often

*Both authors contributed equally to this research.

The authors would like to acknowledge the financial support of the Slovenian Research Agency (ARRS) research core funding No. P2-0095 and the Young Researcher program PR-08346.

Authors' addresses: J. Slak, Jožef Stefan Institute, Jamova cesta 39, Ljubljana, Slovenia, 1000, Faculty of Mathematics and Physics, University of Ljubljana, Jadranska ulica 19, Ljubljana, Slovenia, 1000; email: jure.slak@ijs.si; G. Kosec, Jožef Stefan Institute, Jamova cesta 39, Ljubljana, Slovenia, 1000; email: gregor.kosec@ijs.si.

0098-3500/2021/04-ART28 $15.00
https://doi.org/10.1145/3450966

analogous to the well-established **Finite Element Method (FEM)**, while strong form methods are most often generalization of the **Finite Difference Methods (FDM)**.

Many strong form methods have been proposed throughout the years, starting from **Smooth Particle Hydrodynamics (SPH)** [Benz 1990], followed by generalizations of FDM with the **Finite Point method (FPM)** [Oñate et al. 2001], the Generalized Finite Differences method [Gavete et al. 2003], and **Radial basis function-generated Finite Differences (RBF-FD)** [Tolstykh and Shirobokov 2003], to name a few. A significant development in RBF-FD has been a recently reported by using polyharmonic RBFs augmented with monomials [Bayona et al. 2017] to avoid stagnation errors and allow control over the rate of convergence. Substantial development has also been reported in the stabilization of the method in convection dominated regimes [Shankar and Fogelson 2018], in adaptive solution of elliptic problems [Oanh et al. 2017], in methods for positioning computational nodes [Slak and Kosec 2019a], and in surface meshless methods [Petras et al. 2018; Suchde and Kuhnert 2019].

A number of mature software implementations exist for FEM, such as deal.II [Bangerth et al. 2007], DOLFIN (part of the FEniCS Project) [Logg and Wells 2010], FreeFem++ [Hecht 2012], LifeV [Bertagna et al. 2017], GetFEM [Renard and Poulios 2020], and Firedrake [Rathgeber et al. 2016], to name a few. Such a diverse ecosystem of general purpose implementations has not yet been developed for the field of strong-form meshless methods. There are implementations consisting of Matlab scripts and domain-specific applications, such as MFDMtool [Milewski 2013], GEC_RBFFD [Bayona et al. 2015], MFree2D [Liu 2002], RBFFD_GPU [Bollig 2014], and even a review paper by Nguyen et al. [2008] that specifically deals with computer implementation, includes its own set of Matlab scripts.

Extensible, tested, documented, and published general-purpose libraries for mesh-free methods that would facilitate further research and practical applications of the field are scarce. For older and established particle-based methods, such as SPH, high-quality software packages are available, both open source, such as DualSPHyiscs [Crespo et al. 2015], pySPH [Ramachandran et al. 2019], SPlisHSPlasH [Bender 2016], SWIFT [Schaller et al. 2018]; and commercial, such as SPH flow [Nextflow Software 2015], Fluidix [OneZero Software 2008], Pasimodo [Inpartik & ITM University of Stuttgart 2008], with both of the listings being nowhere near exhaustive. Another such package for particle-based methods is the Aboria library [Robinson and Bruna 2017]. Few commercial mesh-free implementations are known to the authors. One is the Midas MeshFree [MIDAS Information Technology Co. 2018] package, which uses the Implicit Boundary Method and a background integration grid to perform simulations, and claims to perform "finite element analysis." Another package is the MESHFREE software [Fraunhofer Gesellschaft 1999], which implements the Finite Pointset Method [Tiwari and Kuhnert 2003] and has an impressive suite of examples. Yet another package is the NOGRID [NOGRID GmbH 2006], which also implements the Finite Pointset Method. Some other commercial packages, such as Abaqus [Dassault Systèmes 2012] include mesh-free simulations as a part of their suite. However, they are not freely accessible and are usually focused on a single method. In 2014, Hsieh and Pan published ESFM: An essential software framework for meshfree methods [Hsieh and Pan 2014], which is an object-oriented C++ framework for computations using weak-form meshless methods and claims to be the first of its kind. However, it is not publicly available and the authors only show examples of linear elasticity problems in the paper. Another package to note is the RBF Python package [Hines 2015] (although not present in the standard Python Package Index), which implements RBF interpolation and RBF-based PDE solution techniques.

Many packages for PDE solving such as deal.II, DOLFIN, FreeFem++, LifeV, FetFEM, DualSPHyiscs, Aboria, and ESFM libraries use the C++ programming language. FreeFem++

implements its own extended language on top of C++ core, while FEniCS and Firedrake offer a Python interface. Nonetheless, C++ seems to be the language of choice for many such applications, with the potential of adding the bindings for other languages in the future. No open-source C++ library for dealing with strong form meshless methods is known to authors. Therefore, to help further research and development in the field of strong form meshless methods, we present an open source C++ library Medusa (http://e6.ijs.si/medusa).

Our team started the development of Medusa library in 2015 to support our research in the field [Kosec 2018; Kosec et al. 2019] and to ease implementation of applied solutions [Maksić et al. 2019]. Over time, the interface grew and matured, putting emphasis on modularity, extensibility and reusability. Similarly to listed FEM libraries, it relies heavily on the C++ template system and allows the programs to be written independently of the number of spatial dimensions with negligible runtime and memory overhead. Special care is also taken to increase expressiveness and to be able to explicitly translate mathematical notation into program source code. However, source code is still standard compliant C++, which allows the user to use entirety of the C++ ecosystem. The open-source nature of the library is a novelty compared to the other libraries.

The rest of the article is organized as follows: A brief overview of strong-form meshless methods is presented in Section 2, where the most common part of strong form meshless methods are identified and described. This is followed by the presentation of the library in Section 3, which also includes the relevant abstractions and rationale behind some design decisions. Two more interesting computational examples are presented in Section 4 with measurements of execution time presented along with comparison to FreeFem++ presented in Section 5.

2 STRONG FROM MESH-FREE METHODS

Similarly to many other methods, the general parts of the solution procedure for strong form mesh-free methods are:

(1) *Domain discretization*: The geometry of the spatial domain is discretized by placing computational nodes and finding their stencils. This part is described in more detail in Section 2.1.
(2) *Differential operator discretization*: The spatial partial differential operators are discretized using method specific techniques. This part is described in more detail in Section 2.2.
(3) *PDE discretization:* The remaining time-dependent part of the PDE is discretized and then solved either implicitly or explicitly, with time iteration, or by only solving the implicit sparse system once, for elliptic problems. This part is described in more detail in Section 2.3.

Even if the overall problem solution procedure is more complicated and involves coupled equations, additional physical models or non-linearities, such as in computational fluid dynamics, the above three parts represent the core of the solution procedure. From our experience, these parts and their components are the elements worthy of abstraction and general implementation.

A more detailed description of the three parts is given in the following subsections, with their respective implementations presented in Sections 3.1, 3.2, and 3.3.

2.1 Domain Discretization

A discretization of a bounded domain $\Omega \subset \mathbb{R}^d$ consists of N nodes $\mathcal{X} = \{p_0, p_2, \ldots, p_{N-1}\}$ placed in the interior and on the boundary of the domain. Each node is assigned a stencil (also called neighborhood or *support*) consisting of some nodes near it. We will denote the size of the stencil of ith node with n_i and the indices of stencil nodes with $\mathcal{I}(i) = (I_{i,1}, I_{i,2}, \ldots, I_{i,n_i})$. The stencil of

the ith node $\mathcal{N}(i)$ is the n_i-tuple

$$\mathcal{N}(i) = (p_{I_{i,1}}, p_{I_{i,2}}, \ldots, p_{I_{i,n_i}}). \tag{1}$$

Each node should be in its own stencil, and for simplicity, we assume that it is the first one, i.e., $I_{i,1} = i$ holds for all $i = 1, \ldots, N$. Boundary nodes are assigned outer unit normals $\vec{n}_i$.

Generation of nodal distributions has sometimes been considered as an easy and not too relevant first step. This is partly due to the fact that existing mesh generators could be used to generate a suitable mesh and the user can simply discard the connectivity information [Liu 2002]. Besides being conceptually flawed, such approach is also computationally wasteful and does not easily generalize to higher dimensions. Some authors even reported having difficulties to obtain node distributions of sufficient quality [Shankar et al. 2018].

Developed software packages usually support a combination of geometric primitives, which can be rotated, scaled, or translated, and also offer a way for working with real-world geometry, using, e.g., STL files. This is an approach taken by FreeFem++, Fenics and deal.II. The same is true for meshless packages, like DualSPHysics, and the node generation itself is described in detail in the documentation of the `GenCase` command. Other techniques for meshless node generation include iterative approaches [Liu et al. 2010], advancing front methods [Löhner and Oñate 2004], or sphere packing [Choi and Kim 1999]. An algorithm described in Drumm et al. [2008] has been successfully used with the Finite pointset method.

With the rising popularity of non-particle based meshless methods, research into suitable node generation algorithms also increased [Fornberg and Flyer 2015; Zamolo and Nobile 2018]. This includes two algorithms for variable density node generation in irregular domains in arbitrary dimensions: our original algorithm [Slak and Kosec 2019a] published in 2019 and another described in an arXiv preprint [van der Sande and Fornberg 2019]. Both of these algorithms are implemented in Medusa. In addition, Medusa also provides classic discretizations of basic geometric shapes, support for gridded nodes and an ability to easily define custom node generation schemes (e.g., hexagonal). Medusa also offers support for adding so-called "ghost nodes" to the boundary.

The remaining part of the discretization is to define the stencils, which is fully automated and considered part of the solution procedure in nearly all meshless methods. The most widely used type of stencils consists of some number of closest neighbors. Besides those, balanced stencils can be used in adaptive solutions [Oanh et al. 2017]. Both approaches are implemented in Medusa, along with the ability to only restrict the stencils to certain node types. It is also simple to define custom stencil selection algorithms, for example visibility-based stencils [Nguyen et al. 2008], as shown in Section 3.1.

2.2 Differential Operator Discretization

Most strong-form meshless approximations approximate a partial differential operator $\mathcal{L}$ at a point p with a linear functional $\boldsymbol{w}_{\mathcal{L},p}^{\mathsf{T}}$, using an approximation of the form

$$(\mathcal{L}u)(p) \approx \sum_{j \in \mathcal{I}(p)} (\boldsymbol{w}_{\mathcal{L},p})_j u(p_j) = \boldsymbol{w}_{\mathcal{L},p}^{\mathsf{T}} \boldsymbol{u}, \tag{2}$$

where point p is not necessarily one of the computational nodes. However, the stencil indices $\mathcal{I}(p)$ and stencil nodes $\mathcal{N}(p)$ represent computational nodes. The values $\boldsymbol{w}_{\mathcal{L},p}$ are called *stencil weights* or sometimes shape functions, as a legacy terminology originating from weak-form methods. Other approximations such as of Hermite type collocation [Li and Mulay 2013] are also possible, but less common.

We will describe two possibilities to obtain the stencil weights $\boldsymbol{w}_{\mathcal{L},p}$ that cover many meshless formulations and are also included in Medusa by default. The first is the **generalized weighted least squares (GWLS)** method, which includes many commonly used meshless approximations, such as SPH approximations [Benz 1990], Finite Point Method [Oñate et al. 2001], Generalized Finite Difference method [Gavete et al. 2003], **radial basis functions-generated finite differences (RBF-FD)** [Tolstykh and Shirobokov 2003], meshless local strong-form method [Slak and Kosec 2019b], Finite Pointset Method [Tiwari and Kuhnert 2003], diffuse approximate methods [Wang et al. 2012], and many more.

