Parallel shared-memory open-source code for simulations of transient problems using isogeometric analysis, implicit direction splitting and residual minimization (IGA-ADS-RM)

Marcin Łoś Maciej Paszynski

AGH University of Krakow, Poland.

Abstract

We present an open-source parallel shared-memory C++ software for simulations of transient phenomena on tensor product grids, with the following features: (1) it supports isogeometric finite element method discretizations; (2) it employs alternating-directions (ADS) linear cost $\mathcal{O}(N)$ solver; (3) it uses implicit time-integration schemes suitable for ADS, including Peaceman-Reachford, Douglass-Gunn, Adams-Moulton, generalized alpha, and BDF; (4) it works for 2D/ 3D problems; (5) it enables residual minimization stabilization; (6) it supports scalar, vector fields, and systems of PDEs; (7) it provides a ParaView interface; (8) it supports an interface to parallel MUMPS direct solver for problems not suitable for ADS solver; (9) it also supports interface to Preconditioned Conjugate Gradients (PCG) solver; (10) it includes a large library of problems: (a) non-stationary heat transfer (2D/3D); (b) stationary advection-diffusion (2D); (c) non-stationary advection-diffusion (2D/3D); (d) laminar flow (Stokes equations) (2D/3D); (e) Navier-Stokes (2D); (f) pollution propagation (2D/3D); (g) pathogen propagation (3D).

Keywords: Residual minimization; Isogeometric analysis; Transient problems; Discontinuous Petrov-Galerkin method; Fast solvers

Contents

1 Introduction

Simulations of transient problems require discretization in time and space. For space discretization, higher-order finite element method provides reliable and accurate solutions. The isogeometric analysis (IGA) employs higher-order and continuity B-spline basis functions for discretization with finite element method [21, 7]. The numerical problem is solved in every time step using either explicit or implicit methods. In the explicit method, the problem to solve in every time step is related to the inversion of the mass matrix, which is a simple task. The price to pay is the Courrant-Friedrichs-Levy (CFL) condition [8], telling that the larger the spatial mesh, the smaller the time step size. Implicit time integration schemes can be employed to avoid these limitations. However, the implicit time integration schemes result in expensive matrices to be factorized. One way to avoid this problem is to use IGA on regular patches of elements. It can be solved efficiently using alternating direction solver [28, 13, 34, 4], rediscovered for isogeometric analysis with variational splitting. It requires splitting the differential operator and introducing a special time-integration scheme. Thus, for the time discretization, we introduce the time integration schemes, such as Crank-Nicolson [9] or Peaceman-Reachford [29] with Strang [32] splitting, suitable for direction-splitting and fast linear cost $\mathscr{O}(N)$ spatial solver. Difficult computational problems, such as advection-dominated diffusion [16] or Navier-Stokes with high Reynolds numbers [17], require special stabilization methods. We employ the residual minimization method [6], or Discontinuous-Galerkin [30, 24] for automatic stabilization of difficult computational problems. To support parallel shared-memory computations, we employ a GALOIS environment [31, 1, 19, 23, 22].

We present an open-source parallel shared-memory C++ software for simulations of time-dependent problems with IGA discretizations on regular patches of elements. In this code, we combine the following unique features:

- we provide higher-order and continuity discretization with B-spline basis functions as provided by IGA;
- we implement alternating-directions (ADS) linear computational cost $\mathscr{O}(N)$ solver using variational splitting;
- we use implicit time-integration schemes suitable for direction splitting, including Peaceman-Reachford, Douglass-Gunn, Adams-Moulton, generalized-α, and BDF;
- our code works for two- and three-dimensional problems;
- it supports scalar, vector fields, and systems of Partial Differential Equations (PDEs);
- it provides a ParaView interface for the generation of pictures and movies of simulations;
- it supports an interface to parallel MUMPS direct solver [14, 15] for problems not suitable for ADS solver;
- it supports Preconditioned Conjugate Gradient (PCG) solver;
- it incorporates residual minimization and Discontinuous Galerkin stabilization.

We call our method an isogeometric analysis implicit direction splitting residual minimization solver (IGA-ADS-RM).

The IGA-ADS-RM code has been built based on our explicit dynamics code IGA-ADS [25]. There are the following novelties of IGA-ADS-RM with respect to the IGA-ADS code. It supports implicit dynamics suitable for direction splitting using generalized-α, Adams-Moulton (AM), backward differentiation formulae (BDF), Strang splitting with Crank-Nicolson method, Peaceman-Reachford, Douglas-Gunn and Jean-Luc Guermond and

Petar Minev implicit time integration schemes. It provides a preconditioned conjugate gradient solver. It enables residual minimization with the direction-splitting method. It enables Discontinuous Galerkin stabilization. It also provides a library, with formulations and numerical results obtained with our IGA-RM-ADS code described in several papers:

- Simulations of pollution removal by artificially generated shock waves using -diffusion model and directions splitting solver [27];
- Residual minimization with preconditioned conjugate gradients (PCG) solver for stationary advection-dominated diffusion problems [5];
- Residual minimization with direction-splitting and higher-order time integration scheme for time-dependent Stokes and Navier-Stokes problems with implicit time integration scheme [35];
- Residual minimization with direction-splitting and higher-order time integration scheme for time-dependent advection-dominated diffusion problems using Strang splitting with Crank-Nicolson and Peaceman-Reachford implicit time integration scheme [36];
- Discontinuous Galerkin and residual minimization method with Preconditioner Conjugate Gradient (PCG) solver for stabilization of Stokes solver [37];
- Stabilized simulations of COVID-19 pathogen spread using direction-splitting solver with Douglas-Gunn implicit time integration scheme [38];
- 3D simulations of the hail cannon [26].

In this paper, we focus on the code structure and implementation of these new features in the IGA-ADS-RM code.

The structure of the paper is as follows. In Section 2, we compare RM-IGA-ADS with other IGA software available. Section 3 introduces the Residual Minimization IGA method in the context of non-stationary problems. Section 4 presents a high-level overview of the structure of the code. Multiple model problems are introduced in Section 5, together with annotated example implementations. Section 6 presents the numerical results of these example codes. Section 7 describes an automatic mesh refinement strategy based on the residual provided by the residual minimization method. Section 8 discusses various approaches to improving integration efficiency, and their applicability to the common use cases of our code. Finally, section 9 presents scalability results on shared-memory architecture machines.

2 Comparison of RM-IGA-ADS solver with alternative solvers

There are several high-quality numerical solvers for running simulations using B-spline discretizations. The list includes the PetIGA solver [10], GeoPDE solver [33], and IGA-ADS solver [25]. The main differences between PetIGA and the RM-IGA-ADS solver are the following

- PetIGA software is linked with PETSc, which provides several direct and iterative solvers
- RM-IGA-ADS relies on the linear computational cost alternating direction solver. The alternating direction solver is a fast implementation of the direct solver for tensor product geometries. It also supports an interface to a dedicated PCG iterative solver for residual minimization computations.
- Both RM-IGA-ADS and PETSc support an interface to the MUMPS solver.
- PetIGA supports arbitrary geometries of the computational domain, while RM-IGA-ADS works on tensor product geometries.
- RM-IGA-ADS contains the residual minimization method for automatic stabilization of difficult simulations, using Galerkin and DG discretizations, and it provided several build examples, including advection-dominated diffusion, Stokes, and Navier-Stokes problem. The implementation of the residual minimization method in PetIGA is possible, but it is not straightforward. Implementing the residual minimization and DG discretization methods in PetIGA is possible, but it is not straightforward.
- The RM-IGA-ADS is straightforward to install and use.
- RM-IGA-ADS supports parallelization on multi-core parallel machines using the Galois library, while the PetIGA supports parallel solvers available through the PETSc library.

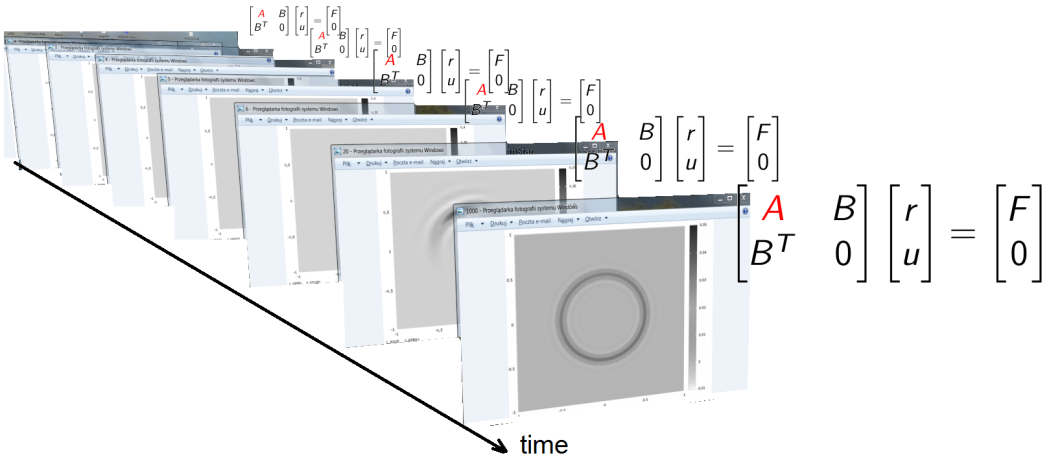


Figure 1: Stabilization of time-steps with residual minimization method

The differences between GeoPDE and RM-IGA-ADS are summarized below:

- GeoPDE is a MATLAB code that supports local adaptivity with hierarchical B-splines. The RM-IGA-ADS only supports global refinement on the tensor structure mesh.
- GeoPDE employs MATLAB direct and iterative solvers, while RM-IGA-ADS supports dedicated alternating directions and linear computational cost solvers. RM-IGA-ADS supports an interface to the MUMPS solver and a dedicated PCG iterative solver for residual minimization computations.
- RM-IGA-ADS is dedicated to residual minimization computations, while GeoPDE supports different formulations within the Galerkin method
- GeoPDE supports arbitrary geometries, while RM-IGA-ADS works on tensor product grids.

Finally, RM-IGA-ADS is an extension of IGA-ADS into residual minimization and Discontinuous Galerkin formulations.

3 Non-stationary residual minimization method

Our code employs unconditionally stable higher-order time integration schemes. It stabilizes time steps with the residual minimization method (see Figure 2). The spatial discretization is based on B-splines. Namely,

- It employs time-integration schemes suitable for direction-splitting (Strang scheme with Backward Euler method, Strang scheme with Crank-Nicolson method, generalized α method, Adams-Moultan (AM), Backward Differentiation Formulae (BDF), Peaceman Rachford, Douglass-Gun, as well as Jean-Luc Guermond scheme for Navier-Stokes equations).
- It employs unconditionally stable time integration schemes (Strang scheme with Crank-Nicolson method, generalized α method, Adams-Moultan (AM), Backward Differentiation Formulae (BDF), Peaceman Rachford, Douglass-Gun, as well as Jean-Luc Guermond scheme for Navier-Stokes equations).
- It employs higher-order time integration schemes (generalized α method, Adams-Moultan (AM), Backward Differentiation Formulae (BDF)).
- It solves the separate problems in each time step, using B-spline-based discretization.
- The stabilization in time is obtained using the residual minimization method (see Appendix A). In this method, we solve the saddle-point problem in every time step,

$$\begin{matrix}\text{test}\ \{\\ \text{trial}\ \{\end{matrix}\underbrace{\begin{bmatrix} G & B^T \\ B & 0 \end{bmatrix}}_{test\quad trial}\begin{bmatrix} r \\ u \end{bmatrix} = \begin{bmatrix} F \\ 0 \end{bmatrix}, \tag{1}$$

  where G is the Gram matrix, and B is the problem matrix. There are different discretizations for the trial and test space.

The residual minimization method can be derived in the following way. For a general weak problem: Find $u \in U$ such that

$$b(u,v) = l(v), \quad \forall v \in V, \tag{2}$$

we define the operator $B : U \to V'$ such as

$$\langle Bu, v\rangle = b(u,v), \tag{3}$$

where $\langle \cdot, \cdot \rangle$ is the duality pairing between V and V'. Now (2) is

$$Bu - l = 0. \tag{4}$$

We wish to minimize the residual

$$u_h = argmin_{w_h \in U_h} \frac{1}{2} \|Bw_h - l\|^2_{V'}. \tag{5}$$

We introduce the Riesz operator $R_V : V \ni v \to (v,.) \in V'$ being the isometric isomorphism to project the problem back to V

$$u_h = argmin_{w_h \in U_h} \frac{1}{2} \|R_V^{-1}(Bw_h - l)\|^2_V \tag{6}$$

The minimum is attained at u_h when the Gâteaux derivative is equal to 0 in all directions

$$G(u_h) = \frac{1}{2} \|R_V^{-1}(Bu_h - l)\|^2_V \tag{7}$$

$$dG(u_h; w_h) = \lim_{h \to 0} \frac{G(u_h + h w_h) - G(u_h)}{h}$$

since $\|a+b\|^2 = \|a\|^2 + 2(a,b) + \|b\|^2$ we have

$$\begin{aligned} 2G(u_h + h w_h) = \|R_V^{-1}(B(u_h + h w_h) - l)\|^2_V = \\ \|R_V^{-1}(Bu_h - l) + h R_V^{-1} B w_h\|^2_V = \\ \|R_V^{-1}(Bu_h - l)\|^2_V + 2h \left(R_V^{-1}(Bu_h - l), R_V^{-1} B w_h\right)_V + h^2 \|R_V^{-1} B w_h\|^2_V \end{aligned}$$

so

$$\frac{G(u_h + h w_h) - G(u_h)}{h} = \left(R_V^{-1}(Bu_h - l), R_V^{-1} B w_h\right)_V + \frac{1}{2} h \|R_V^{-1} B w_h\|^2_V$$

In the limit $h \to 0$ we have

$$dG(u_h; w_h) = \left(R_V^{-1}(Bu_h - l), R_V^{-1} B w_h\right)_V \tag{8}$$

If u_h is minimum, then for each w_h we have $G(u_h + h w_h)$ gets minimum for $h = 0$, so the Gâteaux derivative has to be zero

$$(R_V^{-1}(Bu_h - l), R_V^{-1}(B w_h))_V = 0 \quad \forall w_h \in U_h \tag{9}$$

We define the residual $r = R_V^{-1}(Bu_h - l)$ and we get

$$(r, R_V^{-1}(B w_h)) = 0 \quad \forall w_h \in U_h \tag{10}$$

From the definition of R_V for all functionals $f \in V'$

$$(v, R_V^{-1} f)_V = \langle f, v\rangle \ (= f(v) \text{ from definition of } \langle \cdot, \cdot \rangle) \tag{11}$$

so in particular for $f = Bw_h$ and $v = r$ we get

$$\langle Bw_h, r\rangle = 0 \qquad \forall w_h \in U_h \tag{12}$$

From the definition of the residual we have

$$(r,v)_V = \langle Bu_h - l, v\rangle, \quad \forall v \in V. \tag{13}$$

Thus, from (12) and (13), our problem reduces to the following semi-infinite problem: Find $(r, u_h) \in V \times U_h$ such as

$$\begin{array}{lll} (r,v)_V & -\langle Bu_h, v\rangle = \langle l, v\rangle, & \forall v \in V, \\ \langle Bw_h, r\rangle & = 0, & \forall w_h \in U_h. \end{array} \tag{14}$$

We discretize the test space $V_h \in V$ to get the discrete problem: Find $(r_h, u_h) \in V_h \times U_h$ such as

$$\begin{array}{lll} (r_h, v_h)_{V_h} & -\langle Bu_h, v_h\rangle = \langle l, v_h\rangle, & \forall v \in V_h, \\ \langle Bw_h, r_h\rangle & = 0, & \forall w_h \in U_h, \end{array} \tag{15}$$

where $(\cdot,\cdot)_{V_h}$ is an inner product in V_h, $\langle Bu_h, v_h\rangle = b(u_h, v_h)$, and $\langle Bw_h, r_h\rangle = b(w_h, r_h)$.

4 Structure of the code

Running each simulation requires overwriting methods from the following class. The methods `before` and `after` are called once at the beginning and at the end of the simulation. The methods `before_step` and `after_step` are called before and after each time step of the simulation. The method `step` implements the time step of the simulation.

```
class simulation_base {
    // executed once before the entire simulation
    virtual void before() { }
    // executed once after the entire simulation
    virtual void after() { }
    // executed before each time step
    virtual void before_step(int iter, double t) { }
    // timestep itself - put all the calculations here
    virtual void step(int iter, double t) { }
    // executed after each time step
    virtual void after_step(int iter, double t) { }

    // executes teh whole simulation (all the timesteps)
    void run();
};

// dimension-specific base classes
class simulation_2d : public simulation_base { ... };
class simulation_3d : public simulation_base { ... };
```

We will explain the sketch of the simulation using the three-dimensional model heat transfer problem. We seek the temperature distribution $[0,1]^3 \times [0,T] = \Omega \times I \ni (x,y,z,t) \to u(x,y,z,t) \in \mathscr{R}$ such that

$$\begin{cases} \partial_t u = \Delta u & \text{in } \Omega \times I, \\ \nabla u \cdot \hat{n} = 0 & \text{in } \partial\Omega \times I, \\ u(\cdot,0) = u_0 & \text{in } \Omega. \end{cases}$$

We define our simulation class by deriving from one of the two base classes for 2D and 3D problems:

```
class heat_3d : public simulation_3d  { ... };
```

We introduce Forward Euler method where $\partial_t u^t = \frac{u^{t+1}-u^t}{\tau}$ with time step size τ. We derive the weak formulation from the integration by parts $\left(u_h^{n+1}, v_h\right) = \left(u_h^n, v_h\right) - \tau\left(\nabla u_h^n, \nabla v_h\right)$. The simulation is configured in `main` method with quadratic B-splines on $12 \times 12 \times 12$ mesh. We run 5000 times step with $\tau = 10^{-7}$. We will need the first derivatives to implement the weak formulation.

```
#include "heat_3d.hpp"

int main() {
    // 12x12x12 mesh, quadratic B-spline basis functions
    ads::dim_config dim{2, 12};
    // 5,000 time steps, dt = 10^-7
    ads::timesteps_config steps{5000, 1e-7};
    // formulation uses values and first derivatives
    int ders = 1;

    ads::config_3d c{dim, dim, dim, steps, ders};
    ads::problems::heat_3d sim{c};
    sim.run();
}
```

The `heat_3d` class provides the pilot of the simulation. The `before` method for the explicit solver generates the mass matrices and computes the projection of the initial state. The `before_step` method stores the previous (or initial) time step solution. The `step` method generates the right-hand side and solves the L2 projection problem with mass matrix for the explicit solver. The problem formulation is implemented in the `compute_rhs` method. The term `u.val * v.val` corresponds to $\left(u_h^n, v_h\right)$, the `- steps.dt * gradient_prod` term corresponds to $-\tau\left(\nabla u_h^n, \nabla v_h\right)$, where `gradient_prod = grad_dot(u, v);`.

```
#include "ads/executor/galois.hpp"
#include "ads/output_manager.hpp"
#include "ads/simulation.hpp"

class heat_3d : public simulation_3d {
```

```
private:
    using Base = simulation_2d;
    vector_type u, u_prev; // current and previous solution

    output_manager<3> output;
    galois_executor executor{4};

public:
    explicit heat_3d(const config_3d& config)
    : Base{config}
    , u{shape()}
    , u_prev{shape()}
    , output{x.B, y.B, 50} { }

    // member functions
    // ...
};
```

```
// called once at the beginning of the simulation
void before() override {
    prepare_matrices();
    auto init = [this](double x, double y, double z) {
        /* initial state formula */
    };
    // L2 projection of initial state
    projection(u, init);
    solve(u);
}

// called before every step
void before_step(int /*iter*/, double /*t*/) override {
    using std::swap;
    swap(u, u_prev);
}

// time step implementation
void step(int /*iter*/, double /*t*/) override {
    compute_rhs();
    solve(u);
}
```

```
void compute_rhs() {
    auto& rhs = u;
    zero(rhs);

    // parallel loop over mesh elements
    executor.for_each(elements(), [&](index_type e) {
        auto U = element_rhs(); // buffer for element DoF entries
        double J = jacobian(e);
        for (auto q : quad_points()) {
            double w = weight(q);
            for (auto a : dofs_on_element(e)) {
                auto aa = dof_global_to_local(e, a);
                // compute values and gradients at the quadrature point
                value_type v = eval_basis(e, q, a);
                value_type u = eval_fun(u_prev, e, q);

                double gradient_prod = grad_dot(u, v);
                double val = u.val * v.val - steps.dt * gradient_prod;
                U(aa[0], aa[1]) += val * w * J;
            }
        }
        executor.synchronized([&]() { update_global_rhs(rhs, U, e); });
    });
    integration_timer.stop();
}
```

5 Collection of exemplary problems

In this section, we will present how to implement several exemplary problems in the RM-IGA-ADS code.

5.1 Time-dependent advection-dominated diffusion problem

We start with the unstable time-dependent advection-dominated diffusion problem. We seek the scalar concentration field $[0,1]^2 \times [0,T] = \Omega \times I \ni (x,y,t) \to u(x,y,z,t) \in \mathscr{R}$ such that

$$\begin{cases} \partial_t u - \varepsilon \Delta u + \beta \cdot \nabla u = f & \text{in } \Omega \times I, \\ u|_{\partial\Omega} = 0 & \text{in } \partial\Omega \times I, \\ u(\cdot,0) = u_0 & \text{in } \Omega. \end{cases}$$

Here ε is the diffusion constant, $\beta = (\beta_x, \beta_y)$ stands for the advection "wind", $f(x,y,t)$ is the source term. This problem is unstable when $\varepsilon << \|\beta\|$. We employ one of the time-integration schemes (Backward Euler, Crank-Nicolson, Peaceman-Rachford) using the splitting of the operator as $-\varepsilon\Delta u + \beta \cdot \nabla u = \underbrace{-\varepsilon\,\partial_{xx}u + \beta_x \partial_x u}_{\mathscr{L}_1}$ $\underbrace{-\varepsilon\,\partial_{yy}u + \beta_y \partial_y u}_{\mathscr{L}_2}$. For the Peaceman-Richford time integration scheme we employ

$$\begin{cases} \dfrac{u^{n+1/2} - u^n}{\tau/2} + \mathscr{L}_1 u^{n+1/2} = f^{n+1/2} - \mathscr{L}_2 u^n, \\ \dfrac{u^{n+1} - u^{n+1/2}}{\tau/2} + \mathscr{L}_2 u^{n+1} = f^{n+1/2} - \mathscr{L}_1 u^{n+1/2}. \end{cases} \tag{16}$$

The variational formulation of the scheme (16) is

$$\begin{cases} (u^{n+1/2}, v) + \dfrac{\tau}{2}\left(\alpha \dfrac{\partial u^{n+1/2}}{\partial x}, \dfrac{\partial v}{\partial x}\right) + \dfrac{\tau}{2}\left(\beta_x \dfrac{\partial u^{n+1/2}}{\partial x}, v\right) = \\ (u^n, v) - \dfrac{\tau}{2}\left(\alpha \dfrac{\partial u^n}{\partial y}, \dfrac{\partial v}{\partial y}\right) - \dfrac{\tau}{2}\left(\beta_y \dfrac{\partial u^n}{\partial y}, v\right) + \dfrac{\tau}{2}(f^{n+1/2}, v), \\ \\ (u^{n+1}, v) + \dfrac{\tau}{2}\left(\alpha \dfrac{\partial u^{n+1}}{\partial y}, \dfrac{\partial v}{\partial y}\right) + \dfrac{\tau}{2}\left(\beta_y \dfrac{\partial u^{n+1}}{\partial y}, v\right) = \\ (u^{n+1/2}, v) - \dfrac{\tau}{2}\left(\alpha \dfrac{\partial u^{n+1/2}}{\partial x}, \dfrac{\partial v}{\partial x}\right) - \dfrac{\tau}{2}\left(\beta_x \dfrac{\partial u^{n+1/2}}{\partial x}, v\right) + \dfrac{\tau}{2}(f^{n+1/2}, v), \end{cases} \tag{17}$$

Here $(\cdot,\cdot)$ denotes the inner product of $L^2(\Omega)$. In the matrix form, this scheme translates into Kronecker product matrices

$$\begin{cases} \left[M^x + \dfrac{\tau}{2}(K^x + G^x)\right] \otimes M^y u^{n+1/2} = \\ \quad M^x \otimes \left[M^y - \dfrac{\tau}{2}(K^y + G^y)\right] u^n + \dfrac{\tau}{2} F^{n+1/2}, \\ M^x \otimes \left[M^y + \dfrac{\tau}{2}(K^y + G^y)\right] u^{n+1} = \\ \quad \left[M^x - \dfrac{\tau}{2}(K^x + G^x)\right] \otimes M^y u^{n+1/2} + \dfrac{\tau}{2} F^{n+1/2}, \end{cases} \tag{18}$$

where $M^{x,y}$, $K^{x,y}$ and $G^{x,y}$ are the 1D mass, stiffness and advection matrices, respectively.

Alternatively, in the Strang splitting scheme we divide the problem $u_t + \mathscr{L}u = f$ into s

$$\begin{cases} P_1 : u_t + \mathscr{L}_1 u = f, \\ P_2 : u_t + \mathscr{L}_2 u = 0, \end{cases} \tag{19}$$

the scheme integrates the solution from t_n to t_{n+1} into sub-steps (see Figure 2):

$$\begin{cases} \text{Solve } P_1 : u_t + \mathscr{L}_1 u = f, \text{ in } (t_n, t_{n+1/2}), \\ \text{Solve } P_2 : u_t + \mathscr{L}_2 u = 0, \text{ in } (t_n, t_{n+1}), \\ \text{Solve } P_1 : u_t + \mathscr{L}_1 u = f, \text{ in } (t_{n+1/2}, t_{n+1}), \end{cases} \tag{20}$$

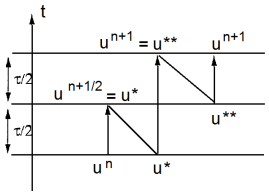


Figure 2: Strang splitting. In this method, we split the differential operator into two parts, we perform half step forward with the first part of the operator, followed by the full step forward with the second part of the operator, and finally, half step forward with the first part of the operator.

and we can employ different methods in each sub-step. The Strang splitting with Backward Euler method reads

$$\begin{cases} \dfrac{u^{n+1/2}-u^n}{\tau/2}+\mathscr{L}_1 u^{n+1/2}=f^{n+1/2}, \\ \dfrac{u^{n+1}-u^n}{\tau}+\mathscr{L}_2 u^{n+1}=0, \\ \dfrac{u^{n+1}-u^{n+1/2}}{\tau/2}+\mathscr{L}_1 u^{n+1}=f^{n+1}. \end{cases} \tag{21}$$

and the weak form

$$\begin{cases} (u^{n+1/2},v)+\dfrac{\tau}{2}\left(\alpha\dfrac{\partial u^{n+1/2}}{\partial x},\dfrac{\partial v}{\partial x}\right)+\dfrac{\tau}{2}\left(\beta_x\dfrac{\partial u^{n+1/2}}{\partial x},v\right) \\ \quad =(u^n,v)+\dfrac{\tau}{2}(f^{n+1/2},v), \\ (u^{n+1},v)+\tau\left(\alpha\dfrac{\partial u^{n+1}}{\partial y},\dfrac{\partial v}{\partial y}\right)+\tau\left(\beta_y\dfrac{\partial u^{n+1}}{\partial y},v\right)=(u^n,v), \\ (u^{n+1},v)+\dfrac{\tau}{2}\left(\alpha\dfrac{\partial u^{n+1}}{\partial x},\dfrac{\partial v}{\partial x}\right)+\dfrac{\tau}{2}\left(\beta_x\dfrac{\partial u^{n+1}}{\partial x},v\right) \\ \quad =(u^{n+1/2},v)+\dfrac{\tau}{2}(f^{n+1},v), \end{cases} \tag{22}$$

In the matrix form, the Strang splitting scheme with Backward Euler method reads

$$\begin{cases} \left[M^x+\dfrac{\tau}{2}(K^x+G^x)\right]\otimes M^y u^*=M^x\otimes M^y u^n+\dfrac{\tau}{2}F^{n+1/2}, \\ M^x\otimes\left[M^y+\tau(K^y+G^y)\right]u^{**}=M^x\otimes M^y u^*, \\ \left[M^x+\dfrac{\tau}{2}(K^x+G^x)\right]\otimes M^y u^{n+1}=M^x\otimes M^y u^{**}+\dfrac{\tau}{2}F^{n+1}. \end{cases} \tag{23}$$

Finally, if we select the Crank-Nicolson method for Strang scheme we obtain

$$\begin{cases} \dfrac{u^{n+1/2}-u^n}{\tau/2}+\dfrac{1}{2}(\mathscr{L}_1 u^{n+1/2}+\mathscr{L}_1 u^n)=\dfrac{1}{2}(f^{n+1/2}+f^n), \\ \dfrac{u^{n+1}-u^n}{\tau}+\dfrac{1}{2}(\mathscr{L}_2 u^{n+1}+\mathscr{L}_2 u^n)=0, \\ \dfrac{u^{n+1}-u^{n+1/2}}{\tau/2}+\dfrac{1}{2}(\mathscr{L}_1 u^{n+1}+\mathscr{L}_1 u^{n+1/2})=\dfrac{1}{2}(f^{n+1}+f^{n+1/2}). \end{cases}$$

This Strang splitting scheme with Crank-Nicolson method in the weak form

$$
\begin{cases}
(u^{n+1/2},v)+\dfrac{\tau}{4}\left(\alpha\dfrac{\partial u^{n+1/2}}{\partial x},\dfrac{\partial v}{\partial x}\right)+\dfrac{\tau}{4}\left(\beta_x\dfrac{\partial u^{n+1/2}}{\partial x},v\right)= \\
=(u^{n},v)-\dfrac{\tau}{4}\left(\alpha\dfrac{\partial u^{n}}{\partial x},\dfrac{\partial v}{\partial x}\right)-\dfrac{\tau}{4}\left(\beta_x\dfrac{\partial u^{n}}{\partial x},v\right)+\dfrac{\tau}{4}(f^{n+1/2}+f^{n},v), \\
\\
(u^{n+1},v)+\dfrac{\tau}{2}\left(\alpha\dfrac{\partial u^{n+1}}{\partial y},\dfrac{\partial v}{\partial y}\right)+\dfrac{\tau}{2}\left(\beta_y\dfrac{\partial u^{n+1}}{\partial y},v\right)= \\
=(u^{n},v)-\dfrac{\tau}{2}\left(\alpha\dfrac{\partial u^{n}}{\partial y},\dfrac{\partial v}{\partial y}\right)-\dfrac{\tau}{2}\left(\beta_y\dfrac{\partial u^{n}}{\partial y},v\right), \\
\\
(u^{n+1},v)+\dfrac{\tau}{4}\left(\alpha\dfrac{\partial u^{n+1}}{\partial x},\dfrac{\partial v}{\partial x}\right)+\dfrac{\tau}{4}\left(\beta_x\dfrac{\partial u^{n+1}}{\partial x},v\right)= \\
=(u^{n+1/2},v)-\dfrac{\tau}{4}\left(\alpha\dfrac{\partial u^{n+1/2}}{\partial x},\dfrac{\partial v}{\partial x}\right)-\dfrac{\tau}{4}\left(\beta_x\dfrac{\partial u^{n+1/2}}{\partial x},v\right)+\dfrac{\tau}{4}(f^{n+1}+f^{n+1/2},v).
\end{cases}
\tag{24}
$$

and in the matrix form

$$
\begin{cases}
\left[M^x+\dfrac{\tau}{4}(K^x+G^x)\right]\otimes M^y u^* = \\
\quad\left[M^x-\dfrac{\tau}{4}(K^x+G^x)\right]\otimes M^y u^n+\dfrac{\tau}{4}(F^{n+1/2}+F^n), \\
M^x\otimes\left[M^y+\dfrac{\tau}{2}(K^y+G^y)\right]u^{**}=M^x\otimes\left[M^y-\dfrac{\tau}{2}(K^y+G^y)\right]u^*, \\
\left[M^x+\dfrac{\tau}{4}(K^x+G^x)\right]\otimes M^y u^{n+1} = \\
\quad\left[M^x-\dfrac{\tau}{4}(K^x+G^x)\right]\otimes M^y u^{**}+\dfrac{\tau}{4}(F^{n+1}+F^{n+1/2}).
\end{cases}
\tag{25}
$$

The implementation of different time marching schemes is available in the `step` routine. The system of linear equations generated in sub-step depends on the selected `method` that follows `enum class scheme { BE, CN, peaceman_rachford, strang_BE, stranf_CN }` corresponding to Backward Euler, Crank-Nicolson, Peaceman-Rachfored, Strang (also called the leapfrog) with either Backward-Euler or Crank-Nicolson implementation.

```
void step(int /*iter*/, double t) override {
    // auxiliary definitions
    auto dt = steps.dt;
    auto f = [&](point_type x, double s) {
        return forcing(x[0], x[1], epsilon, s);
    };
    auto F = [&](double s) {
        return [&, s](point_type x) { return f(x, s); };
    };
    auto Favg = [&](double s1, double s2) {
        return [=, &f](point_type x) {
            return 0.5 * (f(x, s1) + f(x, s2));
        };
    };
    auto zero = [&](point_type) { return 0; };

    // system generation depends on the scheme
    // schemes are defined as enumeration type
    // enum class scheme { BE, CN, peaceman_rachford, strang_BE, strang_CN };
    if (method == scheme::BE) {
        substep(u, true, true, dt, dt, 0, 0, dt, F(t + dt));
    }
    if (method == scheme::CN) {
        substep(u, true, true,   dt/2, dt/2, -dt/2, -dt/2,   dt, Favg(t, t + dt));
    }
    if (method == scheme::peaceman_rachford) {
        substep(u, true, true,   dt/2,    0,     0, -dt/2,   dt/2, F(t + dt/2));
```

```
        substep(u, true, true,       0, dt/2, -dt/2,      0,    dt/2, F(t + dt/2));
    }
    // strang is a temmplate for splitting schemes - substeps can be handled
    // using any ordinary time marching scheme
    if (method == scheme::strang_BE) {
        substep(u, false, true,     dt/2,  0,    0, 0,    dt/2, F(t + dt/2));
        substep(u, true,  false,       0, dt,    0, 0,      dt, zero);
        substep(u, false, true,     dt/2,  0,    0, 0,    dt/2, F(t + dt));
    }
    if (method == scheme::strang_CN) {
        substep(u, false,  true,    dt/4,     0,    -dt/4,      0,    dt/2, Favg(t, t +
            dt/2));
        substep(u, true,  false,       0, dt/2,        0, -dt/2,      dt, zero);
        substep(u, false,  true,    dt/4,     0,    -dt/4,      0,    dt/2, Favg(t + dt
            /2, t + dt));
    }
}
```

The `step` method, depending on the selected scheme, calls the `substep` method several times, parameterized with the selected time integration scheme, to obtain a structure required for a sub-step of the scheme.

```
// Computes the next substep solution solving
//   (1 + Lx_lhs Lx + Ly_lhs Ly) u_next = (1 + Lx_rhs Lx + Ly_rhs Ly) u_prev + F
// using iGRM with scalar product
//   (u, v) + sx (Dx u, Dx v) + sy (Dy u, Dy v)
template <typename Fun>
void substep(vector_type& u, bool x_refine, bool y_refine,
             double Lx_lhs, double Ly_lhs,
             double Lx_rhs, double Ly_rhs, double dt, Fun&& f) {
    // choose appropriate test space
    // in iGRM we refine the space in one direction only
    dimension& Vx = x_refine ? this->Vx : Ux;
    dimension& Vy = y_refine ? this->Vy : Uy;

    // buffer for solution and residual representation
    vector_view r_rhs{full_rhs.data(), {Vx.dofs(), Vy.dofs()}};
    vector_view u_rhs{full_rhs.data() + r_rhs.size(), {Ux.dofs(), Uy.dofs()}};

    std::fill(begin(full_rhs), end(full_rhs), 0);
    compute_rhs(Lx_rhs, Ly_rhs, Vx, Vy, r_rhs, u_rhs, dt, std::forward<Fun>(f));

    zero_bc(r_rhs, Vx, Vy);
    zero_bc(u_rhs, Ux, Uy);

    // build the system matrix and solve using MUMPS
    int size = Vx.dofs() * Vy.dofs() + Ux.dofs() * Uy.dofs();
    mumps::problem problem(full_rhs.data(), size);
    double sx = x_refine ? 0 : 1;
    double sy = y_refine ? 0 : 1;
    assemble_problem(problem, Lx_lhs, Ly_lhs, sx, sy, Vx, Vy, matrices(x_refine,
        y_refine));
    solver.solve(problem);

    copy_solution(u_rhs, r_rhs, u);
}
```

The `substep` method calls the `assemble_problem` to obtain a system of linear equations solved in a given sub-step. Namely, we build the system of linear equations resulting from the application of the residual minimization method. The Gram matrix A is built using the assumed inner product. The problem matrix B is constructed according to the selected time-integration method and is suitable for direction splitting. The problem matrix B for the residual minimization method is built using trial basis functions for rows and test basis functions for columns. The B^T matrix is constructed in a symmetric way, with test basis function in rows and trial basis functions in columns.

```
void assemble_problem(mumps::problem& problem, double cx, double cy,
                      double sx, double sy, const dimension& Vx,
                      const dimension& Vy, const matrix_set& M) {
    auto N = Vx.dofs() * Vy.dofs();

    // Gram matrix
    for (auto i : internal_dofs(Vx, Vy)) {
        for (auto j : overlapping_internal_dofs(i, Vx, Vy)) {
            int ii = linear_index(i, Vx, Vy) + 1;
```

```
            int jj = linear_index(j, Vx, Vy) + 1;

            // explicit construction of the Kronecker product
            double val = kron(M.MVx, M.MVy, i, j)
                       + sx * kron(M.KVx, M.MVy, i, j)
                       + sy * kron(M.MVx, M.KVy, i, j);
            problem.add(ii, jj, val);
        }
    }

    // Dirichlet BC - upper left
    for_boundary_dofs(Vx, Vy, [&](index_type dof) {
        int i = linear_index(dof, Vx, Vy) + 1;
        problem.add(i, i, 1);
    });

    // B, B^T
    for (auto i : dofs(Vx, Vy)) {
        for (auto j : dofs(Ux, Uy)) {
            double MM = kron(M.MUVx, M.MUVy, i, j);
            double Lx =
                c_diff[0] * kron(M.KUVx, M.MUVy, i, j) + beta[0] * kron(M.AUVx, M.
                    MUVy, i, j);
            double Ly =
                c_diff[1] * kron(M.MUVx, M.KUVy, i, j) + beta[1] * kron(M.MUVx, M.
                    AUVy, i, j);
            double val = MM + cx * Lx + cy * Ly;

            if (val != 0) {
                if (!is_boundary(i, Vx, Vy) && !is_boundary(j, Ux, Uy)) {
                    int ii = linear_index(i, Vx, Vy) + 1;
                    int jj = linear_index(j, Ux, Uy) + 1;

                    problem.add(ii, N + jj, -val);
                    problem.add(N + jj, ii, val);
                }
            }
        }
    }

    // Dirichlet BC - lower right
    for_boundary_dofs(Ux, Uy, [&](index_type dof) {
        int i = linear_index(dof, Ux, Uy) + 1;
        problem.add(N + i, N + i, 1);
    });
}
```

```
    // B, B^T
    for (auto i : dofs(Vx, Vy)) {
        for (auto j : dofs(Ux, Uy)) {
            double MM = kron(M.MUVx, M.MUVy, i, j);
            double Lx = c_diff[0] * kron(M.KUVx, M.MUVy, i, j)
                      + beta[0] * kron(M.AUVx, M.MUVy, i, j);
            double Ly = c_diff[1] * kron(M.MUVx, M.KUVy, i, j)
                      + beta[1] * kron(M.MUVx, M.AUVy, i, j);
            double val = MM + cx * Lx + cy * Ly;

            bool bd = is_boundary(i, Vx, Vy) || is_boundary(j, Ux, Uy);
            if (val != 0 && !bd) {
                int ii = linear_index(i, Vx, Vy) + 1;
                int jj = linear_index(j, Ux, Uy) + 1;
                problem.add(ii, N + jj, -val);
                problem.add(N + jj, ii, val);
            }
        }
    }
    // Dirichlet BC - lower right
    for_boundary_dofs(Ux, Uy, [&](index_type dof) {
        int i = linear_index(dof, Ux, Uy) + 1;
        problem.add(N + i, N + i, 1);
    });
}
```

Finally, the right-hand side appropriate for a selected time integration scheme and sub-step structure is generated within `compute_rhs` method.

```
// compute generic RHS of the iGRM system: [-L 0]
template <typename Fun>
void compute_rhs(double cx, double cy, const dimension& Vx, const dimension& Vy,
                 vector_view& r_rhs, vector_view& u_rhs, double dt, Fun&& F) {
    // parallel loop over the mesh elements
    executor.for_each(elements(Vx, Vy), [&](index_type e) {
        auto R = vector_type{{Vx.basis.dofs_per_element(), Vy.basis.dofs_per_element
            ()}};
        auto U = vector_type{{Ux.basis.dofs_per_element(), Uy.basis.dofs_per_element
            ()}};

        double J = jacobian(e);
        for (auto q : quad_points(Vx, Vy)) {
            double W = weight(q);
            double WJ = W * J;
            auto x = point(e, q);
            value_type uu = eval(u, e, q, Ux, Uy);

            for (auto a : dofs_on_element(e, Vx, Vy)) {
                auto aa = dof_global_to_local(e, a, Vx, Vy);
                value_type v = eval_basis(e, q, a, Vx, Vy);

                double M = uu.val * v.val;
                double Lx = c_diff[0] * uu.dx * v.dx + beta[0] * uu.dx * v.val;
                double Ly = c_diff[1] * uu.dy * v.dy + beta[1] * uu.dy * v.val;
                double lv = M + cx * Lx + cy * Ly + dt * F(x) * v.val;
                double val = -lv;
                R(aa[0], aa[1]) += val * WJ;
            }
        }
        executor.synchronized([&]() {
            update_global_rhs(r_rhs, R, e, Vx, Vy);
            update_global_rhs(u_rhs, U, e, Ux, Uy); // U is always 0
        });
    });
}
```

5.2 Time-dependent Navier-Stokes problem

As the second example, we consider two-dimensional Navier-Stokes equations. We seek the vector velocity field $[0,1]^2 \times [0,T] = \Omega \times I \ni (x,y,t) \to (u_1(x,y,t), u_2(x,y,t)) \in \mathscr{R}^2$ and the scalar pressure field $[0,1]^2 \times [0,T] = \Omega \times I \ni (x,y,t) \to p(x,y,t) \in \mathscr{R}$ such that

$$\begin{cases} \partial_t v + v \cdot \nabla v - \dfrac{1}{Re} \Delta v + \nabla p = f & \text{in } \Omega \times I \\ \nabla \cdot v = 0 & \text{in } \Omega \times I \\ v|_{\partial\Omega} = g & \text{in } \partial\Omega \times I \\ v(\cdot,0) = v^0 & \text{in } \Omega \\ p(\cdot,0) = p^0 & \text{in } \Omega \end{cases}$$

The first equation is really a system of two equations, since $v = (v_1, v_2)$, so $f(x,y,t) = (f_1(x,y,t), f_2(x,y,t))$ is the vector forcing field, Re is the Reynolds number describing the properties of the fluid flow, g is the prescribed velocity on the boundary of the domain, and $v^0(x,y,t) = \left(v_1^0(x,y,t), v_2^0(x,y,t)\right)$ and $p^0(x,y,t)$ are the initial velocity and pressure configurations. Following the ideas presented by Petar Minev and Jean Luc-Guermound [18], we

replace this original problem with the singularly perturbed problem

$$\begin{cases} \partial_t v_\varepsilon + v_\varepsilon \cdot \nabla v_\varepsilon - \dfrac{1}{Re}\Delta v_\varepsilon + \nabla p_\varepsilon = f & \text{in } \Omega \times I \\ \varepsilon A p_\varepsilon + \nabla \cdot v_\varepsilon = 0 & \text{in } \Omega \times I \\ \varepsilon \partial_t p_\varepsilon = p_\varepsilon - \chi \dfrac{1}{Re} \nabla \cdot v_\varepsilon & \text{in } \Omega \times I \\ v_\varepsilon|_{\partial\Omega} = g & \text{in } \partial\Omega \times I \\ v_\varepsilon(\cdot, 0) = v_0 & \text{in } \Omega \\ p_\varepsilon(\cdot, 0) = p_0 & \text{in } \Omega \end{cases}$$

where ε, A, χ are model parameters.

This perturbed problem allows introducing stable direction splitting scheme, with the following sub-steps:

- pressure predictor: $\tilde{p}^{n+\frac{1}{2}} = p^{n-\frac{1}{2}} + \phi^{n-\frac{1}{2}}$
- velocity update:

$$\left(1 - \frac{\tau \partial_x^2}{2Re}\right) v^{n+\frac{1}{2}} = \left(1 + \frac{\tau \partial_y^2}{2Re} - v^n \cdot \nabla\right) v^n - \frac{\tau}{2} \nabla \tilde{p}^{n+\frac{1}{2}} + \frac{\tau}{2} f^{n+\frac{1}{2}}$$

$$\left(1 - \frac{\tau \partial_y^2}{2Re}\right) v^{n+1} = \left(1 + \frac{\tau \partial_y^2}{2Re}\right) v^{n+\frac{1}{2}} - \frac{\tau}{2} \nabla \tilde{p}^{n+\frac{1}{2}} + \frac{\tau}{2} f^{n+\frac{1}{2}}$$

- pressure update

$$\begin{aligned} (1 - \partial_x^2)\psi &= -\frac{1}{\tau} \nabla \cdot v^{n+\frac{1}{2}} \\ (1 - \partial_y^2)\phi^{n+\frac{1}{2}} &= \psi \end{aligned}$$

$$p^{n+\frac{1}{2}} = p^{n-\frac{1}{2}} + \phi^{n+\frac{1}{2}} - \chi \frac{1}{Re} \nabla \cdot \frac{1}{2}\left(v^{n+1} + v^n\right)$$

These sub-steps are implemented in our `step` method

```
void step(int iter, double t) override {
    // first step - simple summation
    compute_pressure_predictor();

    update_velocity_igrm(iter, t);
    update_pressure_igrm();
}
```

The first comes the implementation of the velocity update

```
void update_velocity_igrm(int /*i*/, double t) {
    // auxiliary definitions
    auto dt = steps.dt;
    auto f = [&](point_type x, double s) { return problem.forcing(x, s); };
    auto F = [&](double s) { return [&, s](point_type x) { return f(x, s); }; };
    auto Re = problem.Re;
    auto conv = problem.navier_stokes ? dt / 2 : 0; // if false, nonlinear term
        omitted

    auto dU1 = trial.U1x.dofs() * trial.U1y.dofs();
    auto dU2 = trial.U2x.dofs() * trial.U2y.dofs();
    auto dim_trial = dU1 + dU2;

    auto DU1 = test.U1x.dofs() * test.U1y.dofs();
    auto DU2 = test.U2x.dofs() * test.U2y.dofs();
    auto dim_test = DU1 + DU2;

    // Substep 1
    // buffer for the whole right hand side
    std::vector<double> rhs(dim_test + dim_trial);
    // views on parts of the right-hand side
    vector_view rhs_vx1{rhs.data(), {test.U1x.dofs(), test.U1y.dofs()}};
    vector_view rhs_vy1{rhs.data() + DU1, {test.U2x.dofs(), test.U2y.dofs()}};
    vector_view vx1{rhs.data() + dim_test, {trial.U1x.dofs(), trial.U1y.dofs()}};
```

```
    vector_view vy1{vx1.data() + dU1, {trial.U2x.dofs(), trial.U2y.dofs()}};

    compute_rhs(rhs_vx1, rhs_vy1,    // right-hand sides
                vx, vy,              // v0 = v from previous step
                vx, vy,              // v from previous substep
                p_star,              // pressure predictor
                F(t + dt / 2),       // forcing
                0, 0,                // v0 coeffs
                0, -dt / (2 * Re),   // v  coeffs
                -conv,               // u * \/u coeff (N-S term)
                dt / 2,              // pressure coeff
                dt / 2               // forcing coeff
    );

    apply_velocity_bc(vx1, trial.U1x, trial.U1y, t + dt, 0);
    apply_velocity_bc(vy1, trial.U2x, trial.U2y, t + dt, 1);

    mumps::problem problem_vx1{rhs};
    assemble_matrix_velocity(problem_vx1, dt / (2 * Re), 0);
    solver.solve(problem_vx1);

    // update the state
    vx_prev = vx;
    vy_prev = vy;
    // (vx, vy) := (vx2, vy2);
    for (auto i : dofs(trial.U1x, trial.U1y)) {
        vx(i[0], i[1]) = vx2(i[0], i[1]);
    }
    for (auto i : dofs(trial.U2x, trial.U2y)) {
        vy(i[0], i[1]) = vy2(i[0], i[1]);
    }
}
```

```
    // Substep 2
    std::vector<double> rhs2(dim_test + dim_trial);
    vector_view rhs_vx2{rhs2.data(), {test.U1x.dofs(), test.U1y.dofs()}};
    vector_view rhs_vy2{rhs2.data() + DU1, {test.U2x.dofs(), test.U2y.dofs()}};
    vector_view vx2{rhs2.data() + dim_test, {trial.U1x.dofs(), trial.U1y.dofs()}};
    vector_view vy2{vx2.data() + dU1, {trial.U2x.dofs(), trial.U2y.dofs()}};

    compute_rhs(rhs_vx2, rhs_vy2,    // right-hand sides
                vx, vy,              // v0 = v from previous step
                vx1, vy1,            // v from previous substep
                p_star,              // pressure predictor
                F(t + dt / 2),       // forcing
                0, 0,                // v0 coeffs
                -dt / (2 * Re), 0,   // v  coeffs
                -conv,               // u * \/u coeff (N-S term)
                dt / 2,              // pressure coeff
                dt / 2               // forcing coeff
    );

    apply_velocity_bc(vx2, trial.U1x, trial.U1y, t + dt, 0);
    apply_velocity_bc(vy2, trial.U2x, trial.U2y, t + dt, 1);

    mumps::problem problem_vx2{rhs2};
    assemble_matrix_velocity(problem_vx2, 0, dt / (2 * Re));
    solver.solve(problem_vx2);
```

For each of the sub-steps we need to generate the right-hand side. This is done with the `compute_rhs` method

```
// Compute RHS as
// (v, w) + ax (dv0/dx, dw/dx) + ay (dv0/dy, dw/dy) +
//          bx (dv/dx, dw/dx) + by (dv/dy, dw/dy) +
//          conv * u * \/u
//          c (\/p, w) + d  (f, w)
template <typename RHS, typename S1, typename S2, typename S3, typename Fun>
void compute_rhs(RHS& rhsx, RHS& rhsy, const S1& vx0, const S1& vy0, const S2& vx,
    const S2& vy,
                     const S3& p, Fun&& forcing, double ax, double ay, double bx,
                         double by,
                     double conv, double c, double d) const {
    using shape = std::array<int, 2>;
```

```
    auto u1_shape = shape{test.U1x.basis.dofs_per_element(), test.U1y.basis.
        dofs_per_element()};
    auto u2_shape = shape{test.U2x.basis.dofs_per_element(), test.U2y.basis.
        dofs_per_element()};

    // parallel loop over elements
    executor.for_each(elements(trial.Px, trial.Py), [&](index_type e) {
        auto vx_loc = vector_type{u1_shape};
        auto vy_loc = vector_type{u2_shape};

        double J = jacobian(e);
        for (auto q : quad_points(trial.Px, trial.Py)) {
            double W = weight(q);
            auto x = point(e, q);
            auto F = forcing(x);

            // compute values and gradients of two previous stated
            // (previous time step and possibly previous half-step)
            value_type vvx0 = eval(vx0, e, q, trial.U1x, trial.U1y);
            value_type vvy0 = eval(vy0, e, q, trial.U2x, trial.U2y);
            value_type vvx = eval(vx, e, q, trial.U1x, trial.U1y);
            value_type vvy = eval(vy, e, q, trial.U2x, trial.U2y);
            value_type pp = eval(p, e, q, trial.Px, trial.Py);

            for (auto a : dofs_on_element(e, test.U1x, test.U1y)) {
                auto aa = dof_global_to_local(e, a, test.U1x, test.U1y);
                value_type v = eval_basis(e, q, a, test.U1x, test.U1y);

                double val = vvx.val * v.val
                    //
                             + ax * vvx0.dx * v.dx
                               //
                             + ay * vvx0.dy * v.dy
                               //
                             + bx * vvx.dx * v.dx
                               //
                             + by * vvx.dy * v.dy
                               //
                             + c * pp.val * v.dx
                               //
                             + conv * (vvx.val * vvx.dx + vvy.val * vvx.dy) * v.val
                               //
                             + d * F[0] * v.val;
                vx_loc(aa[0], aa[1]) += val * W * J;
            }
            for (auto a : dofs_on_element(e, test.U2x, test.U2y)) {
                auto aa = dof_global_to_local(e, a, test.U2x, test.U2y);
                value_type v = eval_basis(e, q, a, test.U2x, test.U2y);

                double val = ... // same for the y velocity components
                vy_loc(aa[0], aa[1]) += val * W * J;
            }

        }
        // update of the global RHS vector
        executor.synchronized([&]() {
            update_global_rhs(rhsx, vx_loc, e, test.U1x, test.U1y);
            update_global_rhs(rhsy, vy_loc, e, test.U2x, test.U2y);
        });
    });
}
```

The next the pressure update sub-step is implemented in `update_pressure_igrm`

```
void update_pressure_igrm() {
    auto dim_trial = trial.Px.dofs() * trial.Py.dofs();
    auto dim_test = test.Px.dofs() * test.Py.dofs();

    // Step 1 - computing psi
    std::vector<double> rhs(dim_test + dim_trial);
    vector_view rhs_p1{rhs.data(),
                       {test.Px.dofs(), test.Py.dofs()}};
    vector_view p1{rhs.data() + dim_test,
```

```
                    {trial.Px.dofs(), trial.Py.dofs()}};

    compute_rhs_pressure_1(rhs_p1, vx, vy,
                           test.Px, test.Py, steps.dt);
    mumps::problem problem_px{rhs};
    assemble_matrix_pressure(problem_px, 1, 0);
    solver.solve(problem_px);

    // Step 2 - computing phi^{n+1}
    std::vector<double> rhs2(dim_test + dim_trial);
    vector_view rhs_p2{rhs2.data(),
                       {test.Px.dofs(), test.Py.dofs()}};
    vector_view p2{rhs2.data() + dim_test,
                   {trial.Px.dofs(), trial.Py.dofs()}};

    compute_rhs_pressure_2(rhs_p2, p1, test.Px, test.Py);
    mumps::problem problem_py{rhs2};
    assemble_matrix_pressure(problem_py, 0, 1);
    solver.solve(problem_py);

    // Put solution into phi
    for (auto i : dofs(trial.Px, trial.Py)) {
        phi(i[0], i[1]) = p2(i[0], i[1]);
    }

    // Update pressure
    apply_pressure_corrector();
}
```

We also generate the right-hand sides for this sub-step in `compute_rhs_pressure_1` and `compute_rhs_pressure_2`.

```
template <typename RHS, typename Sol>
void compute_rhs_pressure_1(RHS& rhs, const Sol& vx, const Sol& vy, const dimension&
     Vx,
                            const dimension& Vy, double dt) const {
    using shape = std::array<int, 2>;
    auto p_shape = shape{Vx.basis.dofs_per_element(), Vy.basis.dofs_per_element()};

    // paralell loop over elements, quadrature points and basis functions
    executor.for_each(elements(trial.Px, trial.Py), [&](index_type e) {
        auto loc = vector_type{p_shape};

        double J = jacobian(e);
        for (auto q : quad_points(trial.Px, trial.Py)) {
            double W = weight(q);
            value_type vvx = eval(vx, e, q, trial.U1x, trial.U1y);
            value_type vvy = eval(vy, e, q, trial.U2x, trial.U2y);

            for (auto a : dofs_on_element(e, Vx, Vy)) {
                auto aa = dof_global_to_local(e, a, Vx, Vy);
                value_type v = eval_basis(e, q, a, Vx, Vy);

                double val = -1 / dt * (vvx.dx + vvy.dy) * v.val;
                loc(aa[0], aa[1]) += val * W * J;
            }
        }
        executor.synchronized([&]() { update_global_rhs(rhs, loc, e, Vx, Vy); });
    });
}
```

The implementation of the `compute_rhs_pressure_2` is similar, except for `double val = pp.val * v.val;`. The Gram matrix A and the problem matrix B are generated in `assembly_matrix_pressure` routine.

```
void assemble_matrix_pressure(mumps::problem& problem, double cx, double cy) const {
    // Here we build a matrix used to calculate psi and phi^{n+1/2}
    // Stabilization using iGRM leads to the following structure:
    //   G    B
    //   B^T  0

    // Assembling the Gram matrix - G
    for (auto i : dofs(test.Px, test.Py)) {
        for (auto j : overlapping_dofs(i, test.Px, test.Py)) {
            int ii = linear_index(i, test.Px, test.Py) + 1;
            int jj = linear_index(j, test.Px, test.Py) + 1;
```

```
785
786             if (!is_pressure_fixed(i) && !is_pressure_fixed(j)) {
787                 auto eval = [&](auto form) {
788                     return integrate(i, j, test.Px, test.Py, test.Px, test.Py, form)
789                         ;
790                 };
791                 double val = eval([](auto w, auto p) { return w.val * p.val; });
792                 problem.add(ii, jj, val);
793             }
794         }
795     }

796     // Assembling B (and B^T at the same time)
797
798     // This auxiliary function makes sure the entries of B and B^T
799     // blocks are placed correctly inside the full matrix
800     auto DP = test.Px.dofs() * test.Py.dofs();
801     auto put = [&](int i, int j, int si, int sj, double val) {
802         int ii = i + si;
803         int jj = j + sj;
804         problem.add(ii, DP + jj, val);
805         problem.add(DP + jj, ii, val);
806     };
807
808     for (auto i : dofs(test.Px, test.Py)) {
809         for (auto j : overlapping_dofs(i, test.Px, test.Py, trial.Px, trial.Py)) {
810             if (!overlap(i, test.Px, test.Py, j, trial.Px, trial.Py))
811                 continue;
812
813             int ii = linear_index(i, test.Px, test.Py) + 1;
814             int jj = linear_index(j, trial.Px, trial.Py) + 1;
815             auto eval = [&](auto form) {
816                 return integrate(i, j, test.Px, test.Py, trial.Px, trial.Py, form);
817             };
818
819             double value = eval([cx, cy](auto u, auto v) {
820                 return u.val * v.val + cx * u.dx * v.dx + cy * u.dy * v.dy;
821             });
822             put(ii, jj, 0, 0, value);
823         }
824     }
825 }

826 void apply_pressure_corrector() {
827     vector_type rhs{{trial.Px.dofs(), trial.Py.dofs()}};
828
829     double chi = 0;
830     compute_rhs_pressure_update(rhs, chi);
831     mumps::problem problem(rhs.data(), rhs.size());
832     assemble_matrix(problem, 0, 0, false, false, trial.Px, trial.Py);
833     solver.solve(problem);
834
835     p = rhs;
836 }

837 template <typename RHS>
838 void compute_rhs_pressure_update(RHS& rhs, double chi) const {
839     auto Re = problem.Re;
840     using shape = std::array<int, 2>;
841     auto p_shape = shape{trial.Px.basis.dofs_per_element(), trial.Py.basis.
842         dofs_per_element()};
843
844     // parallel loop over elements, quadratures points and basis functions
845     executor.for_each(elements(trial.Px, trial.Py), [&](index_type e) {
846         auto loc = vector_type{p_shape};
847
848         double J = jacobian(e);
849         for (auto q : quad_points(trial.Px, trial.Py)) {
850             double W = weight(q);
851             value_type pp = eval(p, e, q, trial.Px, trial.Py);
852             value_type pphi = eval(phi, e, q, trial.Px, trial.Py);
853             value_type vvx = eval(vx, e, q, trial.U1x, trial.U1y);
854             value_type vvy = eval(vy, e, q, trial.U2x, trial.U2y);
```

```
            value_type vvx_prev = eval(vx_prev, e, q, trial.U1x, trial.U1y);
            value_type vvy_prev = eval(vy_prev, e, q, trial.U2x, trial.U2y);

            for (auto a : dofs_on_element(e, trial.Px, trial.Py)) {
                auto aa = dof_global_to_local(e, a, trial.Px, trial.Py);
                value_type v = eval_basis(e, q, a, trial.Px, trial.Py);

                double vdiv = vvx.dx + vvy.dy;
                double vdiv_prev = vvx_prev.dx + vvy_prev.dy;
                double val = (pp.val + pphi.val - 0.5 * chi / Re * (vdiv + vdiv_prev
                    )) * v.val;
                loc(aa[0], aa[1]) += val * W * J;
            }
        }
        executor.synchronized([&]() { update_global_rhs(rhs, loc, e, trial.Px, trial
            .Py); });
    });
}
```

```
void assemble_matrix(mumps::problem& problem, double cx, double cy, bool bcx, bool
    bcy,
                     const dimension& Ux, const dimension& Uy) const {
    // loop over pairs of overlapping DoFs
    for (auto i : dofs(Ux, Uy)) {
        for (auto j : overlapping_dofs(i, Ux, Uy)) {
            int ii = linear_index(i, Ux, Uy) + 1;
            int jj = linear_index(j, Ux, Uy) + 1;

            bool at_bdx = is_boundary(i[0], Ux) || is_boundary(j[0], Ux);
            bool at_bdy = is_boundary(i[1], Uy) || is_boundary(j[1], Uy);
            bool fixed = (at_bdx && bcx) || (at_bdy && bcy);

            if (!fixed) {
                auto form = [cx, cy](auto u, auto v) {
                    return u.val * v.val + cx * u.dx * v.dx + cy * u.dy * v.dy;
                };
                auto product = integrate(i, j, Ux, Uy, Ux, Uy, form);
                problem.add(ii, jj, product);
            }
        }
    }
    // account for boundary conditions if necessary
    for_boundary_dofs(Ux, Uy, [&](index_type dof) {
        int i = linear_index(dof, Ux, Uy) + 1;
        bool at_bdx = is_boundary(dof[0], Ux) || is_boundary(dof[0], Ux);
        bool at_bdy = is_boundary(dof[1], Uy) || is_boundary(dof[1], Uy);
        bool fixed = (at_bdx && bcx) || (at_bdy && bcy);
        if (fixed) { problem.add(i, i, 1); }
    });
}
```

5.3 Discontinuous Galerkin method

Similarly to the Galerkin method enhanced with the residual minimization, we also introduce the Discontinuous Galerkin method, and we enhance with residual minimization scheme. First, we illustrate the DG method on the example of the Poisson equation with Dirichlet boundary data. We seek the scalar field $[0,1]^2 = \Omega \ni (x,y) \rightarrow u(x,y) \in \mathscr{R}$ such that

$$\begin{cases} -\Delta u = f \\ u|_{\partial\Omega} = g \end{cases} \tag{26}$$

with the Dirichlet boundary condition prescribed by $\partial\Omega \ni (x,y) \rightarrow u(x,y) \in \mathscr{R}$. Our DG formulation is based on standard Symmetric Interior Penalty (SIP) method:

$$a_h^{\text{sip}}(u_h, v_h) = l_h(v_h) \tag{27}$$

where

$$
\begin{aligned}
a_h^{\text{sip}}(u_h, v_h) &= (\nabla_h u_h, \nabla_h v_h) \\
&\quad - \sum_{F \in \mathscr{F}_h} \int_F \left(\{\nabla_h u_h\} \cdot n_F [v_h] + \{\nabla_h v_h\} \cdot n_F [u_h] \right) d\sigma \\
&\quad + \sum_{F \in \mathscr{F}_h} \frac{\eta}{h_F} \int_F [u_h][v_h] \, d\sigma \\
l_h(v_h) &= (f, v_h) + \sum_{F \in \mathscr{F}_h^\partial} \int_F \left(-\nabla g \cdot n_F v_h + \frac{\eta}{h_F} g v_h \right) d\sigma
\end{aligned}
\tag{28}
$$

Here $[\psi(x)] = \psi|_{T_1}(x) - \psi|_{T_2}(x)$ denotes the jump acres the shared edge between elements T_1 and T_2, and $\{\psi\} = \frac{\psi|_{T_1}(x) + \psi|_{T_2}(x)}{2}$ defines an average. The n_F denotes the versor normal to the element edge. The setup of the DG formulation in the code is done in the following way. We define the number of elements in `elems`. We introduce equally spaced one-dimensional grids along x and y axis span between 0 and 1. We build regular 2d mesh from these grids. We define B-splines of order p and continuity c. We define the quadrature to integrate the B-splines exactly.

```
// Define the XY grid
int elems = 128;
auto xs = ads::evenly_spaced(0.0, 1.0, elems);
auto ys = ads::evenly_spaced(0.0, 1.0, elems);
auto mesh = ads::regular_mesh{xs, ys};

// p - spline order, c - spline continuity
auto bx = ads::make_bspline_basis(xs, p, c);
auto by = ads::make_bspline_basis(ys, p, c);

// Create representations of the discrete space
auto space = ads::space{&mesh, bx, by};

// Use quadrature with p+1 points to ensure accuracy
auto quad = ads::quadrature{&mesh, p + 1};
```

We will use the MUMPS solver [14, 15] to solve the DG problem.

```
// Allocate space for matrix and the right-hand side
auto F = std::vector<double>(n);
auto problem = ads::mumps::problem{F.data(), n};

// These functions instruct the integration code
// where to put the computed values
auto out = [&problem](int row, int col, double val) {
    if (val != 0) {
        problem.add(row + 1, col + 1, val);
    }
};
auto rhs = [&F](int J, double val) { F[J] += val; };

// We are using MUMPS as our solver
auto solver = ads::mumps::solver{};
```

We assembly the system, component by component, following the formula 28.

```
// Assemble the first part of a^sip - elementwise gradient product
assemble(space, quad, out, [](auto u, auto v, auto /*x*/) {
    return dot(grad(u), grad(v));
});

// Assemble the second part - edge integrals
auto form = [eta](auto u, auto v, auto /*x*/, const auto& edge) {
    const auto& n = edge.normal;
    const auto h = length(edge.span);
    return - dot(grad(avg(v)), n) * jump(u).val
           - dot(grad(avg(u)), n) * jump(v).val
           + eta / h * jump(u).val * jump(v).val;
};
assemble_facets(mesh.facets(), space, quad, out, form);
```

The right-hand side for the DG code is formulated in `assemble_rhs` routine,

```
// As for the matrix - assemble first part of the linear form
// 'poisson' is an object representing the problem data
assemble_rhs(space, quad, rhs, [&poisson](auto v, auto x) {
    return v.val * poisson.f(x);  // f - part of problem data
});

// Second part - edge integrals
auto bd_form = [eta, &poisson](auto v, auto x, const auto& edge) {
    const auto& n = edge.normal;
    const auto h = length(edge.span);
    const auto g = poisson.g(x);  // g - part of problem data
    return - dot(grad(v), n) * g
           + eta/h * g * v.val;
};
// Here we only need the boundary edges, no the full skeleton
assemble_rhs(mesh.boundary_facets(), space, quad, rhs, bd_form);
```

The solution process and the postprocessing are invoked from the `main` routine in the following way

```
// Solve the assembled linear system
solver.solve(problem);

// Function object representing the solution
auto u = ads::bspline_function(&space, F.data());

// Compute error in L2 norm (assuming known exact solution)
auto err = error(mesh, quad, L2{}, u, poisson.u());
fmt::print("error = {:.6}\n", err);

// Output
save_to_file("result.data", u);
```

5.4 Discontinuous Galerkin with residual minimization for Stokes problem

To illustrate the stabilization with the DG mixed with the residual minimization method, we focus now on the Stokes equation. We seek the vector velocity field $[0,1]^3 = \Omega \ni (x,y,z) \to (u_1(x,y,z), u_2(x,y,z)) \in \mathscr{R}^3$ and the scalar pressure field $[0,1]^3 = \Omega \ni (x,y,z) \to p(x,y,z) \in \mathscr{R}$ such that

$$\begin{cases} -\Delta u + \nabla p = f \\ \nabla \cdot u = 0 \\ u|_{\partial\Omega} = 0 \end{cases} \tag{29}$$

First, we write the standard DG formulation that will stand for the problem matrix B in the residual minimization setup

$$\begin{cases} a_h(u_h^{\mathrm{DG}}, v_h) + b_h(v_h, p_h^{\mathrm{DG}}) = (f, v_h) \\ -b_h(u_h^{\mathrm{DG}}, q_h) + s_h(p_h^{\mathrm{DG}}, q_h) = 0 \end{cases} \tag{30}$$

where

$$\begin{aligned} a_h(w_h, v_h) &= (\nabla_h u_h, \nabla_h v_h) + \sum_{F \in \mathscr{F}_h} \frac{\eta}{h_F} \int_F [u_h][v_h]\, d\sigma \\ &\quad - \sum_{F \in \mathscr{F}_h} \int_F \left(\{\nabla_h u_h\} \cdot n_F [v_h] + \{\nabla_h v_h\} \cdot n_F [u_h] \right) d\sigma \\ b_h(v_h, q_h) &= -\int_\Omega q_h \nabla \cdot v_h \, dx + \sum_{F \in \mathscr{F}_h^0} \int_F [v_h] \cdot n_F \{q_h\}\, d\sigma \\ s_h(p_h, q_h) &= \sum_{F \in \mathscr{F}_h^0} h_F \int_F [p_h][q_h]\, d\sigma \end{aligned} \tag{31}$$

Second, we denote the problem (31) in the following way: Find $\varphi \in V_h$ such that

$$C_h(\varphi_h, \psi_h) = (f, \psi_h) \quad \forall \psi_h \in V_h \tag{32}$$

and we construct the residual minimization stabilization on top of (31).

$$\varphi_h = argmin_{w_h \in U_h} \| C_h(w_h, *) - (f, *) \|_{V_h^*} \tag{33}$$

Namely, we solve

$$\begin{aligned}(r_h,\psi_h)_{V_h}+C_h(\varphi_h,\psi_h)=(f,\psi_h)_\Omega \quad \forall \psi_h \in V_h\\ C_h(w_h,r_h)=0 \quad \forall w_h \in U_h\end{aligned} \tag{34}$$

In the residual minimization method, following [11], we introduce the following norm for building the Gram matrix

$$\begin{aligned}|||(v_h,q_h)|||^2_{Stokes}=\sum_{i=1,\ldots,d}\left(\sum_{K\in T_h}\|\nabla v_{h,i}\|^2_{L^2(K)}+\sum_{F\in F_h}\frac{1}{h_F}\|[v_{h,i}]\|^2_{L^2(F)}\right)+\\ \|q_h\|^2_{L^2(\Omega)}+\sum_{F\in F_h^{internal}}h_F\|[q_h]\|^2_{L^2(F)}\end{aligned} \tag{35}$$

The problem setup defines three one-dimensional grids with elements spanning between 0 and 1. It builds regular 3d mesh using these one-dimensional grids. It also constructs quadrature to integrate trial B-splines of order *p_trial*, and test B-splines of order *p_test*. It builds trial and test B-splines with given order and continuity along *x*, *y*, and *z* axes. It employs them for approximation of the velocity and pressure.

```
// Mesh and quadrature rules
auto xs = ads::evenly_spaced(0.0, 1.0, elems);
auto ys = ads::evenly_spaced(0.0, 1.0, elems);
auto zs = ads::evenly_spaced(0.0, 1.0, elems);
auto mesh = ads::regular_mesh3{xs, ys, zs};
auto quad = ads::quadrature3{&mesh, std::max(p_test, p_trial) + 1};

// Test space splines
auto Bx = ads::make_bspline_basis(xs, p_test, c_test);
auto By = ads::make_bspline_basis(ys, p_test, c_test);
auto Bz = ads::make_bspline_basis(zs, p_test, c_test);

// Space factory ensures correct global DoF numbering
auto tests = space_factory{};
auto Wx = tests.next<ads::space3>(&mesh, Bx, By, Bz);
auto Wy = tests.next<ads::space3>(&mesh, Bx, By, Bz);
auto Wz = tests.next<ads::space3>(&mesh, Bx, By, Bz);
auto Q = tests.next<ads::space3>(&mesh, Bx, By, Bz);
auto N = Wx.dof_count() + Wy.dof_count() + Wz.dof_count()
       + Q.dof_count();
```

```
// Trial space splines
auto bx = ads::make_bspline_basis(xs, p_trial, c_trial);
auto by = ads::make_bspline_basis(ys, p_trial, c_trial);
auto bz = ads::make_bspline_basis(zs, p_trial, c_trial);

// Trial space components
auto trials = space_factory{};
auto Vx = trials.next<ads::space3>(&mesh, bx, by, bz);
auto Vy = trials.next<ads::space3>(&mesh, bx, by, bz);
auto Vz = trials.next<ads::space3>(&mesh, bx, by, bz);
auto P = trials.next<ads::space3>(&mesh, bx, by, bz);
auto n = Vx.dof_count() + Vy.dof_count() + Vz.dof_count()
       + P.dof_count();
```

We will employ the MUMPS solver in this example. We build the Gram matrix and the problem matrix to be solved.

```
auto F = std::vector<double>(N + n);
auto problem = ads::mumps::problem{F.data(), N + n};
// G - Gram matrix, B - bilinear form matrix
// Full matrix structure:
//   G    B
//   B^T  0
auto G = [&problem](int row, int col, double val) {
    if (val != 0) {
        problem.add(row + 1, col + 1, val);
    }
};
auto B = [&problem, N](int row, int col, double val) {
    if (val != 0) {
        problem.add(row + 1, N + col + 1, val);
```

```
            problem.add(N + col + 1, row + 1, val);
        }
};
auto rhs = [&F](int row, double val) { F[row] += val; };
```

The Gram matrix and the problem matrices are assembled step by step according to formula 34 and 35

```
// Compute volume integrals of the test space product (Gram matrix)
assemble(Wx, quad, G, [](auto ux, auto vx, auto /*x*/) { return dot(grad(ux), grad(
    vx)); });
assemble(Wy, quad, G, [](auto uy, auto vy, auto /*x*/) { return dot(grad(uy), grad(
    vy)); });
assemble(Wz, quad, G, [](auto uz, auto vz, auto /*x*/) { return dot(grad(uz), grad(
    vz)); });
assemble(Q,  quad, G, [](auto p,  auto q,  auto /*x*/) { return p.val * q.val;
              });

// Compute volume integrals of the problem bilinear form
assemble(Vx, Wx, quad, B, [](auto ux, auto vx, auto /*x*/) { return dot(grad(ux),
    grad(vx)); });
assemble(Vy, Wy, quad, B, [](auto uy, auto vy, auto /*x*/) { return dot(grad(uy),
    grad(vy)); });
assemble(Vz, Wz, quad, B, [](auto uz, auto vz, auto /*x*/) { return dot(grad(uz),
    grad(vz)); });
assemble(P,  Wx, quad, B, [](auto p,  auto vx, auto /*x*/) { return - p.val * vx.dx;
              });
assemble(P,  Wy, quad, B, [](auto p,  auto vy, auto /*x*/) { return - p.val * vy.dy;
              });
assemble(P,  Wz, quad, B, [](auto p,  auto vz, auto /*x*/) { return - p.val * vz.dz;
              });
assemble(Vx,  Q, quad, B, [](auto ux, auto  q, auto /*x*/) { return   ux.dx * q.val;
              });
assemble(Vy,  Q, quad, B, [](auto uy, auto  q, auto /*x*/) { return   uy.dy * q.val;
              });
assemble(Vz,  Q, quad, B, [](auto uz, auto  q, auto /*x*/) { return   uz.dz * q.val;
              });
```

```
// Compute edge integrals of the test space product (Gram matrix)
assemble_facets(mesh.facets(), