Parallel Coordinate Free Implementation of Local Meshless Method

Jure Slak, Gregor Kosec
Jožef Stefan Insitute, Parallel and Distributed Systems Laboratory, Ljubljana, Slovenia
jure.slak@ijs.si, gregor.kosec@ijs.si

Abstract – This paper presents an implementation of a Meshless Local Strong Form Method that allows users to write elegant code expressed in terms of abstract mathematical objects, such as operators and fields, consequently avoiding working directly with matrix and array indices, which is tedious and error prone. This is achieved by using object oriented programming techniques for definition of abstract concepts and leveraging C++'s powerful templating mechanism. It is demonstrated that code written this way has little-to-no performance overhead compared to classical numerical code while being more expressive and readable, which gravely shortens model development and testing phases. The overall functionality of presented implementation is illustrated on numerical examples from classical thermodynamics, linear elasticity and fluid dynamics in one, two and three dimensions.

I. Introduction

The most cumbersome part of the implementation of numerical methods for solving partial differential equations (PDEs) is working with array and matrix indices for different differential operations, which is usually tedious, error prone, and hard to debug; enough so, that this aspect of programming numerical procedures has been studied before, especially in the context of choosing an appropriate programming language [1]. Traditional programming techniques require that the programmer keeps track of the meaning of all details in the code, which can be very hard to manage even for simple problems and becomes unmanageable for more complex systems of PDEs and constitutive relations. Another problem of such implementation is that it does not clearly reflect the problem at hand, making it hard to understand, change, and, maintain, when needed. Another problem is that in traditional implementation code changes with dimensionality of considered space, i.e. a 2D PDE solver is substantially different than a 3D solver. To wrestle this problem, a concept of coordinate free numerics was introduced by Haveraaen and Munthe-Kaas [2], postulating that software implementing numerical algorithms should abstract implementation details away from the user, exposing only an external interface that provides the necessary features. In such an implementation the user can think in terms of fields and differential operators instead of matrix indices and node enumerations. The difference in these approaches is illustrated in Figure 1.

The abstract approach is used extensively in almost all areas of mathematical programming, such as machine learning, network analysis and graphics, but implementations of numerical methods seem to lag behind the capabilities of modern programming languages.

$$\begin{tikzcd}
\text{Abstract PDE description} \arrow[r, "\text{fields, operators}"] \arrow[d, "\text{manual discretization}"'] \arrow[dr, dashed, bend left, "\text{coordinate free}"] \arrow[dr, dashed, bend right, "\text{conventional}"'] & \text{Abstract program} \arrow[d, "\text{automatic discretization (library)}"] \\
\text{Discrete description} \arrow[r, "\text{arrays, indexes}"'] & \text{Discrete program}
\end{tikzcd}$$

Figure 1. Comparison between coordinate free and conventional programming.

One of the often cited reasons for this lag is that abstractions make the code slower and cause unnecessary bloat. It has been proven that this is not necessarily the case when the library Eigen [3] for numerical linear algebra appeared in 2010, offering expressive object oriented syntax for matrix manipulation with speed comparable to LAPACK and BLAS.

In this paper we present an abstract implementation environment for solving systems of PDES with built-in parallelism. We use a Meshless Local Strong Form Method (MLSM) [4] as a numerical backbone, which seems to be an ideal numerical method for a coordinate free implementation due to its modular formulation. Similar approaches are already well established for the finite element method [5, 6] and particle-particle simulations [7], however no such implementations for meshless methods exist to the best of authors' knowledge.

The rest of the paper is organized as follows. In section II the numerical method along with a coordinate free parallel implementation concept is presented, in section III the numerical examples are demonstrated, and the conclusions are written in section IV.

II. Local Strong Form Meshless Method

Consider a boundary value problem of form

$$\mathcal{L}u = f, \text{ in } \Omega \tag{1}$$
$$\mathcal{R}u = g, \text{ on } \partial\Omega, \tag{2}$$

where $\Omega \subseteq \mathbb{R}^d$ is a domain u is an unknown scalar or vector field, $\mathcal{L}$ and $\mathcal{R}$ are linear partial differential operators and f and g are known functions. To obtain a discrete approximation of PDE, N points, also called nodes, are chosen in the domain Ω, of which N_i are in the interior and N_b on the boundary. Every point is also assigned a set of neighbors, called support domain.

For all N_i internal points p, the operator $\mathcal{L}$ at point p is approximated over support domain of p as

$$(\mathcal{L}u)(p) \approx \chi_{\mathcal{L}}(p) \tag{3}$$

where $\chi_{\mathcal{L}}$ is called a shape function for operator $\mathcal{L}$ at point p and only depends on the local geometry of the domain. For details on how to compute $\chi_{\mathcal{L}}$ the reader is referred to [4]. An important observation is that these shape functions can be computed beforehand and stored. A similar procedure is used to discretize any boundary conditions, in case they contain differential operators.

After computing the shape functions, two main approaches to obtain a numerical solution exist. Steady state problems will be solved implicitly, by approximating the equation

$$\mathcal{L}u = f \tag{4}$$

with a linear equation

$$\boldsymbol{\chi}_{\mathcal{L}}(p) \cdot \boldsymbol{u} = f(p), \tag{5}$$

where $\boldsymbol{u}$ is the vector of unknown function values in support domain of point p and $\cdot$ denotes the dot product. Adding equations for boundary conditions in boundary nodes, all equations can be gathered in a $N \times N$ sparse system

$$\begin{bmatrix} \boldsymbol{\chi}_{\mathcal{L}}(p_1) \\ \vdots \\ \boldsymbol{\chi}_{\mathcal{L}}(p_{N_i}) \\ \boldsymbol{\chi}_{\mathcal{R}}(p_{N_i+1}) \\ \vdots \\ \boldsymbol{\chi}_{\mathcal{R}}(p_{N_i+N_b}) \end{bmatrix} \begin{bmatrix} u_1 \\ \vdots \\ u_N \end{bmatrix} = \begin{bmatrix} f(p_1) \\ \vdots \\ f(p_N) \end{bmatrix}. \tag{6}$$

This system can be solved to obtain an approximation for function u at points p_i.

For transient problems, time derivative can be discretized using any standard procedure, such as explicit or implicit Euler method or Runge-Kutta methods, while spatial derivatives are again approximated using shape functions.

The presented formulation is convenient for coordinate free implementation, since in case of using RBFs for construction of shape functions all involved building blocks require only awareness of distances to surrounding nodes, which can be implemented at the lowest level. The reader is referred to [4] for more details.

A. *Coordinate free implementation*

The main purpose of coordinate free implementation of numerical methods is to abstract as many parts of the code as possible in order to present a flexible and readable interface between numerical operations and the user, e.g. to present concepts such as fields and differential operators. Naturally, the implementation of those concepts is still done in coordinates, however, it is done only once and only at the lowest level and it is not visible to the user, who is dealing with a complex physical model.

In our implementation vector and scalar fields are implemented as plain arrays using a well developed linear algebra library [3]. Domains are discretized with a list of points along with a list of neighbours for each point as described in section II. For boundary nodes, outside unit normals are stored, as well. The main concept that is usually problematic to abstract is the concept of a linear partial differential operator. However, equation (3) offers a way to compute and store different operators. Every linear partial differential operator $\mathcal{L}$ of order k at point p in an arbitrary number of dimensions can be written in standard basis using multiindex notation as

$$\mathcal{L}|_p = \sum_{|I|\le k} a_I(p) \left.\frac{\partial^{|I|}}{\partial x^I}\right|_p, \tag{7}$$

and approximated as

$$\mathcal{L}|_p \approx \sum_{|I|\le k} a_I(p) \boldsymbol{\chi}_{\frac{\partial^{|I|}}{\partial x^I}}\Big|_p. \tag{8}$$

This approach requires the computation of shape functions to be done only for derivatives of form $\frac{\partial^{|I|}}{\partial x^I}$ to approximate any linear operator. Examples in this paper use operators up to the second order and therefore one needs to compute and store shape functions for first and second derivatives over all combinations of coordinate directions for every point in the domain. This enables us to construct any operator of order two via (7), such as ∇^2 or $(\vec{n} \cdot \nabla)$. For example, Laplacian in two dimensions is computed as

$$\nabla^2|_p = \left.\frac{\partial^2}{\partial x^2}\right|_p + \left.\frac{\partial^2}{\partial y^2}\right|_p \approx \boldsymbol{\chi}_{\frac{\partial^2}{\partial x^2}}(p) + \boldsymbol{\chi}_{\frac{\partial^2}{\partial y^2}}(p). \tag{9}$$

The above expression can be easily generalized to a d dimensional space as

$$\nabla^2|_p = \sum_{i=1}^{d} \left.\frac{\partial^2}{\partial x_i^2}\right|_p \approx \sum_{i=1}^{d} \boldsymbol{\chi}_{\frac{\partial^2}{\partial x_i^2}}(p), \tag{10}$$

where x_i stands for i-th coordinate. The implementation of the shape functions computation is independent of domain dimensionality, making approximations of type (8) work seamlessly in all dimensions. The drawback of such approach to solving PDEs is the cost of preparing the shape functions, which has to be done every time nodal positions changes. On stationary domains, which will be considered in this paper, the shape functions need to be computed only once. Furthermore, this computational burden can be eased with parallel execution due to complete independence between different shape function computations.