The second is a more specific **radial basis functions-generated finite differences (RBF-FD)** approximation with monomial augmentation, which also offers some speed improvements. Other custom approximation schemes can be implemented and used, such as schemes that put additional constraints on the center weights to achieve diagonal dominance in differentiation matrices [Suchde and Kuhnert 2019].

2.2.1 Generalized Weighted Least Squares. An approximation of function $u : \mathbb{R}^d \to \mathbb{R}$ around p^* is sought in the form

$$\hat{u}(p) = \sum_{i=1}^{m} \alpha_i b_i \left(\frac{p - p^*}{s} \right) = \boldsymbol{b} \left(\frac{p - p^*}{s} \right)^{\mathsf{T}} \boldsymbol{\alpha} \tag{3}$$

where $\boldsymbol{b} = (b_i)_{i=1}^m$ is a set of *basis functions*, $b_i : \mathbb{R}^d \to \mathbb{R}$, $\boldsymbol{\alpha} = (\alpha_i)_{i=1}^m$ are the unknown coefficients, and s is a positive scaling factor. For simplicity, we will assume that $I(p) = (1, \ldots, n)$ and $N(p) = (p_1, \ldots, p_n)$. Note that if monomials are chosen for b_i, then we obtain the same setup as for the standard **moving/weighted least squares (MLS/WLS)** formulation [Levin 1998].

Using the known values u_i in nearby nodes p_i, the error

$$e_i = \hat{u}(p_i) - u_i = \boldsymbol{b} \left(\frac{p_i - p^*}{s} \right)^{\mathsf{T}} \boldsymbol{\alpha} - u_i \tag{4}$$

can be computed. A weighted norm of the error vector $\boldsymbol{e} = (e_i)_{i=1}^n$ is then minimized. It can be expressed as

$$\|\boldsymbol{e}\|_{2,w}^2 = \sum_{i=1}^{n} (w_i e_i)^2 = \|W\boldsymbol{e}\|_2^2 = \|W(B\boldsymbol{\alpha} - \boldsymbol{u})\|_2^2, \tag{5}$$

where B is a rectangular matrix of dimensions $n \times m$ with rows containing basis function evaluated at points p_i:

$$B = \begin{bmatrix} b_1\left(\frac{p_1 - p^*}{s}\right) & \ldots & b_m\left(\frac{p_1 - p^*}{s}\right) \\ \vdots & \ddots & \vdots \\ b_1\left(\frac{p_n - p^*}{s}\right) & \ldots & b_m\left(\frac{p_n - p^*}{s}\right) \end{bmatrix} = \left[b_j \left(\frac{p_i - p^*}{s} \right) \right]_{j=1,i=1}^{m,n} = \left[\boldsymbol{b} \left(\frac{p_i - p^*}{s} \right)^{\mathsf{T}} \right]_{i=1}^{n}, \tag{6}$$

and W is a diagonal matrix of weights, $W_{ii} = \omega((p_i - p^*)/s)$, where $\omega : \mathbb{R}^d \to (0, \infty)$ is a *weight function*. Choosing $\omega \equiv 1$ gives the unweighted version. The arguments of b_j are shifted and scaled to ensure better conditioning of matrix B [Nguyen et al. 2008].

If we wanted to construct an approximant from known values of u_i, then we could just compute coefficients $\boldsymbol{\alpha}$ with standard methods for solving least square problems, such as normal equations with Cholesky decomposition, QR decomposition, or SVD decomposition. However, to obtain an approximation of $\mathcal{L}|_p$, we express $\boldsymbol{\alpha}$ in closed form using Moore-Penrose pseudoinverse as

$$\alpha = (WB)^{+} W\boldsymbol{u} \tag{7}$$

and substitute it in the definition (3) of $\hat{u}$, which becomes

$$\hat{u}(p) = \boldsymbol{b}\left(\frac{p-p^*}{s}\right)^{\mathsf{T}} (WB)^{+} W\boldsymbol{u}. \tag{8}$$

The value $(\mathcal{L}u)(p)$ can be approximated by applying operator $\mathcal{L}$ to $\hat{u}$, which gives

$$(\mathcal{L}u)(p) \approx (\mathcal{L}\hat{u})(p) = (\mathcal{L}\boldsymbol{b})\left(\frac{p-p^*}{s}\right)^{\mathsf{T}} (WB)^{+} W\boldsymbol{u} = \boldsymbol{w}_{\mathcal{L},p}^{\mathsf{T}}\boldsymbol{u}, \tag{9}$$

where the weights $\boldsymbol{w}_{\mathcal{L},p}^{\mathsf{T}}$ are computed as

$$\boldsymbol{w}_{\mathcal{L},p}^{\mathsf{T}} = (\mathcal{L}\boldsymbol{b})\left(\frac{p-p^*}{s}\right)^{\mathsf{T}} (WB)^{+} W. \tag{10}$$

Note that the computation of Moore-Penrose pseudoinverse is not really necessary, since $\boldsymbol{w}_{\mathcal{L},p}^{\mathsf{T}}$ can be computed by first solving the (possibly) underdetermined system

$$(WB)^{\mathsf{T}} y = (\mathcal{L}\boldsymbol{b})((p-p^*)/s) \tag{11}$$

for y and then computing $\boldsymbol{w}_{\mathcal{L},p} = Wy$. System (11) can be solved using QR, SVD or any other appropriate decomposition, however, depending on m, n, and properties of b_j, it can even be square and positive definite, making it possible to use Cholesky, LDL$^{\mathsf{T}}$, or LU decompositions.

2.2.2 Radial Basis Function-generated Finite Differences with Monomial Augmentation. We again consider a partial differential operator $\mathcal{L}$ at a point p of form

$$(\mathcal{L}u)(p) \approx \sum_{j=1}^{n} (\boldsymbol{w}_{\mathcal{L},p})_j u(p_j) = \boldsymbol{w}_{\mathcal{L},p}^{\mathsf{T}}\boldsymbol{u}, \tag{12}$$

where p_i are the neighboring nodes to p. The unknown weights in approximation (12) can be computed by enforcing equality for n basis functions. A natural choice are monomials, which are also used in FDM, resulting in the Finite Point Method [Oñate et al. 2001].

In the RBF-FD discretization the equality is satisfied for radial basis functions ϕ_j, which are functions

$$\left\{\phi_j(p) = \phi\left(\left\|\frac{p-p^*}{s} - \frac{p_j-p^*}{s}\right\|\right) = \phi\left(\frac{\|p-p_j\|}{s}\right),\ j = 1,\dots,n\right\}, \tag{13}$$

generated by a radial function $\phi : [0,\infty) \to \mathbb{R}$ and defined over the set of nearby centers p_j. The center of the coordinate system is once again shifted to p^* and distances are scaled by $s > 0$ to improve conditioning.

Each ϕ_j, for $j = 1,\dots,n$ gives rise to one linear equation

$$\sum_{i=1}^{n} w_i \phi_j(p_i) = (\mathcal{L}\phi_j)\left(\frac{p-p^*}{s}\right), \tag{14}$$

for unknowns w_i obtained by substituting ϕ_j for u in Equation (2). These equations form the following linear system:

$$\begin{bmatrix} \phi\left(\frac{\|p_1-p_1\|}{s}\right) & \cdots & \phi\left(\frac{\|p_n-p_1\|}{s}\right) \\ \vdots & \ddots & \vdots \\ \phi\left(\frac{\|p_1-p_n\|}{s}\right) & \cdots & \phi\left(\frac{\|p_n-p_n\|}{s}\right) \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} (\mathcal{L}\phi_1)\left(\frac{p-p^*}{s}\right) \\ \vdots \\ (\mathcal{L}\phi_n)\left(\frac{p-p^*}{s}\right) \end{bmatrix}, \tag{15}$$

where ϕ_j have been expanded for clarity. The above system can be written more compactly as

$$A\boldsymbol{w} = \boldsymbol{\ell}_\phi. \tag{16}$$

The matrix A is symmetric, and for some ϕ even positive definite. Other approximation properties are also well studied [Wendland 2004]. Additionally, the computation up to now is the same as using GWLS with $n = m$ and $b_j = \phi_j$.

To ensure consistency up to a certain order, the computation can be augmented with monomials. Let $q_1, \ldots, q_l$ be polynomials forming the basis of the space of d-dimensional multivariate polynomials up to and including total degree m, with $l = \binom{m+d}{d}$.

Additional constraints are enforced by extending Equation (16) as

$$\begin{bmatrix} A & Q \\ Q^\mathsf{T} & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{w} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\ell}_\phi \\ \boldsymbol{\ell}_q \end{bmatrix}, \quad Q = \begin{bmatrix} q_1(p_1) & \cdots & q_l(p_1) \\ \vdots & \ddots & \vdots \\ q_1(p_n) & \cdots & q_l(p_n) \end{bmatrix}, \quad \boldsymbol{\ell}_q = \begin{bmatrix} (\mathcal{L}q_1)(p^*) \\ \vdots \\ (\mathcal{L}q_l)(p^*) \end{bmatrix}, \tag{17}$$

where Q is a $n \times l$ matrix of polynomials evaluated at nodes p_i and $\boldsymbol{\ell}_q$ is the vector of values assembled by applying considered operator $\mathcal{L}$ to the polynomials at p^*.

Weights obtained by solving Equation (17) are taken as values for $\boldsymbol{w}_{\mathcal{L},p}$, while values $\boldsymbol{\lambda}$ are discarded.

2.3 PDE Discretization

With stencil weights $\boldsymbol{w}_{\mathcal{L},p}$ computed, they are mostly used in two main patterns. The first is to explicitly approximate $(\mathcal{L}u)(p)$, with the field u being known, such as in explicit time iteration, during linearization of nonlinear PDEs, or simply to obtain a derivative of the field. The second is in implicit form, when we wish to obtain a field u, such that the field values satisfy a set of linear equations. This usually happens when solving elliptic problems or during time iteration with at least partially implicit methods, such as Crank-Nicholson and implicit Euler's method.

Both usage patterns are described on typical examples in low-level detail in the following sections. We judged that these patterns of spatial approximation are common enough that suitable abstractions are offered in Medusa (see Section 3.3) to avoid error-prone handling of indices, code repetition, and poor readability.

2.3.1 Explicit Evaluation. Consider a sample time-dependent initial value problem on domain Ω

$$\frac{\partial u}{\partial t}(p,t) = (\mathcal{L}u)(p,t) \qquad \text{in } \Omega, \tag{18}$$

$$u(p,t) = f(p,t) \qquad \text{at } t = 0, \tag{19}$$

$$u(p,t) = g_d(p,t) \qquad \text{on } \Gamma_d, \tag{20}$$

$$\frac{\partial u}{\partial \vec{n}}(p,t) = g_n(p,t) \qquad \text{on } \Gamma_n, \tag{21}$$

where Γ_d and Γ_n are Dirichlet and Neumann boundaries, respectively, and f, g_d, and g_n are known functions. Using explicit Euler scheme in time, starting at $t = 0$ with timestep Δt, we define $u_i^k = u(p_i, k\Delta t)$. Time iteration using strong form meshless approximations is performed as follows:

$$u_i^0 = f(p_i), \tag{22}$$

$$u_i^{k+1} = u_i^k + \Delta t \left(\boldsymbol{w}_{\mathcal{L},p_i}^\mathsf{T} u_{I(i)}^k \right), \quad \text{for internal nodes } p_i, \tag{23}$$

$$u_i^{k+1} = g_d(p_i, (k+1)\Delta t), \quad \text{for Dirichlet nodes } p_i, \tag{24}$$

$$u_i^{k+1} = \frac{g_n(p_i, (k+1)\Delta t) - \sum_{j=2}^{n_i} u_{I_{i,j}}^k \sum_{\ell=1}^{d} n_\ell (\boldsymbol{w}_{\partial_\ell, p_i})_j}{\sum_{\ell=1}^{d} n_\ell (\boldsymbol{w}_{\partial_\ell, p_i})_1}, \text{ for Neumann nodes } p_i, \tag{25}$$

where Neumann boundary conditions are obtained by equating the discretized version (28) to g_n and expressing u_i. Explicit discretization of Neumann boundary conditions is obtained by approximating coordinate partial derivatives with their discrete versions

$$\frac{\partial u}{\partial \vec{n}}(p_i, t) = \sum_{\ell=1}^{d} n_\ell (\partial_\ell u)(p_i) \approx \sum_{\ell=1}^{d} n_\ell \boldsymbol{w}_{\partial_\ell, p_i}^{\mathsf{T}} u_{I(i)} = \sum_{\ell=1}^{d} n_\ell \sum_{j=1}^{n_i} (\boldsymbol{w}_{\partial_\ell, p_i})_j u_{I_{i,j}} \tag{26}$$

$$= \sum_{\ell=1}^{d} n_\ell \sum_{j=1}^{n_i} (\boldsymbol{w}_{\partial_\ell, p_i})_j u_{I_{i,j}} = \sum_{j=1}^{n_i} u_{I_{i,j}} \sum_{\ell=1}^{d} n_\ell (\boldsymbol{w}_{\partial_\ell, p_i})_j = \tag{27}$$

$$= u_i \sum_{\ell=1}^{d} n_\ell (\boldsymbol{w}_{\partial_\ell, p_i})_1 + \sum_{j=2}^{n_i} u_{I_{i,j}} \sum_{\ell=1}^{d} n_\ell (\boldsymbol{w}_{\partial_\ell, p_i})_j, \tag{28}$$

where we used $I_{i,1} = i$ and $u_{I(i)}$ is the vector of function values in stencil nodes $u_{I(i)} = (u(p_j))_{j \in I(i)}$.

The Equations (22–25) contain explicit evaluations of meshless discretizations on known fields. Similar expressions, containing the same explicit evaluations can be obtained for other time discretizations or for vector functions u.

2.3.2 Implicit Solution. Consider a boundary value problem

$$\mathcal{L}u = f \qquad \text{in } \Omega, \tag{29}$$

$$u = g_d \qquad \text{on } \Gamma_d, \tag{30}$$

$$\frac{\partial u}{\partial \vec{n}} = g_n \qquad \text{on } \Gamma_n, \tag{31}$$

where Γ_d and Γ_n are Dirichlet and Neumann boundaries, respectively, and f, g_d, and g_n are known functions. Each of the above equations is approximated by a linear equation in corresponding computational nodes. The system of linear equations can be written as $Mu = r$, where ith row of the system corresponds to the equation that holds in node p_i. Formally, the matrix M and right-hand side r are given by

$$M_{i,I_{i,j}} = (\boldsymbol{w}_{\mathcal{L},p_i})_j, \quad \text{for } j = 1, \ldots, n_i, \qquad r_i = f(p_i), \qquad \text{for internal nodes } p_i, \tag{32}$$

$$M_{i,i} = 1, \qquad r_i = g_d(p_i), \qquad \text{for Dirichlet nodes } p_i, \tag{33}$$

$$M_{i,I_{i,j}} = \sum_{\ell=1}^{d} n_\ell (\boldsymbol{w}_{\partial_\ell, p_i})_j, \quad \text{for } j = 1, \ldots, n_i, \qquad r_i = g_n(p_i), \qquad \text{for Neumann nodes } p_i. \tag{34}$$

Matrix M is a sparse matrix with at most $\sum_{i=1}^{N} n_i$ nonzero entries. Solution of the system $Mu = r$ is the numerical approximation of u.

The Equations (32–34) define the unknown field u implicitly by using stencil weights. Similar approximations can be obtained for vector equations, or in implicit timestepping schemes.

3 SOFTWARE DESCRIPTION

Looking at existing finite element software packages and based on our experience with implementing strong-form meshless PDE solution procedures, we isolated a set of implementation requirements:

- *Modularity.* Ability to change approximation, node generation, stencil selection, and other algorithms is of crucial importance for fast prototyping that is needed in research. The goal of Medusa is that different reported meshless methods can be rapidly constructed by using different combinations of provided classes.
- *Dimension independence.* The mathematical PDE formulation is independent of the dimension of the problem, and we strive to conserve this property in the implementation as well. Implemented approximations, node placing algorithms, and operators can be used in any domain dimensionality simply by changing a template parameter, e.g., there is virtually no difference between code for solution of problem in 2D or 3D, or any other dimensionality.
- *Extensibility.* Allowing users to define their own shapes, approximations, and operators enables wide applicability, e.g., implementing additional stabilizations such as upwind or hyperviscosity is straightforward.
- *Composability.* The library can be used with other already existing infrastructure, such as ODE or nonlinear solvers, or HPC infrastructure, such as PETSc or Trilinos.
- *Readability.* A clear mapping from mathematical notation to code helps reduce errors in the code. Additionally, dealing with objects representing abstract concepts such as operators, vector fields, and domains directly instead of matrices and lists of indices also helps avoid bugs.
- *Small overhead due to the abstraction*: The runtime has small and often negligible overheads in comparison with "bare-bones" implementations.
- *Parallelization.* When possible, parallelization can be handled internally, so the program can remain relatively unchanged if the user decides for parallel execution.
- *Ease of use.* This involves easy import and export of common file formats, access to examples, and technical documentation.

We designed the Medusa library with above requirements in mind, and although still lacking in some of the above requirements, such as distributed parallelism, we believe it is a useful tool. The library is written in C++ using object-oriented approach and C++'s strong template system to achieve modularity, extensibility, and dimension independence. The library has no requirements, apart from the C++ standard library and optionally the HDF5 C library [Folk et al. 2011] for reading and writing binary HDF5 files. However, we include four open-source third-party libraries, namely, the Eigen [Guennebaud et al. 2010] library for linear algebra, nanoflann [Blanco and Rai 2014] library for spatial-search structures, tinyformat [Foster et al. 2011] library for simple formatting, and RapidXML [Kalicinski 2011] for XML file processing. These four libraries have been packaged together with Medusa source code for simplicity. An external version of Eigen can be easily used as well.

Medusa is licensed under MIT license, but the included libraries Eigen, nanoflann, tinyformat, and RapidXML are licensed under Mozilla Public License (v. 2.0), BSD license, Boost Software License, and dual Boost Software license/MIT license, respectively. The repository also includes the Google test library, which is licensed under BSD 3-Clause "New" or "Revised" License, but is used for unit testing purposes and not necessary for core functionality.

The official website of the library is http://e6.ijs.si/medusa. The library is developed using the `git` versioning system and the development is ongoing on GitLab https://gitlab.com/e62Lab/

medusa. The library uses `cmake` build system and can be used as a `cmake` submodule or as a standard standalone static C++ library. Long compile times associated with large amounts of C++ templates are somewhat mitigated by separating declarations from template definitions into `Medusa_fwd.hpp` and other included files, explicitly instantiating most common class instances and linking them. If other instances are desired, then they can be explicitly instantiated or full template definitions available in `Medusa.hpp` can be included.

Quality of implementation is ensured through continuous integration, which build the library and runs its test suite, documentation generation tools, linters, and compiles and runs all examples. This aims to minimize the risk of regressions, stale documentation, or examples and ensures code validity, uniform code style, and validity of system dependencies. The library also includes numerous assertions, which can be disabled at compile time, that help catch errors earlier in the debugging phase. We use Google test testing framework to develop and run over 300 tests. The de facto standard documentation generation tool Doxygen is used to generate the technical documentation, which is available at http://e6.ijs.si/medusa/docs. The `cpplint` style and code checker is used. Additionally, our wiki page is available at http://e6.ijs.si/medusa/wiki, where more detailed explanations of examples, the theory behind the methods, practical applications, and further information about development and potential building issues can be found.

The following section describes main modules of Medusa, dealing with domains, approximations, and PDE discretization. Almost all core classes are templated using a `vec_t` type, which contains two essential pieces of information: the dimension of the computational domain (`vec_t::dim`) and the scalar type used for numerical computations (`vec_t::scalar_t`), e.g., `float` or `complex<double>`.

3.1 Domains

The main class representing domain discretizations is the `template <class vec_t> DomainDiscretization` class, which closely resembles the description of domain discretizations given in Section 2.1. It includes a list of d-dimensional points p_i, each one has an associated integer *type* τ_i, with positive τ_i for internal nodes and negative τ_i for boundary nodes ($\tau_i = 0$ is reserved). The types can be assigned by the user to differentiate between different types of the boundary, with, e.g., -1 representing the Dirichlet and -2 the Neumann boundary. All boundary nodes also have their outer unit normals $\vec{n}_i$ stored. Additionally, stencil indices $I(p_i)$ are stored for each node. Stencils of varying sizes are supported.

Domain discretizations can be constructed by discretizing one of the predefined shapes, including d-dimensional spheres, cubes, 2d polygons, 3d polyhedra (given by STL files), as well as their unions, differences, translations, and rotations. Most of them support discretization of boundaries with arbitrary spacing function h. For discretizations of domain interiors, two dimension independent variable density node generation algorithms are implemented, `GeneralFill` and `GrainDropFill`, based on Slak and Kosec [2019a] and van der Sande and Fornberg [2019], respectively. Other node generation algorithms, such as grid-based fills and surface filling algorithms, are also available.

Two stencil selection algorithms are also available, `FindClosest`, which constructs stencils using the indices of defined number of closest nodes, and `FindBalancedSupport`, which also ensures that stencils are balanced around the central node.

Listing 1 demonstrates some of the capabilities for creating and handling domains. Figure 1 shows the domains produced by the source code in Listing 1. The left part shows a 2D domain with relatively coarse variable density discretization, with interior and boundary nodes and also shows stencils for a few selected nodes. The right part shows a uniform discretization of a 3D model, obtained from an STL file.

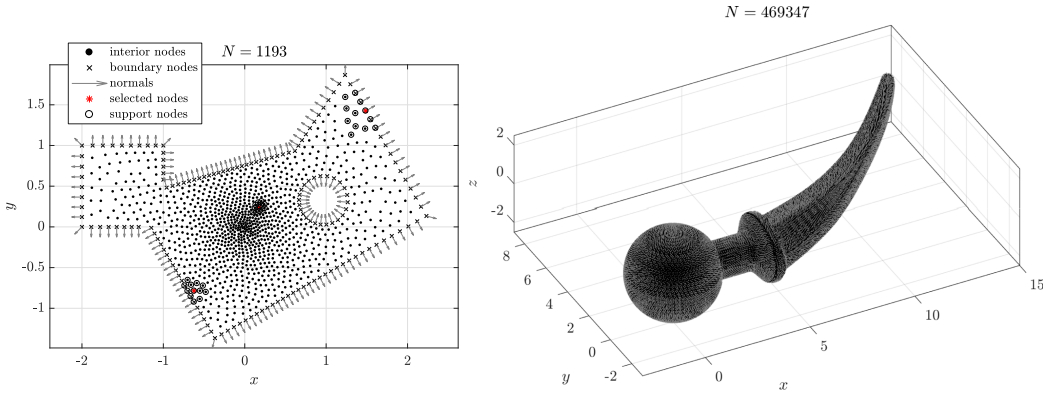

Fig. 1. Domain discretizations produced by Listing 1. A few selected nodes are shown along with their support nodes in the left figure. The right figure shows a denser discretization of a STL model.

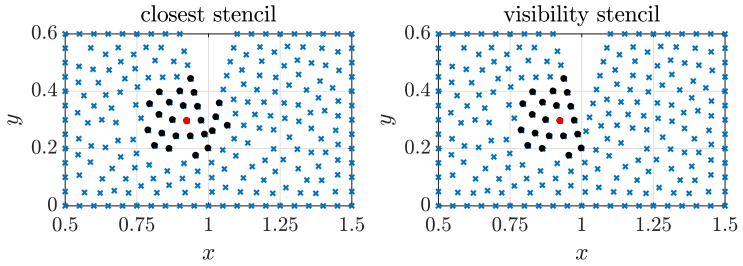

Fig. 2. A comparison of closest-point and visibility based stencils.

```
PolygonShape<Vec2d> poly({{-1, -1}, {2, -1}, {2, 1}, {1, 0.5}, {-1, 1}});
BallShape<Vec2d> ball({1, -0.2}, 0.3);
auto shape = (poly - ball).rotate(PI/6) + BoxShape<Vec2d>({-2, 0}, {-1, 1});
auto h = [](const Vec2d& p) { return 0.025 + p.norm()/20; };
DomainDiscretization<Vec2d> domain = shape.discretizeBoundaryWithDensity(h);
GeneralFill<Vec2d> fill; fill.seed(1);
fill(domain, h);
domain.findSupport(FindClosest(12));

STLShape<Vec3d> stl_shape(STL::read("../data/hip.stl"));
double dx = 0.05;
DomainDiscretization<Vec3d> stl_domain = stl_shape.discretizeBoundaryWithStep(dx);
auto [bot, top] = stl_shape.bbox();
GrainDropFill<Vec3d> gfill(bot, top);
gfill.seed(1).initialSpacing(0.01).maxPoints(1e7);
gfill(stl_domain, dx);
```

Listing 1. Construction and discretization of domains.

The domain discretization can also be constructed or modified manually, by `addInternalNode`, `addBoundaryNode`, or `removeNodes` methods. Similarly, custom stencils can be constructed manually by assigning the stencils for each node. Custom stencil selection algorithms can also be defined in a reusable manner as a callable that accepts the domain and constructs its stencils. Such callable can be supplied to the `findSupport` method. Listing 2 shows a definition of a simple algorithm for computing visibility based stencils and a comparison with closest point stencils is shown in Figure 2. The example is taken from the customization examples included in the Medusa repository.

```
struct VisibilityStencil {
    int num_closest;
    explicit VisibilityStencil(int num_closest) : num_closest(num_closest) {}

    /// Check for straight-line visibility by testing intermediate points with some density.
    template <typename vec_t>
    bool visible(const vec_t& p, const vec_t& q, const DomainDiscretization<vec_t>& domain) const {
        int density = 100;
        vec_t increment = (q-p) / density;
        for (int i = 1; i < density; ++i) {
            if (!domain.contains(p + i*increment)) return false;
        }
        return true;
    }

    /// Method responsible for stencil construction.
    template <typename vec_t>
    void operator()(DomainDiscretization<vec_t>& domain) const {
        int N = domain.size();
        KDTree<vec_t> tree(domain.positions());
        for (int i = 0; i < N; ++i) {
            Range<int> indices = tree.query(domain.pos(i), num_closest).first;
            for (int j : indices) {
                if (visible(domain.pos(i), domain.pos(j), domain)) {
                    domain.support(i).push_back(j);
} } } } };
```

Listing 2. Definition of a visibility-based stencil selection algorithm.

3.2 Approximations

The library currently includes two approximation engines for computing that implement the procedures described in Section 2.2. These are `template <class basis_t, class weight_t, class scale_t, class solver_t> class WLS`, `template <class rbf_t, class vec_t, class scale_t, class solver_t> class RBFFD` , with reasonable defaults for last few parameters. Template parameters allow for various combinations of basis functions b_j, RBFs ϕ, weight functions ω, scaling function s, and solvers to be used. By default, the library includes monomial and RBF bases, Gaussian, Multiquadric, Inverse multiquadric, and Polyharmonic RBFs, three scaling functions, various weights, and a variety of solvers included with Eigen. It is also easy for users to add their own RBFs, weights, and bases. Since templates offer a (static) version of duck typing, any class with the interface conforming to the, e.g., RBF concept as described in the documentation, can be used.[1] Examples of custom definitions are available in the examples that come with Medusa.

The power of this generality is shown in Figure 3, where errors of various approximation setups are shown. The Laplacian operator was approximated on a regular grid G_h of points with spacing h covering the unit square $[0, 1]^2$. The error of the approximation was computed as

$$e_h = \max_{p_i \in G_h} \left| \boldsymbol{w}_{\nabla^2, p_i}^{\mathrm{T}} \boldsymbol{u}_{I(i)} - (\nabla^2 u)(p_i) \right|.$$

The test function was chosen to be $u(x, y) = \sin(\pi x) \sin(\pi y)$. Five different approximation setups were tested:

[1] http://e6.ijs.si/medusa/docs/html/concepts.html.

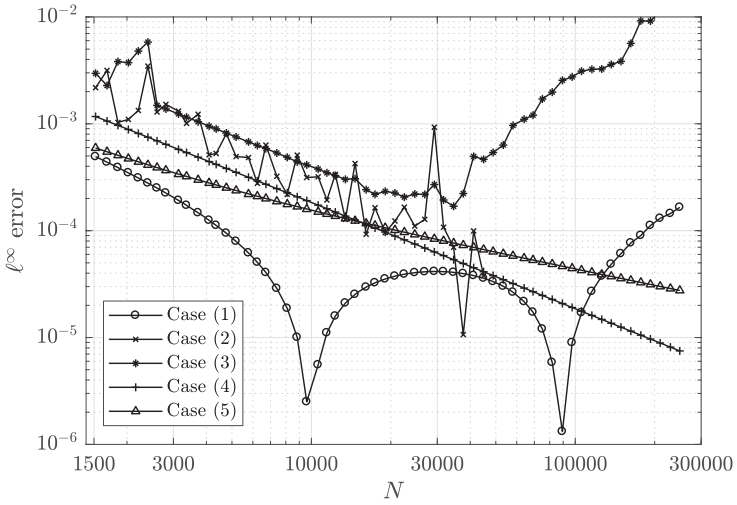

Fig. 3. Error of approximating the Laplacian with different approximation setups. Less than 1 minute of computing time was needed to produce the data for this plot.

(1) RBF-FD with Gaussian RBFs $b_j(p) = \exp(\|p - p_j\|^2/\sigma^2)$, using stencil of $n = 9$ closest nodes with no monomial augmentation, $\sigma = 100$, and with scaling s equal to the distance to the nearest neighbor. LU decomposition was used to solve the system for stencil weights.
(2) Like (1), but with $\sigma = 5$ and without scaling ($s = 1$).
(3) Like (2), but with SVD decomposition.
(4) GWLS with $m = 5$ monomial basis functions up to order 2, $n = 9$ closest nodes, Gaussian weight with $\sigma = 1$, scaling to closest node, and SVD decomposition.
(5) RBF-FD with polyharmonic splines $\phi(r) = r^5$ and monomial augmentation of order $m = 2$ with $n = 12$ closest nodes.

The definition of these setups in Medusa is shown in Listing 3. Stencil sizes are not included, as their computation was already shown in Listing 1.

```
RBFFD<Gaussian<double>, Vec2d, ScaleToClosest, PartialPivLU<MatrixXd>> approx1(100.0);
RBFFD<Gaussian<double>, Vec2d, NoScale, PartialPivLU<MatrixXd>> approx2(5.0);
RBFFD<Gaussian<double>, Vec2d, NoScale, JacobiSVDWrapper<double>> approx3(5.0);
WLS<Monomials<Vec2d>, GaussianWeight<Vec2d>, ScaleToClosest> approx4(2, 1.0);
RBFFD<Polyharmonic<double, 5>, Vec2d, ScaleToClosest, PartialPivLU<MatrixXd>> approx5({}, 2);
```

Listing 3. Definition of various important approximations.

These setups present some of the problems and answers in meshless strong form methods in recent years. The question of choice of the shape parameter for RBFs is a long-standing one, since the shape parameter often presents a tradeoff between accuracy and the condition number of the matrix A [Wendland 2004]. Case (2) exhibits the expected behavior that Gaussian approximations converge until the condition number is too high, and numerical errors become predominant. Jagged behavior can be smoothed by using SVD decomposition, however, the overall outcome is the same. A simple remedy for this instability is to scale the shape parameter (or the space) to keep the condition number constant. This solves the problems with numerical instability but causes the approximation to diverge in a characteristic fashion with two local minimums [Bayona

et al. 2010]. This lack of convergence is also often called divergence due to "stagnation errors." Two more convergent cases are included: one is the Finite point method (Case (4)), which achieves similar behavior and accuracy to FDM [Oñate et al. 2001], and another is RBF-FD using PHS augmented with monomials (Case (5)) [Bayona et al. 2017], where accuracy and convergence order can be easily controlled through augmentation.

3.3 Operators

This module defines one of the core functions of the library, which takes a domain discretization with nodes p_i, an approximation engine and a list of operators $(\mathcal{L}_1, \ldots, \mathcal{L}_\ell)$, and computes and stores stencil weights $(\boldsymbol{w}_{\mathcal{L}_j,p_i})_{i=1,j=1}^{N,\ell}$ for all operators and all computational nodes in the domain. These weights are stored in a `ShapeStorage` class.

The library supports computing shapes for first and second derivatives, as well as for the Laplacian operator. Note that this allows for construction of arbitrary second-order operators as

$$\boldsymbol{w}_{\mathcal{L},p} = \sum_{1\leq|\alpha|\leq 2} a_\alpha(p)\boldsymbol{w}_{\partial_\alpha,p}, \text{ for } \mathcal{L} = \sum_{1\leq|\alpha|\leq 2} a_\alpha(p)\frac{\partial}{\partial x^\alpha}, \tag{35}$$

where $|\alpha| = \sum_{i=1}^d \alpha_i$ and $\frac{\partial}{\partial x^\alpha} = \frac{\partial^{|\alpha|}}{\partial x_1^{\alpha}\cdots x^{\alpha_d}}$ are the standard multiindex notations. This would also cover the Laplacian operator, however, Equation (35) is not necessarily the most efficient nor the most numerically stable way of computing the Laplacian for certain basis functions. User-defined operators are supported as well, with the only requirement being that the user implements application of the operator for a class of basis functions that is used in their code. Our examples include solving the biharmonic equation to demonstrate this extensibility (see `examples/customization/custom_operators_biharmonic.cpp`).

The `ShapeStorage` class stores the computed weights for a chosen set of operators in space-efficient way. These shapes can be used to implicitly express or explicitly compute $\mathcal{L}u$, as described in Sections 2.3 and its subsections.

For given scalar or vector field u, we directly support most common scalar and vector operators, such as coordinate derivatives of first and second order, Laplacian, gradient, divergence, gradient of divergence, directional derivatives, as well as any user-defined operators.

Two examples of PDE solutions will be given in this section to illustrate the functionality of the library for explicit and implicit solving. Special effort was put into readability of the solution procedures to give the user a direct mapping from the mathematical solution procedure to the source code.

3.3.1 *Explicit Operators.*

Consider the problem of type (18–21):

$$\frac{\partial u}{\partial t}(x,y,t) = \nabla^2 u + 5 \qquad \text{in } \Omega, \tag{36}$$

$$u(x,y,t) = 0 \qquad \text{at } t = 0, \tag{37}$$

$$u(x,y,t) = x \qquad \text{on } \Gamma_d, \tag{38}$$

$$\frac{\partial u}{\partial \vec{n}}(x,y,t) = 0 \qquad \text{on } \Gamma_n, \tag{39}$$

on the 2D domain Ω constructed in Listing 1, where Γ_n is the inner circle boundary and Γ_d the outer boundary. The problem is solved in Listing 4 and the solution procedure follows (22–25). The solution is shown on the left side of Figure 4.

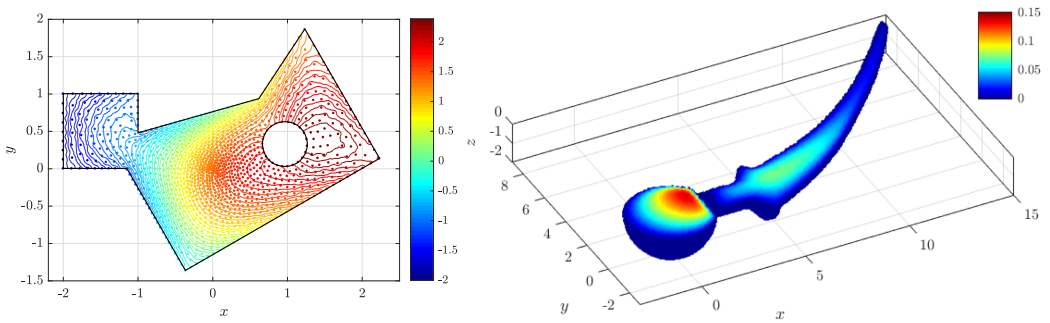

Fig. 4. Solution of the heat equations (36–39) on the left and convection-diffusion problem (40) on the right.

Listing 4 begins after the domain has been constructed and the sets of indices `interior`, `boundary`, and `circle`, corresponding to the interior, outer boundary in inner boundary nodes, respectively, have been defined. We use the WLS approximation with nine Gaussians on stencils of 13 nodes and a Gaussian weight. The `computeShapes` method computes shapes for Laplacian and first derivatives, which are then stored. The explicit operators `op` are a collection of methods that implement the spatial parts of the formulas (22–25) from Section 2.3.1 and greatly help with the readability of the solution procedure. They operate on dense scalar or vector fields that conform to a specified interface, so even `std::vector<vec_t>` can be used, as long as the type `vec_t` supports basic arithmetic operations. The temporal part of the equation can be solved with more advanced methods than the explicit Euler method used in Listing 4, either by using ODE integrators included in Medusa or by using an external solver, such as odeint [Ahnert and Mulansky 2011] or SUNDIALS [Hindmarsh et al. 2005].

```
d.findSupport(FindClosest(15));
FindClosest ss(15); ss.forNodes(circle).searchAmong(interior).forceSelf(true);
d.findSupport(ss);  // more complex stencil selection

WLS<Gaussians<Vec2d>, GaussianWeight<Vec2d>, ScaleToClosest> approx({9, 100.0}, 1.0);

auto storage = d.computeShapes<sh::lap|sh::d1>(approx);
auto op = storage.explicitOperators();

ScalarFieldd u1(N), u2(N);
u1.setConstant(0);
for (int i : boundary) {
    u1[i] = u2[i] = d.pos(i, 0);
}
double dt = 1e-4, max_t = 2.5;
int t_steps = max_t / dt;
for (int t = 0; t < t_steps; ++t) {
    for (int i : interior) {
        u2[i] = u1[i] + dt*(op.lap(u1, i) + 5.0);
    }
    for (int i : circle) {
        u2[i] = op.neumann(u1, i, d.normal(i), 0.0);
    }
    u2.swap(u1);
}
```

Listing 4. Solving the heat equations (36–39) explicitly.

459 *3.3.2 Implicit Operators.* Consider a boundary value problem of type (29–31):

$$-2\nabla^2 u + 8\,(2,1,-1)\cdot\nabla u = 1 \text{ in } \Omega, \quad u = 0 \text{ on } \partial\Omega, \tag{40}$$

460 where Ω is the right domain in Figure 1. The Listing 5 shows the source code needed to solve the
461 problem implicitly, as described in Section 2.3.2.

```
auto d = DomainDiscretization<vec_t>::load(domain_file, "domain2");
int N = d.size();
d.findSupport(FindClosest(45));
RBFFD<Polyharmonic<double, 3>, vec_t, ScaleToClosest> approx({}, 2);
auto storage = d.computeShapes<sh::lap|sh::d1>(approx);

Eigen::SparseMatrix<double, Eigen::RowMajor> M(N, N);
M.reserve(storage.supportSizes());
Eigen::VectorXd rhs = Eigen::VectorXd::Zero(N);
auto op = storage.implicitOperators(M, rhs);
for (int i : d.interior()) {
    -2.0*op.lap(i) + 8.0*op.grad(i, {2.0, 1.0, -1}) = 1.0;
}
for (int i : d.boundary()) {
    op.value(i) = 0.0;
}
Eigen::PardisoLU<decltype(M)> solver(M);
Eigen::VectorXd u = solver.solve(rhs);
```

462 Listing 5. Solving convection-diffusion equation (40) implicitly.

463 For approximation, we use the Polyharmonic RBFs with augmentation of order 2 on
464 stencils of 45 closest nodes. After computing the weights, the appropriately allocated sparse
465 matrix and right side are assembled. The matrix is defined and provided to Medusa by
466 the user and is assumed to be correctly preallocated. Medusa offers a helper for that,
467 as shown in Listing 5. By giving the user control of the matrix, additional equations
468 can be inserted in the matrix, which can be useful when solving problems with addi-
469 tional constraints or coupled problems. An example of both of these are included in the
470 Medusa's example suite (see `examples/coupled_domains/poisson_coupled_domains.cpp` and
471 `examples/poisson_equation/poisson_neumann_2D.cpp`).
472 The implicit operators `op` hold a reference to the matrix and the right-hand side and fill them
473 with the appropriate weights, taken from `storage`, implementing formulas (32–34). This is done
474 to improve readability; note the similarity between the line of the source code, which defines the
475 equation in the interior, and the Equation (40). The implicit system can also be amended manually,
476 if desired. The intuitive mathematical syntax supports expressions of form $\sum_\ell \alpha_\ell \mathcal{L}_\ell u = r$, where
477 0th, 1st, and 2nd derivatives are supported for $\mathcal{L}_\ell$, as well as directional derivatives, gradients of
478 divergence, Laplacian, and any user-defined operators (an example of the user-defined Biharmonic
479 operator from Medusa's example suite also shows its usage in implicit solving). Another benefit of
480 this system is that (in `DEBUG` mode) checks are performed that the operators added together always
481 write to the same matrix row, to avoid indexing errors.
482 Overall, the abstractions for implicit and explicit operators are in our opinion one of the best
483 features of Medusa library. They allow the user to think in terms of field and operators, instead
484 in terms of arrays and indices, which are much more error-prone. This shift has been present in
485 FEM implementations for a while, with FreeFem++, Fenics, and deal.II implementing these types of
486 abstractions; however, it has not been noted in the strong form community, and finite difference
487 codes are often riddled with poorly readable discretization code, clouding the overall problem

solution procedure. Munthe-Kaas and Haveraaen in 1996 introduced the concept of coordinate free numerics [Munthe-Kaas and Haveraaen 1996], which encompasses this idea, and Medusa has been investigated in this direction as well [Slak and Kosec 2018].

3.4 Miscellaneous

There are a few additional modules in the library that simplify its usage or offer often needed utilities. The "types" module implements nicer interfaces and additional functionality to types used to represent (physical) vectors, scalar fields, vector fields, and containers, while retaining full compatibility with Eigen. Input and output capabilities from and to CSV, XML, and HDF file formats are supported. Some basic integrators for solving ODEs, such as RK4, are also included.

4 EXAMPLES

Plenty of examples are included in the Medusa repository (see the `examples/` folder) and a tutorial for solving the Poisson equation is available on the website. The examples included in the repository offer many different setups for solving Poisson boundary value problems, which are used to demonstrate different features. Additionally, the repository includes examples solving problems from electromagnetic scattering, which includes support for complex numbers, Navier-Stokes equations for fluid simulation, problems from linear elasticity, and simulation of wave propagation. The instructions for compiling and running these examples are available on the wiki and from the README in the examples folder.

In this article, we include examples from linear elasticity and fluid mechanics. The source code for both examples is included in the article's repository: https://gitlab.com/e62Lab/2019_P_Medusa.

4.1 Linear Elasticity

Small displacements in an isotropic homogeneous linearly elastic material under stress are described by Cauchy-Navier equations

$$(\lambda + \mu)\nabla(\nabla \cdot \vec{u}) + \mu\nabla^2\vec{u} = \vec{f}, \tag{41}$$

where $\vec{u}$ are unknown displacements, $\vec{f}$ is the loading body force, and λ and μ are material constants, called Lamé parameters. The stress tensor σ is computed as

$$\sigma = \lambda \operatorname{tr}(\varepsilon) I + 2\mu\varepsilon, \quad \varepsilon = \frac{\nabla\vec{u} + (\nabla\vec{u})^{\mathrm{T}}}{2}, \tag{42}$$

where I is the identity tensor.

We consider a beam of dimensions $L \times W$ in 2D and $L \times W \times T$ in 3D, occupying the area $[0, L] \times [0, W]$ in 2D and $[0, L] \times [0, W] \times [0, T]$ in 3D. The beam is fixed on the side with the first coordinate equal to 0, experiences a downwards traction of size F on the side with the first coordinate equal to W and zero traction elsewhere. Note that this is not the classical Timoshenko beam, although the library was also validated against that problem [Slak and Kosec 2019b].

Additionally, some cavities (also with no traction boundary conditions) were added to the domain. The problems were solved for $L = 15$, $W = 5$, $T = 2$, with $E = 72.1 \cdot 10^9$, $\nu = 0.33$ and $F = 1,000$. Polyharmonic radial basis function on 25 nearest nodes with monomial augmentation od second order were used both in 2D and 3D. The results are shown in Figure 5, colored according to von Mises stress.

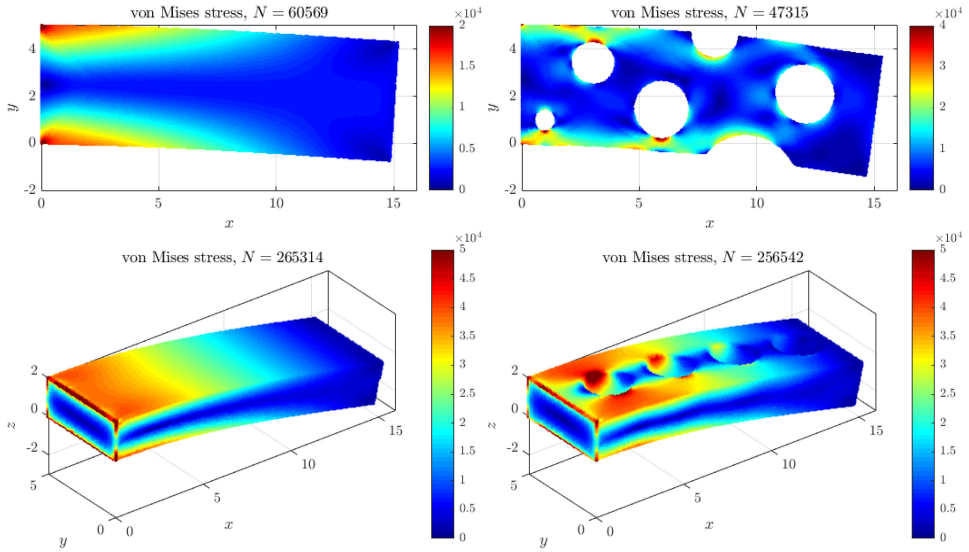

Fig. 5. Cantilever beams with and without cavities in 2D and 3D. Displacements are multiplied by a factor 10^5 in 2D and $5 \cdot 10^4$ in 3D.

4.2 Simulation of Natural Convection

The natural convection problem is governed by coupled Navier-Stokes, mass continuity, and heat transfer equations

$$\frac{\partial \boldsymbol{v}}{\partial t} + (\boldsymbol{v} \cdot \nabla)\boldsymbol{v} = -\frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2 \boldsymbol{v} + \frac{1}{\rho}\boldsymbol{b}, \tag{43}$$

$$\nabla \cdot \boldsymbol{v} = 0, \tag{44}$$

$$\boldsymbol{b} = \rho(1 - \beta(T - T_{\text{ref}}))\boldsymbol{g}, \tag{45}$$

$$\frac{\partial T}{\partial t} + \boldsymbol{v} \cdot \nabla T = \frac{\lambda}{\rho c_p}\nabla^2 T, \tag{46}$$

where $\boldsymbol{v}(u, v, w)$, p, T, μ, λ, c_p, ρ, $\boldsymbol{g}$, β, T_{ref}, and $\boldsymbol{b}$ stand for velocity, pressure, temperature, viscosity, thermal conductivity, specific heat, density, gravitational acceleration, coefficient of thermal expansion, reference temperature for Boussinesq approximation, and body force, respectively. The problem is defined on a unit square domain with vertical walls kept at constant different temperatures, while horizontal walls are adiabatic. In generalization to 3D, front and back walls are also assumed to be adiabatic [Wang et al. 2017]. The problem is solved with implicit timestepping and projection method for pressure-velocity coupling [Slak and Kosec 2019a]. We used PHS RBF-FD with augmentation of order 2 to solve the problem. Results in terms of velocity and temperature contour plots are presented in Figure 6 for Prandtl number 0.71 and Rayleigh numbers 10^8 in 2D and 10^6 in 3D, respectively. The obtained numerical values for the regular cases are compared with reference solutions in Table 1. More details about the solution procedure and results can be found in [Slak and Kosec 2019a] and the source code is available in this articles repository.

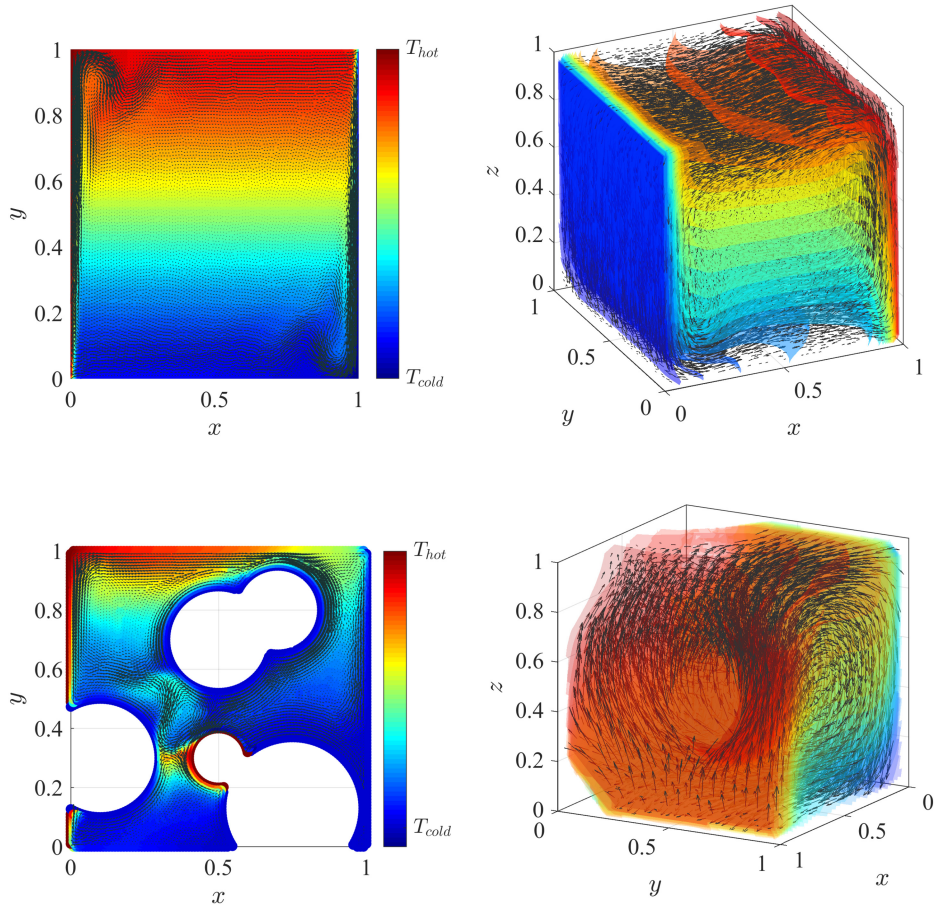

Fig. 6. Solution of natural convection problem for Ra=10^8 in 2D (top left), Ra=10^6 in 3D (top right), and on irregular 2D and 3D domains (bottom row).

Table 1. Comparison of Results, Obtained with Medusa, and Reference Data

	Ra	$v_{max}(x, 0.5)$			x			$u_{max}(0.5, y)$			y		
		present	(a)	(b)	present	(a)	(b)	present	(a)	(b)	present	(a)	(b)
2D	10^6	0.2628	0.2604	0.2627	0.037	0.038	0.039	0.0781	0.0765	0.0782	0.847	0.851	0.861
	10^7	0.2633	0.2580	0.2579	0.022	0.023	0.021	0.0588	0.0547	0.0561	0.870	0.888	0.900
	10^8	0.2557	0.2587	0.2487	0.010	0.011	0.009	0.0314	0.0379	0.0331	0.918	0.943	0.930
	Ra	$w_{max}(x, 0.5, 0.5)$			x			$u_{max}(0.5, 0.5, z)$			z		
		present	(c)	(d)	present	(c)	(d)	present	(c)	(d)	present	(c)	(d)
3D	10^4	0.2295	0.2218	0.2252	0.850	0.887	0.883	0.2135	0.1968	0.2013	0.168	0.179	0.183
	10^5	0.2545	0.2442	0.2471	0.940	0.931	0.935	0.1564	0.1426	0.1468	0.144	0.149	0.145
	10^6	0.2564	0.2556	0.2588	0.961	0.965	0.966	0.0841	0.0816	0.0841	0.143	0.140	0.144

The reference sources are (a): Couturier and Sadat [2000], (b): Kosec and Šarler [2008], (c): Wang et al. [2017], (d): Fusegi et al. [1991].

5 BENCHMARKS

While the design of Medusa is mainly focused on modularity and extensibility, we still take care that the implementation is reasonably efficient. To this end, we compare the performance of Medusa with the mature FreeFem++ library for solving PDEs. Note that we will be comparing two different methods for solving PDEs, which by themselves have different complexity, and it is not the purpose of this measurements to compare the methods, nor the quality of implementations. We simply wish to establish that Medusa execution times are in the same ballpark as the FreeFem++ ones for the same problem.

The comparison is done on the Poisson boundary value problem

$$-\nabla u = f \text{ in } \Omega,\ u = u_0 \text{ on } \partial\Omega, \tag{47}$$

for $u_0(x) = \prod_{i=1}^{d} \sin(\pi x_i)$ and $f = -\nabla^2 u_0$ on $\Omega = B(0,1) \setminus B(0,1/2)$ in 2D and 3D. Medusa implementation uses RBF-FD with PHS on $n = 9$ and $n = 35$ closest nodes in 2D and 3D, respectively. FreeFem++ implementation solves the corresponding variational formulation using P1 elements. The problem itself and the FreeFem++ code were taken from FreeFem++'s own example suite.

FreeFem++ and its dependencies were compiled from source, as was Medusa. Medusa is heavily templated to achieve extensibility and speed, and the price is paid with longer compile times. We took some steps to reduce the compile times through explicitly instantiating the main structures for most commonly used types, which reduced compile times significantly. After the initial compilation on the library, each subsequent compilation of the source code takes around 12 s for the solution of Equation (47).

Both implementations were run single-threaded on a laptop computer with `Intel(R) Core(TM) i7-7700HQ CPU @ 2.80 GHz` processor with 16 GB of DDR4 RAM. Each time measurement was repeated nine times and the median values are shown in Figure 7, with error bars showing standard deviation of the measurements.

Both methods attain expected convergence rate $N^{-2/d}$ and similar accuracy, with RBF-FD performing slightly worse. The difference in execution times is almost exclusively due to node placing in Medusa being faster than meshing in FreeFem++. The execution time is also highly dependent on the number of stencil nodes, which can be lowered or increased, and on the choice of sparse linear solver and its parameters. The Conjugate Gradient solver was chosen in FreeFem because it performed best, and BiCGStab with ILUT$(5, 10^{-2})$ preconditioner was chosen for Medusa. The solvers took approximately the same amount of time.

Parts of the Medusa solution procedure were also timed separately: namely, domain discretization, stencil selection, stencil weight computation, matrix assembly, preconditioner computation, iterative solution, and post-process error computation. Figure 8 shows these times with respect to the number of nodes and a ratio of time spent on each part of the solution procedure. These measurements also show the scaling behavior of different parts of the solution procedure. Computational time complexity of most parts is linear or log linear, with the exception of the linear solver. For most problems with explicit time iteration, the iteration itself is so time-consuming that domain discretization and weight computation are negligible, since they are only performed once at the beginning of the iteration.

Execution time ratio can vary significantly in different setups. For 2D problems with second-order methods, construction of domain discretization can take more than 50% of the total time. For high-order methods with large support sizes and augmentation orders, weight computation can severely dominate, even as high as 80%. For more complicated problems and larger N, linear solver can take up almost 90% of the time.

These separate time measurements also serve as a guideline for optimization and parallelization. Weight computation is trivially parallelizable and is already included in Medusa for shared memory

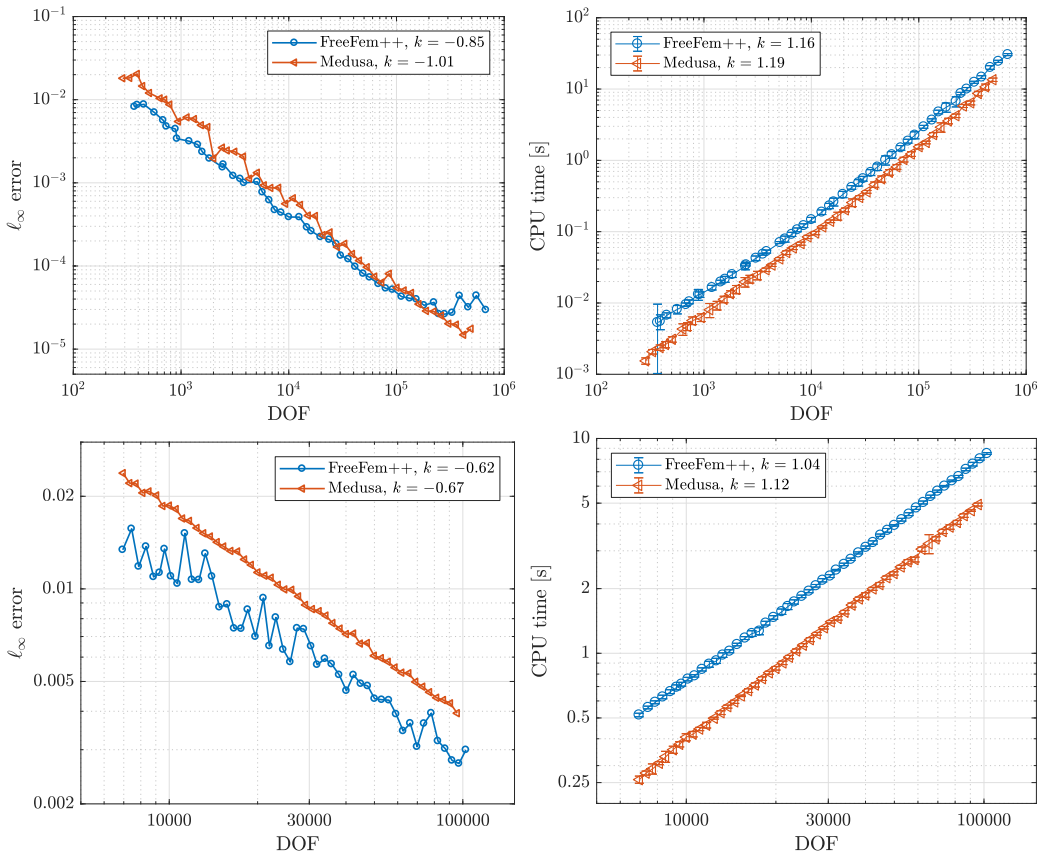

Fig. 7. Errors and execution times of FreeFem++ and Medusa when solving Equation (47) in 2D and 3D. Each time measurement was repeated nine times. The median value is shown with error bars representing the standard deviation.

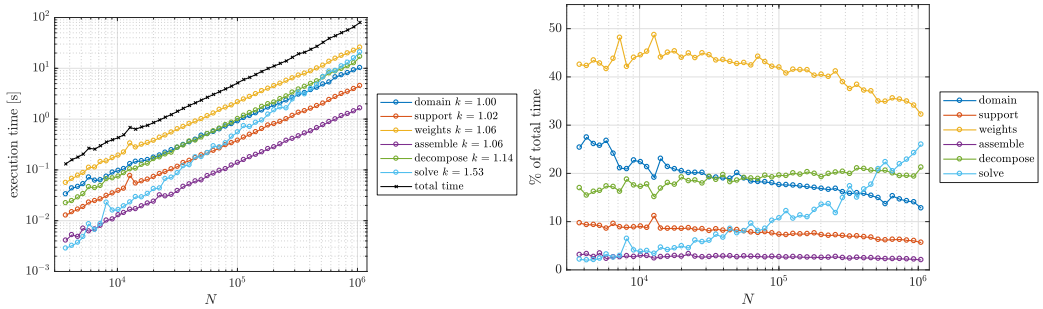

Fig. 8. Errors and execution times of FreeFem++ and Medusa when solving Equation (47) in 2D and 3D.

architectures, using OpenMP. Support for parallel sparse solvers is also included in Eigen, and other parallelization efforts are ongoing.

Additionally, we reviewed the cost of abstractions in performance critical sections by comparing execution time with a "bare-bones" implementation [Slak and Kosec 2018] and by analyzing

assembly instructions with Compiler Explorer [Godbolt et al. 2019] until we were satisfied with incurred costs, which are now small to negligible.

6 CONCLUSIONS AND OUTLOOK

In this article, we presented an overview of abstractions and implementation of Medusa, a general purpose C++ library for solving PDEs with strong-form methods. The library provides core elements of meshless solution procedures as standalone blocks that can be pieced together or swapped to ease research, development, and testing of meshless methods, all in a dimension-independent manner. It allows to define custom node generation and stencil selection procedures, basis functions, weights functions, RBFs, approximation schemes, and linear operators, relying heavily on C++ templating system and most commonly used classes are explicitly instantiated to avoid long compile times. We have demonstrated this modularity and extensibility by constructing several reported mesh-free methods, and many more examples are available in the documentation. Special attention is also paid to readability of the resulting code, which closely resembles the mathematical description of the problem and allows the user to think in terms of operators and fields instead of arrays and indices. The library is also tested for correctness with a suite of unit tests and offers technical documentation and other informal discussions on its website. A basic comparison of Medusa with FreeFem++ on a Poisson problem showed it is comparable in execution time for similar accuracy. The Medusa library repository is available at https://gitlab.com/e62Lab/medusa and this article's repository (https://gitlab.com/e62Lab/2019_P_Medusa), which contains the source code of all examples, along with the plotting scripts and the source of this manuscript.

Although Medusa is primarily intended as a research platform for mesh-free community, it offers enough features for solving 3D coupled problems, such as the illustrated thermo-fluid transport problem in an irregular 3D domain. Other problems, such as linear elasticity, complex-valued electromagnetic scattering, and wave propagation are also included in the examples.

The ongoing and future development of Medusa is aimed in several directions. One is to increase the geometric capabilities of Medusa by adding a module for discretization of parametric surfaces, and potentially extending it to handle Computer-aided Design objects, pushing Medusa a step closer to the engineering simulation software.

Another important direction is parallelism, since at the moment only naive shared memory parallelization of modules that are trivial to execute in parallel is offered. We are developing a parallel version of node positioning algorithms as well as a domain decomposition module required for distributed parallel execution.

Finally, binding for other languages, such as Python or Julia, is planned to be included, to increase the prototyping ability and accessibility of the library.

Throughout all other development, we will also (albeit conservatively) extend the set of approximations, bases, node generation algorithms, and other elements offered by default, with useful developments from ongoing research in core meshless areas. Potential future additions include better support for adaptivity and coupled problems.

ACKNOWLEDGMENTS

The authors would like to acknowledge other contributors to the Medusa library (and its previous unpublished versions), listed in alphabetical order: Urban Duh, Mitja Jančič, Maks Kolman, Jure Lapajne, Jure Močnik - Berljavac, Anja Petković, Anja Pirnat, Ivan Pribec, Tjaž Silovšek and Blaz Stojanovič.

REFERENCES

Karsten Ahnert and Mario Mulansky. 2011. Odeint–solving ordinary differential equations in C++. In *AIP Conference Proceedings*, Vol. 1389. American Institute of Physics, 1586–1589. DOI: https://doi.org/10.1063/1.3637934

W. Bangerth, R. Hartmann, and G. Kanschat. 2007. deal.II – A general purpose object oriented finite element library. *ACM Trans. Math. Softw.* 33, 4 (2007), 24/1–24/27. DOI: https://doi.org/10.1145/1268776.1268779

Victor Bayona, Natasha Flyer, Bengt Fornberg, and Gregory A. Barnett. 2017. On the role of polynomials in RBF-FD approximations: II. Numerical solution of elliptic PDEs. *J. Comput. Phys.* 332 (2017), 257–273. DOI: https://doi.org/10.1016/j.jcp.2016.12.008

V. Bayona, N. Flyer, G. M. Lucas, and A. J. G. Baumgaertner. 2015. A 3-D RBF-FD solver for modeling the atmospheric global electric circuit with topography (GEC-RBFFD v1. 0). *Geosci. Model Dev.* 8, 10 (2015), 3007. DOI: https://doi.org/10.5194/gmd-8-3007-2015

Victor Bayona, Miguel Moscoso, Manuel Carretero, and Manuel Kindelan. 2010. RBF-FD formulas and convergence properties. *J. Comput. Phys.* 229, 22 (2010), 8281–8295. DOI: https://doi.org/10.1016/j.jcp.2010.07.008

Jan Bender. 2016. SPlisHSPlasH. Retrieved from https://github.com/InteractiveComputerGraphics/SPlisHSPlasH.

W. Benz. 1990. Smooth particle hydrodynamics: A review. In *The Numerical Modelling of Nonlinear Stellar Pulsations*. Springer, 269–288. DOI: https://doi.org/10.1007/978-94-009-0519-1_16

Luca Bertagna, Simone Deparis, Luca Formaggia, Davide Forti, and Alessandro Veneziani. 2017. The LifeV library: Engineering mathematics beyond the proof of concept. *arXiv preprint arXiv:1710.06596* (2017).

Jose Luis Blanco and Pranjal Kumar Rai. 2014. nanoflann: A C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees. Retrieved from https://github.com/jlblancoc/nanoflann.

Evan Bollig. 2014. Radial Basis Function Finite Differences on the GPU. Retrieved from https://github.com/bollig/rbffd_gpu/.

Y. Choi and S. Kim. 1999. Node generation scheme for meshfree method by Voronoi diagram and weighted bubble packing. In *5th US National Congress on Computational Mechanics*.

H. Couturier and S. Sadat. 2000. Performance and accuracy of a meshless method for laminar natural convection. *Numer. Heat Transf.: Part B: Fundam.* 37, 4 (2000), 455–467. DOI: https://doi.org/10.1080/10407790050051146

Alejandro J. C. Crespo, José M. Domínguez, Benedict D. Rogers, Moncho Gómez-Gesteira, S. Longshaw, R. Canelas, Renato Vacondio, A. Barreiro, and O. García-Feal. 2015. DualSPHysics: Open-source parallel CFD solver based on Smoothed Particle Hydrodynamics (SPH). *Comput. Phys. Commun.* 187 (2015), 204–216. DOI: https://doi.org/10.1016/j.cpc.2014.10.004

Dassault Systèmes. 2012. Abaqus Unified FEA. Retrieved from https://www.3ds.com/products-services/simulia/products/abaqus/.

Christian Drumm, Sudarshan Tiwari, Jörg Kuhnert, and Hans-Jörg Bart. 2008. Finite pointset method for simulation of the liquid–liquid flow field in an extractor. *Comput. Chem. Eng.* 32, 12 (Dec. 2008), 2946–2957. DOI: https://doi.org/10.1016/j.compchemeng.2008.03.009

Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. ACM, 36–47. DOI: https://doi.org/10.1145/1966895.1966900

Bengt Fornberg and Natasha Flyer. 2015. Fast generation of 2-D node distributions for mesh-free PDE discretizations. *Comput. Math. Applic.* 69, 7 (2015), 531–544. DOI: https://doi.org/10.1016/j.camwa.2015.01.009

Chris Foster et al. 2011. tinyformat: Minimal, type safe printf replacement library for C++. Retrieved from http://rapidxml.sourceforge.net.

Fraunhofer Gesellschaft. 1999. MESHFREE. Retrieved from https://www.meshfree.eu/.

T. Fusegi, Jae Min Hyun, K. Kuwahara, and B. Farouk. 1991. A numerical study of three-dimensional natural convection in a differentially heated cubical enclosure. *Int. J. Heat Mass Transf.* 34, 6 (1991), 1543–1557. DOI: https://doi.org/10.1016/0017-9310(91)90295-p

L. Gavete, M. L. Gavete, and J. J. Benito. 2003. Improvements of generalized finite difference method and comparison with other meshless method. *Appl. Math. Model.* 27, 10 (2003), 831–847. DOI: https://doi.org/10.1016/S0307-904X(03)00091-X

Matt Godbolt, Rubén Rincón, Patrick Quist, Austin Morton, Jared Wyles, Chedy Najjar, Simon Brand, and Filipe Cabecinhas. 2019. Compiler explorer. Retrieved from https://github.com/mattgodbolt/compiler-explorer.

Gaël Guennebaud, Benoît Jacob et al. 2010. Eigen v3. Retrieved from http://eigen.tuxfamily.org.

Frédéric Hecht. 2012. New development in FreeFem++. *J. Numer. Math.* 20, 3-4 (2012), 251–266. DOI: https://doi.org/10.1515/jnum-2012-0013

Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. 2005. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.* 31, 3 (2005), 363–396. DOI: https://doi.org/10.1145/1089014.1089020

Trever Hines. 2015. RBF: Python package containing the tools necessary for radial basis function (RBF) applications. Retrieved from https://github.com/treverhines/RBF.

Yo-Ming Hsieh and Mao-Sen Pan. 2014. ESFM: An essential software framework for meshfree methods. *Adv. Eng. Softw.* 76 (2014), 133–147. DOI: https://doi.org/10.1016/j.advengsoft.2014.06.006

Inpartik & ITM University of Stuttgart. 2008. Pasimodo. Retrieved from https://www.itm.uni-stuttgart.de/en/software/pasimodo/.

Marcin Kalicinski. 2011. RapidXml. Retrieved from https://github.com/c42f/tinyformat.

Gregor Kosec. 2018. A local numerical solution of a fluid-flow problem on an irregular domain. *Adv. Eng. Softw.* 120 (2018), 36–44. DOI: https://doi.org/10.1016/j.advengsoft.2016.05.010

Gregor Kosec and Božidar Šarler. 2008. Solution of thermo-fluid problems by collocation with local pressure correction. *Int. J. Numer. Meth. Heat Fluid Flow* 18, 7/8 (2008), 868–882. DOI: https://doi.org/10.1108/09615530810898999

Gregor Kosec, Jure Slak, Matja Depolli, Roman Trobec, Kyvia Pereira, Satyendra Tomar, Thibault Jacquemin, Stéphane P. A. Bordas, and Magd Abdel Wahab. 2019. Weak and strong from meshless methods for linear elastic problem under fretting contact conditions. *Tribol. Internat.* (2019). DOI: https://doi.org/10.1016/j.triboint.2019.05.041

David Levin. 1998. The approximation power of moving least-squares. *Math. Comput.* 67, 224 (1998), 1517–1531. DOI: https://doi.org/10.1090/s0025-5718-98-00974-0

Hua Li and Shantanu S. Mulay. 2013. *Meshless Methods and Their Numerical Properties*. CRC Press.

Gui-Rong Liu. 2002. *Mesh Free Methods: Moving Beyond the Finite Element Method*. CRC Press. DOI: https://doi.org/10.1201/9781420040586

Ying Liu, Yufeng Nie, Weiwei Zhang, and Lei Wang. 2010. Node placement method by bubble simulation and its application. *Comput. Model. Eng. Sci.* 55, 1 (2010), 89.

Anders Logg and Garth N. Wells. 2010. DOLFIN: Automated finite element computing. *ACM Trans. Math. Softw.* 37, 2 (2010), 20. DOI: https://doi.org/10.1145/1731022.1731030

Rainald Löhner and Eugenio Oñate. 2004. A general advancing front technique for filling space with arbitrary objects. *Int. J. Numer. Meth. Eng.* 61, 12 (2004), 1977–1991. DOI: https://doi.org/10.1002/nme.1068

M. Maksić, V. Djurica, A. Souvent, J. Slak, M. Depolli, and G. Kosec. 2019. Cooling of overhead power lines due to the natural convection. *Int. J. Electric. Power Energy Syst.* 113 (Dec. 2019), 333–343. DOI: https://doi.org/10.1016/j.ijepes.2019.05.005

Ltd. MIDAS Information Technology Co.2018. midas MeshFree. Retrieved from http://www.midasmeshfree.com/.

Sławomir Milewski. 2013. Selected computational aspects of the meshless finite difference method. *Numer. Algor.* 63, 1 (2013), 107–126. DOI: https://doi.org/10.1007/s11075-012-9614-6

H. Munthe-Kaas and M. Haveraaen. 1996. Coordinate free numerics: Closing the gap between "pure" and 'applied" mathematics. *ZAMM Z. angew. Math. Mech* 76, S1 (1996), 487–488.

Nextflow Software. 2015. SPH-flow. Retrieved from https://www.nextflow-software.com/solvers/sphflow/.

V. P. Nguyen, T. Rabczuk, S. Bordas, and M. Duflot. 2008. Meshless methods: A review and computer implementation aspects. *Math. Comput. Simul* 79, 3 (2008), 763–813. DOI: https://doi.org/10.1016/j.matcom.2008.01.003

NOGRID GmbH. 2006. NOGRID. Retrieved from https://www.nogrid.com/.

Dang Thi Oanh, Oleg Davydov, and Hoang Xuan Phu. 2017. Adaptive RBF-FD method for elliptic problems with point singularities in 2D. *Appl. Math. Comput.* 313 (2017), 474–497. DOI: https://doi.org/10.1016/j.amc.2017.06.006

Eugenio Oñate, F. Perazzo, and J. Miquel. 2001. A finite point method for elasticity problems. *Comput. Struct.* 79, 22–25 (2001), 2151–2163. DOI: https://doi.org/10.1016/s0045-7949(01)00067-0

OneZero Software. 2008. Fluidix. Retrieved from https://www.fluidix.ca/.

Argyrios Petras, Leevan Ling, and Steven J. Ruuth. 2018. An RBF-FD closest point method for solving PDEs on surfaces. *J. Comput. Phys.* 370 (2018), 43–57. DOI: https://doi.org/10.1016/j.jcp.2018.05.022

Prabhu Ramachandran, Kunal Puri, Aditya Bhosale, A. Dinesh, Abhinav Muta, Pawan Negi, Rahul Govind, Suraj Sanka, Pankaj Pandey, Chandrashekhar Kaushik et al. 2019. PySPH: A Python-based framework for smoothed particle hydrodynamics. *arXiv preprint arXiv:1909.04504* (2019).

Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. 2016. Firedrake: Automating the finite element method by composing abstractions. *ACM Trans. Math. Softw.* 43, 3 (2016), 1–27. DOI: https://doi.org/10.1145/2998441

Yves Renard and Konstantinos Poulios. 2020. GetFEM: Automated FE modeling of multiphysics problems based on a generic weak form language. *ACM Trans. Math. Softw.* 47, 1 (2020).

Martin Robinson and Maria Bruna. 2017. Particle-based and meshless methods with Aboria. *SoftwareX* 6 (2017), 172–178. DOI: https://doi.org/10.1016/j.softx.2017.07.002

Matthieu Schaller, Pedro Gonnet, Peter W. Draper, Aidan B. G. Chalk, Richard G. Bower, James Willis, and Loïc Hausammann. 2018. SWIFT: SPH with inter-dependent fine-grained tasking. *Astrophysics Source Code Library* (2018), ascl–1805.

Varun Shankar and Aaron L. Fogelson. 2018. Hyperviscosity-based stabilization for radial basis function-finite difference (RBF-FD) discretizations of advection–diffusion equations. *J. Comput. Phys.* 372 (2018), 616–639. DOI: https://doi.org/10.1016/j.jcp.2018.06.036

Varun Shankar, Robert M. Kirby, and Aaron L. Fogelson. 2018. Robust node generation for meshfree discretizations on irregular domains and surfaces. *SIAM J. Sci. Comput.* 40, 4 (2018), 2584–2608. DOI: https://doi.org/10.1137/17m114090x

Jure Slak and Gregor Kosec. 2018. Parallel coordinate free implementation of local meshless method. In *MIPRO 2018: 41st International Convention on Information and Communication Technology, Electronics and Microelectronics, May 21–25, 2018, Opatija, Croatia* (2018-05-23) (MIPRO proceedings), Karolj Skala (Ed.). IEEE, Croatian Society for Information and Communication Technology, Electronics and Microelectronics, 194–200. DOI: https://doi.org/10.23919/mipro.2018.8400034

Jure Slak and Gregor Kosec. 2019a. On generation of node distributions for meshless PDE discretizations. *SIAM J. Sci. Comput.* 41, 5 (Oct. 2019), A3202–A3229. DOI: https://doi.org/10.1137/18M1231456

Jure Slak and Gregor Kosec. 2019b. Refined meshless local strong form solution of Cauchy–Navier equation on an irregular domain. *Eng. Anal. Bound. Elem.* 100 (Mar 2019), 3–13. DOI: https://doi.org/10.1016/j.enganabound.2018.01.001

Pratik Suchde and Jörg Kuhnert. 2019. A meshfree generalized finite difference method for surface PDEs. *Comput. Math. Applic.* 78, 8 (Oct. 2019), 2789–2805. DOI: https://doi.org/10.1016/j.camwa.2019.04.030

Sudarshan Tiwari and Jörg Kuhnert. 2003. Finite pointset method based on the projection method for simulations of the incompressible Navier-Stokes equations. In *Meshfree Methods for Partial Differential Equations*. Springer, 373–387. DOI: https://doi.org/10.1007/978-3-642-56103-0_26

A. I. Tolstykh and D. A. Shirobokov. 2003. On using radial basis functions in a "finite difference mode" with applications to elasticity problems. *Comput. Mechan.* 33, 1 (2003), 68–79. DOI: https://doi.org/10.1007/s00466-003-0501-9

Kiera van der Sande and Bengt Fornberg. 2019. Fast variable density 3-D node generation. *arXiv:1906.00636 [math.NA]* (2019).

Cheng-An Wang, Hamou Sadat, and Christian Prax. 2012. A new meshless approach for three dimensional fluid flow and related heat transfer problems. *Comput. Fluids* 69 (2012), 136–146. DOI: https://doi.org/10.1016/j.compfluid.2012.08.017

Peng Wang, Yonghao Zhang, and Zhaoli Guo. 2017. Numerical study of three-dimensional natural convection in a cubical cavity at high Rayleigh numbers. *Int. J. Heat Mass Transf.* 113 (2017), 217–228. DOI: https://doi.org/10.1016/j.ijheatmasstransfer.2017.05.057

Holger Wendland. 2004. *Scattered Data Approximation*. Vol. 17. Cambridge University Press.

Riccardo Zamolo and Enrico Nobile. 2018. Two algorithms for fast 2D node generation: Application to RBF meshless discretization of diffusion problems and image halftoning. *Comput. Math. Applic.* 75, 12 (June 2018), 4305–4321. DOI: https://doi.org/10.1016/j.camwa.2018.03.031

Received January 2019; revised August 2020; accepted September 2020

Author Query

Q1: AU: Please supply the CCS Concepts 2012 codes per the ACM style indicated on the ACM website. Please include the CCS Concepts XML coding as well.