2018

A Parallel Mesh Generator in 3D/4D

Kirill Voronin
Portland State University, kvoronin@pdx.edu

A parallel mesh generator in 3D/4D*

Kirill Voronin

Portland State University
kirill.v.voronin@gmail.com

Abstract

In the report a parallel mesh generator in 3d/4d is presented. The mesh generator was developed as a part of the research project on space-time discretizations for partial differential equations in the least-squares setting. The generator is capable of constructing meshes for space-time cylinders built on an arbitrary 3d space mesh in parallel. The parallel implementation was created in the form of an extension of the finite element software MFEM. The code is publicly available in the Github repository [1].

Introduction

Finite element method is nowadays a common approach for solving various application problems in physics which is known first of all for its easy-to-implement methodology. Along with the other closely related approaches, e.g., finite volume methods or discontinuous Galerkin methods finite element method has proved its efficiency by handling complicated problems in numerous applications. Application problems usually require detailed meshes with large number of nodes in order to resolve all geometrical or physical features of the considered domain.

Traditionally, these methods are used for discretization in space whereas different approaches, e.g., finite differences or Runge-Kutta methods, are used for time discretization. However, in many cases separation of time and space can introduce additional difficulties to the problem, e.g., for problems in time-dependent domains or when conditionally stable time discretization methods are used.

*This work was supported by the ARO under US Army Federal Grant # W911NF-15-1-0590 and # W911NF-16-1-0307. Numerical studies were partially facilitated by equipment acquired under NSFs Major Research Instrumentation grant DMS-1624776.

One of the alternative approaches which has gained much attention and developed greatly recently are space-time methods which provide a simultaneous approximation in both time and space. Apparently, one of the main disadvantages of space-time methods is the increased size of the problem to be solved. However, when combined with appropriate upscaling techniques, these methods can be quite efficient and at the same time avoid some of the drawbacks which are typical for traditional discretization methods.

Certainly, for space-time methods appropriate space-time meshes are to be constructed. While a number of mesh generating software exist for two- and three-dimensional cases, almost none are known to the authors which consider four-dimensional case (three-dimensional space + time). Since space-time approaches introduce an additional dimension to the mesh, it is also important to be able to construct the space-time mesh in parallel. First, this will reduce memory requirements since the space-time mesh as a whole might not fit into the memory of a single computational node. Second, it will reduce the total time for mesh generation.

This report is focusing on the parallel mesh generating algorithm which constructs $(d+1)$-dimensional space-time meshes as tensor-product extensions of given d-dimensional (triangular or tetrahedral) space meshes in parallel, $d = 2, 3$. In section 1 the mesh generating algorithm is described in detail, first the serial version, in subsection 1.1, and then the parallel version, in subsection 1.2. Another important topic for space-time setting, which is visualization, is addressed in section 2. The developed mesh generator is supplied with additional mesh- and grid function-slicing functionality which produces VTK output sutable for bisualization e.g., in ParaView. Finally, several numerical experiments in which we study the space-time approximation properties of the constructed space-time meshes as well as the performance of the meshing algorithm are presented in section 3. A more elaborate description of the overall approach (space-time CFOSLS) can be found in the paper [2], or in an older paper [6].

1 Mesh construction

The developed $(d + 1)$-dimensional $(d = 2, 3)$ parallel mesh generator is dedicated for constructing meshes of tetrahedrons (or pentatops) for space-time cylinders given arbitrary triangular (or tetrahedral) meshes (called *base meshes* below) in the cylinder base. The input parameters for the algorithm are: a base mesh, number of time slabs and a time step. Constructed space-time meshes are essentially tensor-product extensions of the base mesh into the higher-dimensional space. Although the generated space-time meshes have simple structure w.r.t to time, they can be modified via adaptive mesh refinement to adapt better and resolve the subdomain of interest in practical computations.

1.1 Serial algorithm

Here main steps of serial algorithm are discussed. Although mesh storage formats used by mesh generating software can vary across the meshing software, our implementation is based on MFEM [9] framework and therefore we use MFEM mesh format for unstructured meshes as a model. It includes the following key data structures for a serial mesh:

1. elements (given by vertex indices)
2. boundary elements (given by vertex indices)
3. vertices (given by their coordinates)

Having these data available for the base mesh the algorithm creates corresponding data structures for the space-time mesh.