Implementations of operators usually accept a matrix, a point index i and a scalar value α, and write shape coefficients, multiplied by α in the i-th matrix row, to build the system of equation described in (6). For explicit operators, they instead take a vector or scalar field and return the result directly. For example, user can use `op.lap`, representing the Laplacian operator, on a scalar field in 1, 2 or 3-dimensions with the same function name. The return value depends on the input parameters: if the input field is a 3-dimensional scalar field, the explicit `op.lap` will return a 3-dimensional scalar field, and an implicit `op.lap` will fill appropriate elements of the given matrix.

Implementations of described concepts are done using templated C++ classes, which are loosely coupled only by a mutual agreement to expose and expect the same interface. This is similar to duck typing in e.g. Python, but with the benefit of static type checking.

III. Numerical examples

A. Heat equation

First case stems from classical thermodynamics and asks for a steady state heat distribution in a homogeneous heated medium. It is governed by the equation

$$-\alpha \nabla^2 u = q, \tag{11}$$

where q is the volumetric heat source and α is thermal diffusivity. Dirichlet boundary conditions $u|_{\partial\Omega} = u_0$ and Neumann boundary conditions $\frac{\partial u}{\partial \vec{n}} = j$ will be considered.

First, a simple one dimensional case is tackled to assess basic properties of presented solution method. Consider the problem

$$\begin{aligned} u''(x) &= \sin(x), \quad x \in (0,1) \\ u(0) &= 0, u'(1) = 0 \end{aligned} \tag{12}$$

with analytical solution

$$u(x) = x \cos 1 - \sin x. \tag{13}$$

We solve the problem with MLSM using 3 support nodes and 3 monomials $\{1, x, x^2\}$ as basis functions on a regularly distributed nodes. This setup of MLSM is theoretically equivalent to the Finite Difference Method (FDM) and therefore it is worth implementing also standard FDM to compare results and execution times. The error between the actual solution u and the approximate solution $\hat{u}$ is measured in ℓ_∞ norm,

$$\|u - \hat{u}\|_{\ell_\infty} = \max_{x \in X} |u(x) - \hat{u}(x)|, \tag{14}$$

where X is the set of all points in the domain.

Coordinate free code for solving this problem is shown in listing 1, compared to classical code shown in listing 2. Both listings use variables `M` and `rhs` for the matrix and right hand side, respectively. The difference in readability is very pronounced. The user does not need to know almost anything about the method to understand the code in listing 1, while full working knowledge of FDM is required to understand code in listing 2.

```
for (int i : domain.internal()) {
    op.lap(M, i, 1.0);
    double x = domain.positions[i][0];
    rhs(i) = std::sin(x);
}
op.value(M, left, 1.0);
rhs(i) = 0;
op.neumann(M, right, {1}, 1.0);
rhs(i) = 0;
VectorXd solution = M.lu().solve(rhs);
```

Listing 1: Coordinate free code for solving problem (12) with MLSM.

The error analysis of MLSM and FDM methods with respect to number of nodes N is shown in Figure 2. Both methods converge regularly with order 2 as predicted by the theory. The error stops decreasing when the theoretical limit $\sqrt{\varepsilon}$ for second derivative approximation is reached, where ε represents the machine precision. Slightly lower final precision of MLSM is attributed to numerical errors during calculations of the shape functions.

```
vector<Triplet<scalar_t>> ts;
for (int i = 1; i < N-1; ++i) {
    ts.emplace_back(i, i, -2/h/h);
    ts.emplace_back(i, i-1, 1/h/h);
    ts.emplace_back(i, i+1, 1/h/h);
    rhs(i) = std::sin(i*h);
}
rhs(0) = 0;
ts.emplace_back(0, 0, 1);
ts.emplace_back(N-1, N-3, -1./2/h);
ts.emplace_back(N-1, N-2, 2/h);
ts.emplace_back(N-1, N-1, -3./2/h);
rhs(N-1) = 0;
M.setFromTriplets(ts.begin(), ts.end());
VectorXd solution = M.lu().solve(rhs);
```

Listing 2: Classical code for solving problem (12) with FDM.

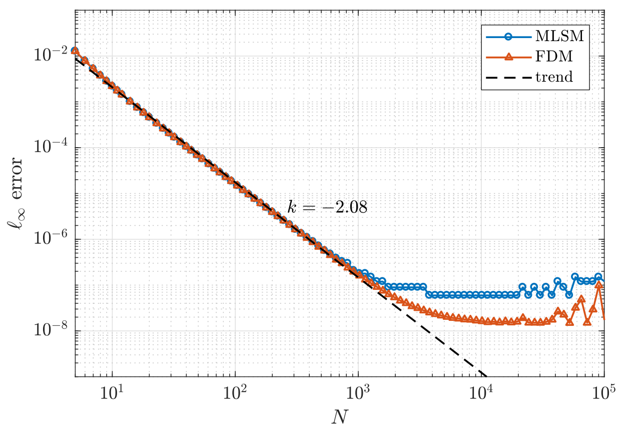

Figure 2. Convergence comparison of MLSM and FDM method.

Figure 3 represents the time spent by both methods.

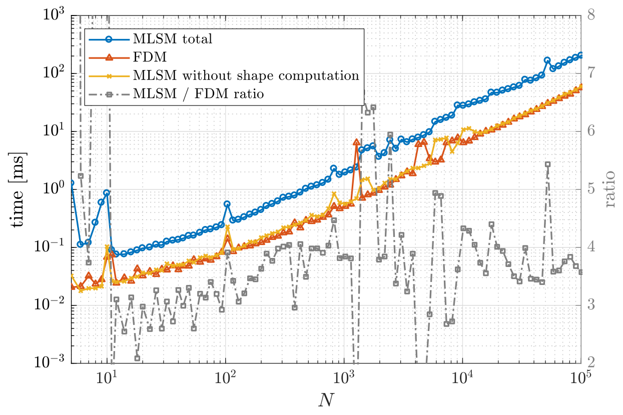

Figure 3. Comaprison of execution time of MLSM and FDM method.

As expected, MLSM is slower due to computation of shape functions, which are known in advance in FDM. However, counting that as part of the preprocessing and measuring only the part equivalent to FDM, it can be seen that the execution times for FDM and MLSM are the same for all practical purposes. This means that the abstractions discussed in section II.A and shown in listing 1 exhibit no measurable slowdown.

Next, a 2-dimensional case is considered with $\alpha = 1$ and $q = -2\pi^2 \sin(\pi x) \sin(\pi y)$ on a square domain $\Omega = [0,1] \times [0,1]$ with Dirichlet boundary conditions $u|_{\partial\Omega} = 0$. The analytical solution is given by

$$u(x,y) = \sin(\pi x) \sin(\pi y), \tag{15}$$

and error is again measured in ℓ_∞ norm given by (14). The problem was solved using MLSM in two setups: first, using monomial $\{1, x, y, x^2, y^2\}$ and 5 support nodes and second, using monomials $\{1, x, y, x^2, y^2, xy\}$ with 9 support nodes and Gaussian weight with $\sigma = 1$. These two setups are commonly used, because they mimic FDM and shall be referred to as MON5/5 and MON6/9, respectively. The code, describing the problem in shown as listing 3.

```
for (int i : domain.internal()) {
    op.lap(M, i, 1.0);
    double x = domain.positions[i][0];
    double y = domain.positions[i][1];
    rhs(i) = -2*pi*pi*std::sin(pi*x)
             *std::sin(pi*y);
}
for (int i : domain.boundary()) {
    op.value(M, i, 1.0);
    rhs(i) = 0;
}
VectorXd solution = M.lu().solve(rhs);
```

Listing 3: Code for solving two dimensional heat equation.

Plots of error of the method are shown in Figure 4. The order of convergence matches the theoretically predicted order 2.

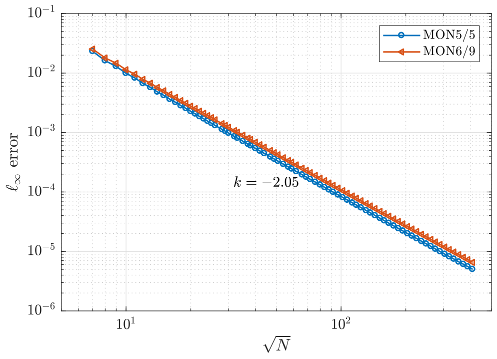

Figure 4. Error measured in ℓ_∞ norm of MLSM solving two dimensional heat eqation using MON5/5 and MON6/9 setups.

Finally, a 3-dimensional case is solved with $\alpha = 1$ and with heat source $q = -3\pi^2 \sin(\pi x)\sin(\pi y)\sin(\pi z)$. The analytical solution is

$$u(x, y) = \sin(\pi x)\sin(\pi y)\sin(\pi z). \tag{16}$$

The code needed to solve this problem is very similar to listing 3, with only right hand side being altered to reflect the new heat source. To solve the problem numerically, MLSM with basis $\{1, x, y, z, x^2, y^2, z^2\}$ and 7 support nodes and with basis $\{1, x, y, z, x^2, y^2, z^2, xy, xz, yz\}$ with 19 support nodes was used. These two setups are referred to as MON7/7 and MON10/19, respectively. The convergence in ℓ_∞ norm is shown in Figure 5. Once again the order of convergence matches the theoretical predictions.

The main benefit of using MLSM is, however, relatively simple consideration of complex geometries. Therefore, in a next example we present a more interesting 3-dimensional example, where a CAD model for aluminium heatsink (see Figure 6 left) is discretized to obtain the domain description.

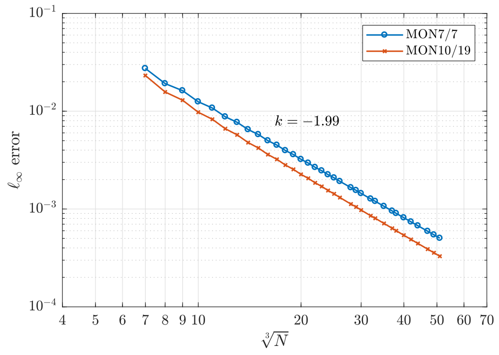

Figure 5. Error in ℓ_∞ norm of MLSM solving two dimensional heat eqation using MON7/7 and MON10/19 setups.

Heat equation (11) with $\alpha = 9.7 \cdot 10^{-5}\,\mathrm{m^2/s}$, with no heat generation, i.e. $q = 0\,\mathrm{W/m^3}$, with Dirichlet boundary conditions $u = 100\,°\mathrm{C}$ on the bottom surface and $u = 20\,°\mathrm{C}$ everywhere else.

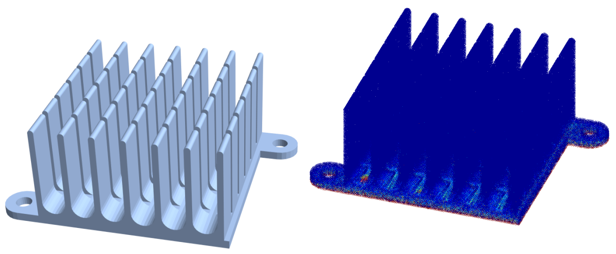

Figure 6. The CAD model for common aluminum heatsink (left), obtained from GrabCAD [8] and its steady heat distribution (right).

The solution is shown in Figure 6 (right), with a central cross section shown in Figure 7.

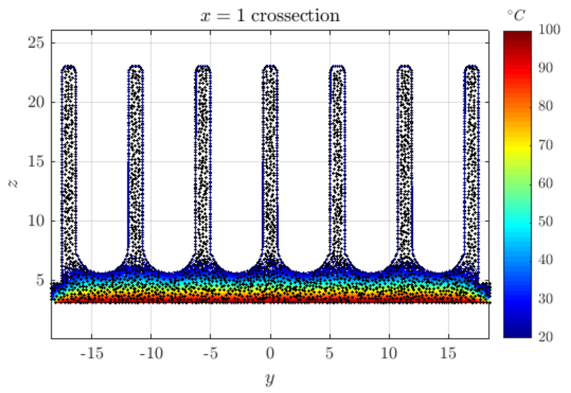

Figure 7. Contours of heat distribution in a crossection of considered heat sink.

B. Cantilever beam

The second problem arises from the theory of linear elasticity, dealing with deformations of solids under loading. The governing equation for elastic homogeneous isotropic materials is the Navier equation

$$(\lambda + \mu)\nabla(\nabla \cdot \vec{u}) + \mu\nabla^2\vec{u} = \vec{f} \tag{17}$$

describing displacements $\vec{u} = (u, v)$ and stresses σ, computed from $\vec{u}$ as

$$\sigma = \lambda(\operatorname{tr}\varepsilon)I + 2\mu\varepsilon, \quad \varepsilon = \frac{\nabla\vec{u} + (\nabla\vec{u})^\mathsf{T}}{2}. \tag{18}$$

Constants λ and μ are called Lamé parameters, I is the identity tensor and tr denotes the trace operator. Lamé parameters are usually expressed in terms of Young's modulus E and Poisson's ratio ν as

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \ \mu = \frac{E}{2(1+\nu)}. \tag{19}$$

A standard example from solid mechanics is the cantilever beam problem, where an ideal thin beam of length L and height D covering the area $[0, L] \times [-D/2, D/2]$ is bent with a force applied at one end while fixed at the other. Timoshenko beam theory offers a closed form solution for displacements and stresses in such a beam under plane stress conditions and a parabolic load on the left side. The solution is widely known and derived in e.g. [9], giving stresses in the beam as

$$\sigma_{xx} = \frac{Pxy}{I}, \ \sigma_{yy} = 0, \ \sigma_{xy} = \frac{P}{2I}\left(\frac{D^2}{4} - y^2\right), \tag{20}$$

and displacements as

$$\begin{aligned} u &= \frac{Py\big(3D^2(\nu+1)-4\big(3L^2+(\nu+2)y^2-3x^2\big)\big)}{24EI}, \\ v &= -\frac{P\big(3D^2(\nu+1)(L-x)+4(L-x)^2(2L+x)+12\nu xy^2\big)}{24EI}, \end{aligned} \tag{21}$$

where $I = \frac{1}{12}D^3$ is the moment of inertia around the horizontal axis, E is Young's modulus, ν is the Poisson's ratio and P is the total load force.

The problem is solved numerically using MLSM with traction boundary conditions $\sigma\vec{n} = \vec{t_0}$ given by (20) prescribed on the top, left and bottom boundary, while displacements $\vec{u} = \vec{u}_0$ given by (21) are prescribed on the right boundary. Code for solution of this problem is shown as listing 4.

```
for (int i : domain.internal()) {
    op.graddiv(M, i, lam + mu);
    op.lapvec(M, i, mu);
    rhs(i) = 0; }
for (int i : domain.bottom()) {
    op.traction(M, i, lam, mu, {0, -1});
    rhs(i) = 0; }
for (int i : domain.top()) {
    op.traction(M, i, lam, mu, {0, 1});
    rhs(i) = 0; }
for (int i : domain.left()) {
    double y = domain.positions[i][1];
    op.traction(M, i, lam, mu, {-1, 0});
    rhs(i) = Vec2d(0,
                -P*(D*D - 4*y*y)/(8*I)); }
for (int i : domain.right()) {
    double y = domain.positions[i][1];
    op.valuevec(M, i, 1.0);
    rhs(i) = Vec2d((P*y*(3*D*D*(1+v)-
                  4*(2+v)*y*y))/ 24.*E*I),
                -(L*v*P*y*y) / (2.*E*I)); }
VectorXd solution = M.lu().solve(rhs);
```

Listing 4: Code for solving the cantilever beam problem.

Errors of displacements and stresses are measured using analogues of the ℓ_∞ norms

$$e(\vec{u}) = \frac{\max_{x\in X}\{\max\{|u(x)-\hat{u}(x)|,|v(x)-\hat{v}(x)|\}\}}{\max_{x\in X}\{\max\{|u(x)|,|v(x)|\}\}}, \tag{22}$$

$$e(\sigma) = \frac{\max_{x\in X}\{\max|\sigma(x)-\hat{\sigma}(x)|}{\max_{x\in X}\{\max|\sigma(x)|\}}, \tag{23}$$

where $\max|\sigma(x)|$ represents the largest element in σ by absolute value. The convergence of MLSM in MON9/9 and MON9/13 setups is shown in Figure 8.

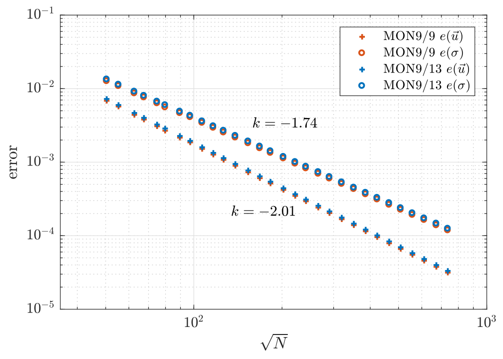

Figure 8. Error of MLSM solving cantilever beam problem using MON9/9 and MON9/13 setups.

To further demonstrate the generality of the method, a domain with arbitrarily positioned holes is considered, subjected to the same boundary conditions as a cantilever beam with traction free conditions on the inside of the holes. A comparison of von Mises stresses (a commonly used yield criterion) in ordinary cantilever beam and a cantilever beam with holes is presented in Figure 9.

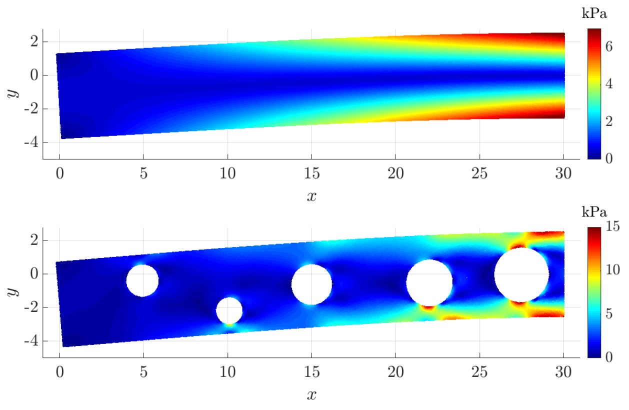

Figure 9. Comparison between solutions of ordinary and drilled cantilever beam problems, colored by von Mises stress.

C. *Lid-driven cavity*

The last example comes from the Computational fluid dynamics (CFD), where the core of the problem lies in solving the Navier-Stokes equations or its variants, e.g. Darcy or Brinkman equation for flow in porous media. The Navier-Stokes equations for incompressible flow are

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u}\cdot\nabla)\vec{u} = -\nabla p + \frac{1}{\mathrm{Re}}\nabla^2\vec{u} \tag{24}$$

$$\nabla\cdot\vec{u} = 0 \tag{25}$$

where $\vec{u} = (u, v)$ is the flow velocity, Re is the Reynolds number and p is the pressure. A standard test for assessment of numerical methods attempting to solve the Navier-Stokes equations is the *lid driven cavity* problem. It has been proposed in 1982 [10] and since then solved by many researchers with wide spectra of different numerical methods. The test is still widely studied and used for validation of novel methods and numerical principles. In lid driven cavity test a 2-dimensional domain is considered, where all boundaries except for the top boundary are of no slip type, while the top boundary is set to the constant velocity. More details on the case can be found in many papers, recently also in meshless context in [11]. There are different algorithms for addressing the pressure-velocity coupling. Here, we will use a pressure correction iteration [12], where a Poisson pressure correction equation

$$\nabla^2 p^{\text{corr}} = \frac{\rho}{\Delta t} \nabla \cdot \vec{u}^{\text{iter}} \tag{26}$$

with normal boundary conditions

$$\frac{\Delta t}{\rho} \frac{\partial p^{\text{corr}}}{\partial \vec{n}} = 0 \tag{27}$$

and regularization that ensures a unique solution

$$\int_\Omega p d\Omega = 0 \tag{28}$$

is used to project velocity towards solenoidal field to ensure mass continuity.

The intermediate velocity is, with present MLSM coordinate free implementation, computed as

```
for (int i : domain.internal()) {
    op.valuevec(M, i, 1 / O.dt);
    op.lapvec(M, i, -O.mu / O.rho);
    op.gradvec(M, i, v1[i]);
    rhs[i] = -1 / O.rho * op.grad(p, i)
             + v1[i] / O.dt;
}
for (int i : domain.boundary()) {
    op.valuevec(M, i, 1);
}
rhs[domain.boundary()]= Vec2d(0, 0);
rhs[domain.top()] = Vec2d(1, 0);
v_iter = solver.solveWithGuess(rhs, v_1);
```

In a next step a pressure correction Poisson equation is solved

```
for (int i : domain.internal()) {
    op.lap(M, i, 1.0);
    rhs_p(i) = O.rho/O.dt*op.div(v_iter, c);
}
for (int i : domain.boundary()) {
    op.neumann(M, i, domain.normal(i), 1.0);
    rhs_p(i) = 0;
}
for (int i = 0; i < N; ++i) {
    M.coeffRef(N, i) = 1;  // Regularization.
    M.coeffRef(i, N) = 1;
    rhs_p(N) = 0
}
P_c = M.lu().solve(rhs_p);
```

Finally, the velocity is corrected to fulfill the divergence free criterion.

```
for (int i : domain.internal()) {
    v[i] -= O.dl*O.dt/O.rho*op.grad(p_c,);
}
```

The example velocity magnitude contour plot of Re = 3200 case is presented in Figure 10. The comparison against reference solution [13, 10] along with convergence information is presented in Figure 11. MLSM solution converges regularly to the same values as the reference solutions for all cases

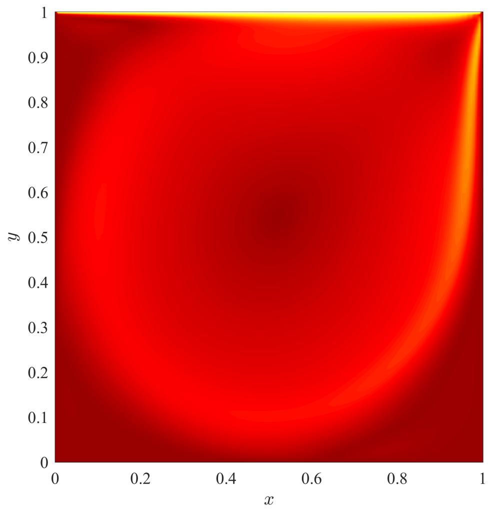

Figure 10. Velocity magnitude contour plot for Re $=$ 3200 case.

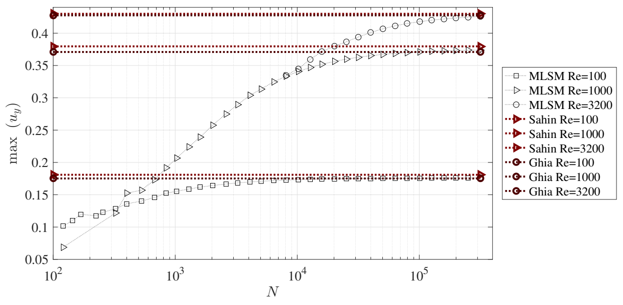

Figure 11. Comparison of present MLSM solution against reference data.

IV. Conclusions

In this paper we presented a concept of Coordinate Free software implementation of numerical solver based on the Meshless Local Strong Form Method. It is demonstrated that after the implementation of mathematical concepts, such as domains, scalar and vector fields, and differential operators, one can numerically solve a wide variety of PDEs with elegant and readable code. It is also demonstrated that coordinate free interfaces bring no observable overhead in execution time.

Acknowledgments

The authors would like to acknowledge the financial support of the Research Foundation Flanders (FWO), The Luxembourg National Research Fund (FNR) and Slovenian Research Agency (ARRS) in the framework of the FWO Lead Agency project: G018916N Multi-analysis of fretting fatigue using physical and virtual experiments and the ARRS research core funding No. P2-0095.

References

[1] Sylwester Arabas, Dorota Jarecka, Anna Jaruga, and Maciej Fijałkowski. Formula translation in Blitz++, NumPy and modern Fortran: A case study of the language choice tradeoffs. *Scientific Programming*, 22(3):201–222, 2014.

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

[3] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3, 2010. URL: `http://eigen.tuxfamily.org`, accessed 2018-01-12.

[4] J Slak and G Kosec. Refined Meshless Local Strong Form solution of Cauchy–Navier equation on an irregular domain. *Engineering Analysis with Boundary Elements*, 2018.

[5] Frédéric Hecht. New development in freefem++. *Journal of numerical mathematics*, 20(3-4):251–266, 2012.

[6] Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*, volume 84. Springer Science & Business Media, 2012.

[7] Roman Trobec, Marjan Šterk, Matej Praprotnik, and D Janežič. Implementation and evaluation of mpi-based parallel md program. *International Journal of Quantum Chemistry*, 84(1):23–31, 2001.

[8] Common Aluminum Heatsink, GrabCAD. URL: `https://grabcad.com/library/tiger-heatsink-mesh-1`, accessed 2018-02-13.

[9] William S Slaughter. *The linearized theory of elasticity*. Springer Science & Business Media, 2012.

[10] U. Ghia, K. Ghia, and C. Shin. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computaional Physics*, 48:387–411, 1982.

[11] G. Kosec. A local numerical solution of a fluid-flow problem on an irregular domain. *Advances in Engineering Software*, 2016.

[12] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer, Berlin, 2002.

[13] M. Sahin and R. Owens. A novel fully implicit finite volume method applied to the lid-driven cavity problem. Part I: High Reynolds number flow calculations. *International Journal for Numerical Methods in Fluids*, 42:57–77, 2003.