1.1.1 Generating vertices

In general, one might consider introducing additional nodes inside the time slabs [3] in order to produce a mesh which approximates better certain features of the space-time domain. However, in the implemented algorithm we take the viewpoint that such mesh improvements should be incorporated into the refinement strategies which are outside of the scope of the mesh generation. Thereby, since no additional vertices are added inbetween the time sections, generation of the space-time vertices is straightforward. Space-time vertices are nothing else but $(d+1)$-dimensional duplicates of base mesh vertices with varying time coordinate.

1.1.2 Generating elements

The implemented algorithm for generating $(d+1)$-dimensional mesh elements follows the guidelines from [3] and [5]. Elements are generated in an element-by-element fashion by the following local procedure applied to each of the base mesh elements:
Local procedure to generate elements:

In a loop over all time slabs:

1. For a given time slab form a $(d+1)$-dimensional prism with a given base mesh element in the base,
2. Decompose the prism into d elements (tetrahedrons or pentatops), see Fig.1.

Thus, each prism will be decomposed into $(d+1)$-simplices which are the new space-time elements.

For the last step two options were considered and implemented.

Figure 1 – A space-time prism in three-dimensional case (left) and its tesselation into tetrahedrons (right).

First approach. The more general approach [3] is to construct a Delaunay triangulation. For creating a Delaunay triangulation one of the standard geometrical packages, e.g., Qhull [10] can be used. However, one should remember that for a prism with regular faces (in our case the lateral faces are regular) Delaunay tesselation is not unique. This might lead to nonconforming meshes due to the incompatible tesselation of the common faces for neighbouring prisms. In order to circumvent this, one can joggle (perturb) time coordinates for the vertices in the prism upper base (see Fig.2). In *serial* algorithm the joggling can be random or based on any other information which is global for neighbouring elements, e.g. global vertex indices. For parallel implementation using information which is global for neighbouring elements is not enough since then this idea may lead to additional data transfers and synchronization points. Another thing to keep in mind is that due to the round-off errors one might get sliver simplices which should be eliminated afterwards.

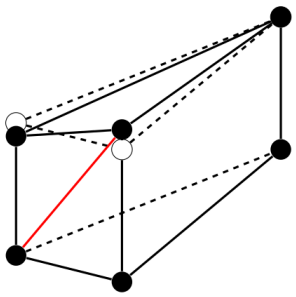

Figure 2 – A space-time prism with perturbed vertices in three-dimensional case. Now the Delaunay tesselation is unique.

Finally, the Delaunay based approach can be written as the following local procedure:
Decomposing $(d+1)$*-dimensional prism: Delaunay triangulation*

1. Perturb vertex coordinates so that Delaunay triangulation of the prism is unique;
2. Create a Delaunay triangulation of the prism (using qhull [10]);

3. Eliminate sliver (almost degenerate) $(d+1)$-simplices.

The elimination can be done by checking the relative simplex volume. The regular volume should be of order $h^3\tau$ where h is the base mesh step and τ is the time step.

Second approach. Another approach is to use a standard tesselation [5] of a $(d+1)$-dimensional prism into $(d+1)$-dimensional simplices.
Decomposing $(d+1)$-dimensional prism: Standard simplex tesselation

1. If prism vertices in the lower and upper bases are denoted as $A_1, ..., A_k$ and $A'_1, ..., A'_k$ correspondingly (see Fig.3), then one can take the following simplices for the prism decomposition: $\{A_1, ..., A_k, A'_1\}$, $\{A_2, ..., A_k, A'_1, A'_2\}$, ..., $\{A_k, A'_1, ..., A'_k\}$.

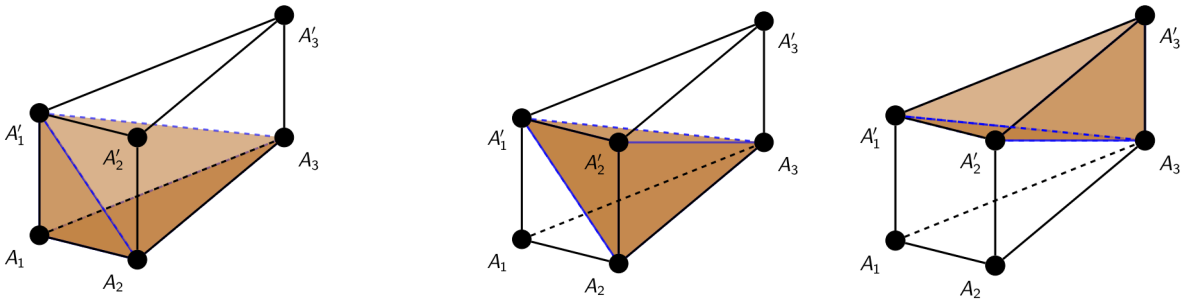

Figure 3 – Decomposition of a three-dimensional space-time prism into three tetrahedrons based on the vertex numbering.

In order to have compatible tesselations across neighbouring elements, the authors of [5] suggest to reorder the prism vertices so that the global ordering of vertex indices is preserved. This idea will be slightly modified for the parallel setup.

1.1.3 Generating boundary elements

It is easy to notice that the boundary of the space-time mesh consists of the following parts:

1. lower cylinder base,
2. upper cylinder base,
3. element faces whose projections onto the base mesh along the time direction are boundary elements of the base mesh.

The first and second types of boundary elements can be added to the structure for space-time boundary elements at any moment. The key question is then how to handle the third type, i.e., how to determine whether any faces of a given base mesh element belong to the boundary of the base mesh. Again, here several options are possible.

In case when no face-to-boundary connection table (or any equivalent) is available for the base mesh, one can create a list of all base mesh boundary elements and look up in it for each base face which leads to $O(n_f \cdot \log n_{be})$ operations where n_f is the number of faces and n_{be} is the number of boundary elements in the base mesh.

Usually, however, face-to-boundary connection table is known so one can simply use it for checking if the given base element's face belongs to the boundary.

1.2 Parallel algorithm

As an input, the parallel algorithm takes a base mesh which is already distributed among the processes. In this case the distributed mesh parts have their own local-to-process numbering as well as data structures for shared entities: faces, planars (in 4D), edges, and vertices.

The parallel implementation consists of two main steps which are:

1. generating local parts of the space-time mesh on the processes,
2. setting up communication structures governing the shared entities between processes.

Certainly, local parts are generated via the serial algorithm on each process, as described in subsection 1.1.

As for the shared entities, one should notice that as soon as shared entities are known for the base mesh, the shared entities can be also defined for the space-time elements by considering projections of space-time entities onto the base. For example, shared planars in 4D (which are essentially triangles) can be defined as those planars whose projection on the base is either a shared edge or a shared face (triangle) of the tetrahedral base mesh.

It is quite easy to compute the total number of shared planars for each process. Consider a lateral face of the 4D space-time prism (which itself is a three-dimensional prism in 4D) decomposed into three tetrahedrons, see Fig.4. There are 2 bases and 2 inner triangles which are projected onto the same base triangle. In addition, for each lateral face of the considered 3D prism (which is a rectangle in 4D), there are 2 triangles which are projected onto the same base edge.

Similarly, knowing shared entities for the base mesh one can define the number of all shared space-time entities via projections:

$$n_{shared\,faces,d+1} = d \cdot n_{steps} \cdot n_{shared\,faces,d}$$

$$n_{shared\,edges,d+1} = (2 \cdot n_{steps} + 1) \cdot n_{shared\,edges,d} + n_{steps} \cdot n_{shared\,vertices,d}$$

$$n_{shared\,vertices,d+1} = (n_{steps} + 1) \cdot n_{shared\,vertices,d}$$

$$n_{shared\,planars,4} = (2 \cdot n_{steps} + (n_{steps} + 1)) \cdot n_{shared\,faces,3} + 2 \cdot n_{steps} \cdot n_{shared\,edges,3}$$

Figure 4 – Two types of shared planars in the lateral face which can be considered as a three-dimensional prism. Projections of the planars onto the base plane are eigher base mesh elements (left) or base mesh edges (right).

Here n_{steps} is the number of time slabs in the space-time cylinder, subscripts d and $d+1$ correspond to the space-time and base meshes.

For example, generation of shared planars can be done by performing the following:
Creating shared planars for the space-time mesh:

In a loop over space-time planars

1. Project the given planar onto the base.
2. Look in the set of shared edges and shared faces of the base mesh for the projection. If found, add a shared planar.

The look-up in the set of base mesh shared entities (step 3 above) implies $O(\, n_{entity,d+1} \cdot \log n_{sharedentities,d})$ operations. The reason for loops over space-time entities is that if standard routines are used for generating faces, edges and planars, one does not know in advance how they are ordered. As a possible improvement, one could reimplement these generating routines so as to know which space-time entities will be shared.

Special care should be taken so that for each group of the processes which share any mesh entities, the order in which shared entities appear in the lists is the same for all processes in the group (requirement by MFEM). Another important requirement is to ensure compatible orientation of shared entities across the neighbouring processes and at the same time preserve the local-to-entity indices ordering which can be required by refinement routines.

In comparison to serial case, parallel setup introduces one additional complication related to the mesh compatibility across neighbouring processes. Usually, mesh parts are handled by different processes and have their own local-to-process vertex numbering. Therefore, one needs to ensure compatibility of the mesh across neighbouring mesh parts which is no longer preserved if one uses local-to-process vertex numbering.

Here one can notice that in general it is not required to have exactly the same vertex perturbation for common vertices across neighbouring elements and processes in Delaunay triangulation. The same is true for the vertex ordering if one uses simple prism decom-

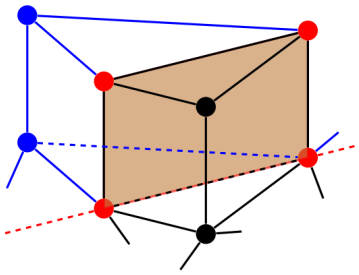

Figure 5 – A shared face between two neighboring space-time prisms in 3D is essentially a rectangle (with red vertices). The same diagonal should be chosen in the rectangle to ensure mesh compatibility.

position into simplices. However, in any case one still needs to generate a compatible triangulation.

The simple solution is to use local-to-element "geometric" ordering of vertices. That is, one can just reorder the vertices so that the following "lexicographical" order is preserved: vertex A is said to be "larger" than vertex B if x-coordinate of A is larger of x-coordinate of B, or their x-coordinates are equal and the next coordinate of A is larger than the next one of B, etc. For example, vertex $(0.0; 1.0; 2.0)$ is larger than $(0.0; 1.0; 0.0)$, but less than $(0.0; 2.0; 1.0)$.

Thus, vertex joggling can be defined to be proportional to the vertex index in this ordering or this ordering can be directly used in simplex decomposition described above. The same trick can be used whenever one needs to orient a space-time entity. Namely, the orientation can be said to be "positive" if its vertex ordering is an even permutation of the "geometric" ordering defined above.

Remark. The presented algorithm is completely parallel since the local procedures depend only on the local-to-process data. The only communication and synchronization point is required in the very end while finalizing the communication structures which are, for example, communicators for groups of processes which share some entities.

2 Visualization

For space-time meshes in 4D it is also important to visualize the obtained results. A common approach is to slice the space-time domain by a set of hyperplanes defined by $t = t_k$ for a series of time moments t_k. The intersection of the $(d+1)$-dimensional space-time mesh and a hyperplane is a d-dimensional mesh (called *slice mesh* below). For tetrahedral and pentatop meshes the slice mesh consists of two types of cells: triangles/quadrilaterals in 3D and tetrahedrons/wedges in 4D (see Fig.6).

In [4] an approach based on cutting edges of the mesh elements was presented. The idea

Figure 6 – Two types of slice mesh cells in 4D: tetrahedrons (left) and wedges(right).

is the following:
Local procedure to compute the slice mesh cells:

1. Check whether a considered space-time element is intersected by a given hyperplane.
2. If it is intersected, two cell types are possible: a tetrahedron or a wedge in 4D, or a triangle or a quadrilateral in 3D.
3. The vertices of the slice mesh cell can be computed by intersecting each edge of the space-time element with the hyperplane, see Fig. 7. This can be done by solving a small local system of linear algebraic equations [4].

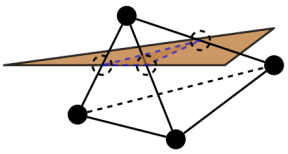

Figure 7 – An example of slicing a space-time element for a three-dimensional case. The slice cell in 3D is in general either a triangle or a quadrilateral.

The described procedure was implemented in parallel. The final output is created using unstructured VTK [12] and can be visualized, e.g., by ParaView [11] or other standard software. An example of a slice mesh distributed over two parallel processes is presented in Figures 1 and 2 (pictures were created using ParaView). The slice mesh consists of tetrahedrons and wedges. The space-time domain for this mesh was a cylinder with three-dimensional ball in the base:

$$\Omega_T = \left\{(x, y, z, t) |\, x^2 + y^2 + z^2 \leq 1, t \in [0, 1]\right\}.$$

In Fig.1 local-to-process mesh parts are shown for the slice with $t = 0.1$ plane and in Fig. 2 the mesh slices for $t = 0.1$ and $t = 0.4$ are shown.

3 Numerical experiments

In this section space-time approximation properties of the constructed meshes as well as mesh generator performance is presented.

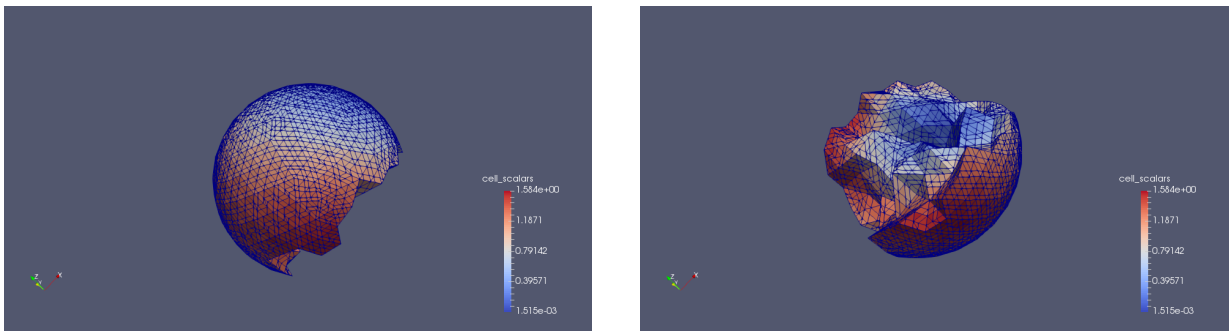

Figure 8 – Slice mesh for $t = 0.1$: process 0 (left) and process 1 (right).

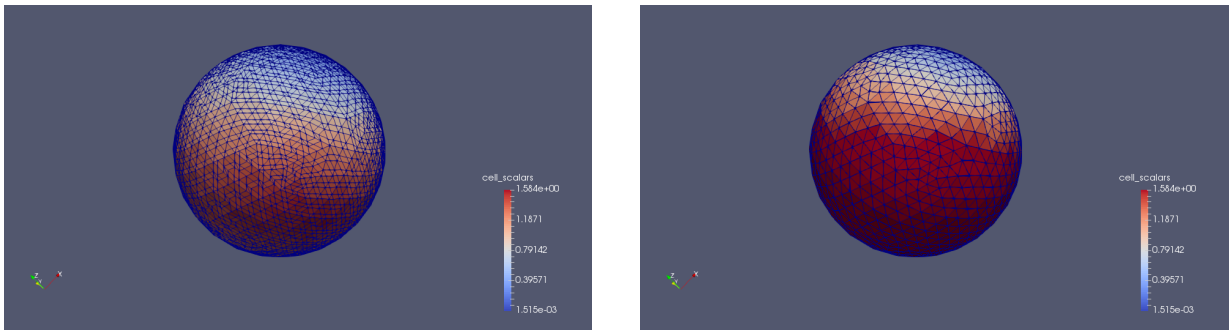

Figure 9 – Slice mesh for $t = 0.1$ (left) and $t = 0.5$ (right).

Numerical experiments were perfomed using resources of the Coeus cluster [13] at the Portland Institute of Computational Science (PICS).

3.1 Mesh generator

In general, time for solving the linear system is usually higher than time for mesh generation. Hence the goal of the performance testing for a parallel mesh generator is to show that the implemented mesh generating algotihm scales well and will not be a performance bottleneck when the number of processes increases. As already mentioned, the presented algorithm requires very few communication between the parallel processes and its nature is essentially parallel since it mainly consists of local procedures. Hence, one should expect it to show a very good scaling with respect to the number of processes. Time and memory scalability of the mesh generator are ilustrated by several simple examples below.

In Table 1 time scalability for space-time mesh generation is presented for a mesh constructed using n_{proc} number of MPI processes from a base mesh with 114,688 elements and 40 time slabs which result in 18,4 mln of elements for the space-time mesh. In the second column configuration of the parallel system is given as $n \times m$, where n is the number of computational nodes and m is the number of cores per node. In the third column the mesh generation time is presented without considering the construction of the base mesh or doing

any additional refinements afterwards. The last column shows the ratio of times between the considered configuration and the previous one (with the ideal ratio being proportional to the change in the number of computational cores which is two in the table).

Table 1 – Parallel mesh generator: Timing

n_{proc}	configuration	time (s)	ratio
1	1×1	301	-
2	2×1	144	2.1
4	4×1	71	2.02
8	8×1	36	1.97
16	16×1	18	2.0
32	16×2	9	2.0
64	16×4	4.9	1.83
128	16×8	2.8	1.75

As expected, the scaling is almost perfect when a few number of processes are used and deteriorates a little bit due to the unavoidable overhead when the local-to-process mesh parts become quite small.

The second table shows memory consumption of the mesh generator for different number of MPI processes. Here the final mesh has 24.5 mln of space-time elements and 1.2 mln of space-time nodes. The first column is again the number of processes, the second is the configuration of the system. The third column shows the peak memory required on the root node. Now the last column shows the ratio between memory consumption for the two consecutive configurations.

Table 2 – Parallel mesh generator: Memory consumption

n_{proc}	configuration	memory(Gb)	ratio
1	1×1	21.4	-
2	2×1	11.0	1.95
4	4×1	5.5	2.0
8	8×1	2.8	1.96
16	16×1	1.4	2.0

From Table 2 we can conclude that the memory scales almost linearly.

Thus, the presented parallel algorithm is able to handle large space-time meshes which can resolve the geometry of the considered problem with desired accuracy.

3.2 Approximation on space-time meshes

Space-time approximation properties of the constructed meshes are studied by the following example problem.

Consider a following linear transport problem in a space-time cylinder $\Omega_T = \Omega \times [0, T]$ where where Ω is a regular (in the most usual sense) space domain specified later:

$$\frac{\partial u}{\partial t} + \mathrm{div}_x \mathcal{L}(u) = q(\mathbf{x}, t), \quad (\mathbf{x}, t) \in \Omega_T$$

with $\mathcal{L} = \mathbf{b}u$ for a given velocity $\mathbf{b}(\mathbf{x}, t)$.

At the bottom boundary of the space time cylinder we have the initial condition

$$u(\mathbf{x}, 0) = u_0(\mathbf{x}).$$

At the lateral part of $\partial\Omega_T$ no boundary conditions are to be imposed since for our choices of velocity $\mathbf{b}$ for simplicity

$$\mathbf{b} \cdot \mathbf{n} = 0, \ (\mathbf{x}, t) \in \partial\Omega \times [0, T].$$

Then, for a constrained first-order system least squares (CFOSLS) formulation [2] we introduce

$$\boldsymbol{\sigma} = \begin{bmatrix} \mathcal{L}(u) \\ u \end{bmatrix}$$

and consider the minimization problem for the following quadratic functional:

$$J(\boldsymbol{\sigma}, u) = \| \boldsymbol{\sigma} - \begin{bmatrix} \mathcal{L}(u) \\ u \end{bmatrix} \|_0^2 + \| \mathrm{div}_{x,t} \boldsymbol{\sigma} - q \|_0^2$$

in $H(\mathrm{div}) \times L^2$ (plus appropriate boundary conditions for the first component) under the constraint

$$(\mathrm{div}_{x,t} \boldsymbol{\sigma}, w) = (q, w) \text{ for all } w \in L^2(\Omega_T).$$

Here $\| \cdot \|_0$ denotes the (vector) L^2 norm w.r.t to the space-time domain Ω_T. For more details, rigorous formulations and examples on the space-time discretizations in CFOSLS we refer to [2].

For the considered test equation we can eliminate the scalar unknown u and obtain a variational problem posed in $H(\mathrm{div})$ (for $\boldsymbol{\sigma}$) $\times L^2$ (for the Lagrange multiplier λ):

$$\begin{array}{ll} ((I - \frac{1}{\mathbf{b}^T\mathbf{b}} \mathbf{b}\mathbf{b}^T)\boldsymbol{\sigma}, \ \boldsymbol{\theta}) + +(\lambda, \ \mathrm{div}_{x,t} \boldsymbol{\theta}) & = 0, \quad \text{for all } \boldsymbol{\theta} \in H(\mathrm{div}), \\ (\mathrm{div}_{x,t} \boldsymbol{\sigma}, \ \mu) & = (f, \mu), \quad \text{for all } \mu \in L_2. \end{array}$$

To approximate the obtained problem in $H(\mathrm{div})$ - L^2 we use $(d+1)$-dimensional Raviart-Thomas finite elements [7] for $H(\mathrm{div})$ coupled with the space of discontinuous constants for L^2 which are the stable pair of f.e. spaces.

After approximation we finally arrive at the following linear system of saddle-point type:

$$\begin{bmatrix} M + BB^T & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\sigma}_h \\ \mu_h \end{bmatrix} = \begin{bmatrix} B^T f \\ f \end{bmatrix}.$$

The linear system was solved by MINRES with a block diagonal preconditioner: diagonal matrix for (1,1) and BoomerAMG preconditioner (from HYPRE [8]) for the Schur complement $B^T M^{-1} B$ for the block (2,2). One should notice here that fmatrix M is symmetric and only positive semi-definite which affects the number of iterations.

The exact solution for the first test example in 3D was:

$$u(x, y, t) = e^t \cdot \sin\left((x - 0.5)^2 + y^2\right)$$

in the cylinder $\Omega_T = \{(x, y, t) | \, x^2 + y^2 \leq 1, t \in [0, 1]\}$. The velocity was given by the formula:

$$\mathbf{b}(x, y, t) = \begin{pmatrix} -y \\ x \end{pmatrix}$$

In the table below errors for σ and u in L_2 norms, as well as the convergence order are given. Here h is the space-time mesh step.

Table 3 – Hyperbolic problem, 3D, cylinder on the circle

h	$\| \sigma \|_{L_2}$	order	$\| u \|_{L_2}$	order	iter
0.26	0.12	-	0.11	-	225
0.12	0.06	1.0	0.05	1.14	407
0.06	0.03	1.0	0.025	1.0	695
0.03	0.014	1.1	0.013	0.94	1140
0.015	0.007	1.0	0.006	1.11	1576

As one can notice, the results demonstrate almost perfect first order for both σ and u.

In 4D case we considered the following example:

$$u(x, y, z, t) = e^t \cdot \sin\left((x - 0.5)^2 + y^2 + (z - 0.25)^2\right)$$

in the cylinder with 3D ball in the base $\Omega_T = \{(x, y, z, t) | \, x^2 + y^2 + z^2 \leq 1, t \in [0, 1]\}$ with velocity:

$$\mathbf{b}(x, y, z, t) = \begin{pmatrix} -y \\ x \\ 0 \end{pmatrix}$$

Again, in the table below errors for σ and u in L_2 norms, as well as the convergence order are presented:

Table 4 – Hyperbolic problem, 4D, cylinder on the ball

h	$\| \sigma \|_{L_2}$	order	$\| \sigma \|_E$	order	$\| u \|_{L_2}$	order	iter
0.26	0.1	-	0.04	-	0.075	-	640
0.13	0.047	1.09	0.02	1.0	0.037	1.02	1282
0.065	0.023	1.03	0.01	1.0	0.018	1.04	2454

Again, along with the non-optimality of the preconditioner (the iteration number increases as $\frac{1}{h}$) one can see the first order of convergence as predicted by the theory.

The numerical solution on a coarse mesh is presented in Fig. 3. Here three slices at time momemnts $t = 0.1$, $t = 0.5$ and $t = 0.9$ are given. The color shows the norm $|\boldsymbol{\sigma}(\mathbf{x})|$ of $\boldsymbol{\sigma}$ pointwise.

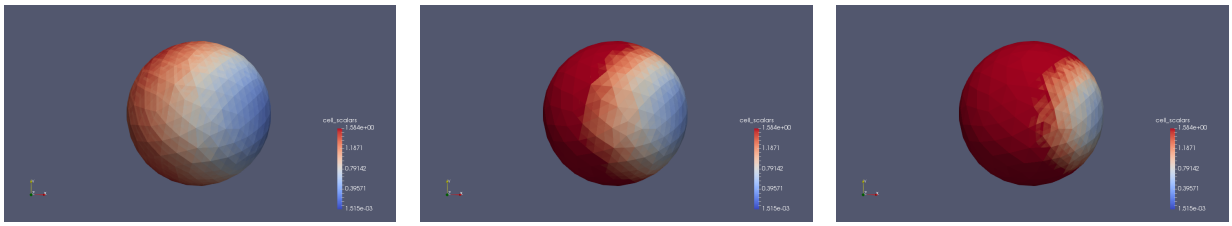

Figure 10 – Numerical solution for hyperbolic problem in 4D. Slices for $t = 0.1$ (left), $t = 0.5$ (middle) and $t = 0.9$ (right).

Acknowledgements

This work was a part of the ARO project # W911NF-15-1-0590 on space-time discretization methods. The author wants to thank the PI, Panayot Vassilevski (PSU, LLNL), and the team members: Paulina Sepulveda (PSU), Chak Shing Lee (LLNL), Martin Neumueller (Linz University).

References

[1] CFOSLS repository, https://github.com/CFOSLS/mfem.

[2] K. Voronin, C.-S. Lee, M. Neumueller, P. Sepulveda, P.S. Vassilevski. Space-time discretizations using constrained first-order system least squares (CFOSLS), Journal of Comp. Physics, Volume 373, p. 863-876, https://doi.org/10.1016/j.jcp.2018.07.024

[3] M. Behr. Simplex Space-Time Meshes in Finite Element Simulations, Int. J. Numer. Meth. Fluids 2008; 57:14211434.

[4] M. Neumueller, O. Steinbach. Refinement of flexible spacetime finite element meshes and discontinuous Galerkin methods, Comput. Visual Sci. (2011) 14:189205, DOI 10.1007/s00791-012-0174-z

[5] E. Karabelas, M. Neumueller. Generating admissible space-time meshes for moving domains in $d + 1$-dimensions, https://arxiv.org/abs/1505.03973.

[6] M. Neumueller, P.S. Vassilevski, U. Villa. Space-time CFOSLS Methods with AMGe Upscaling, 23th International Conference on Domain Decomposition Methods, 2016.

[7] P.A. Raviart, J.M. Thomas. A mixed finite element method for second order elliptic problems, Mathematical Aspects of the Finite Element Method, (I. Galligani, E. Magenes, eds.), Lectures Notes in Math. 606, Springer Verlag, 1977.

[8] HYPRE: A library of High Performance Preconditioners. https://www.llnl.gov/CASC/hypre.

[9] Modular Finite Element Methods (MFEM), http://mfem.org.

[10] Qhull. http://www.qhull.org

[11] U. Ayachit. The ParaView Guide: A Parallel Visualization Application. Kitware Inc., Clifton Park (2015).

[12] W. Schroeder, K. Martin, B. Lorensen. The Visualization Toolkit (4th ed.), Kitware, (2006), ISBN 978-1-930934-19-1.

[13] Coeus cluster. http://www.pi4cs.org/equipment.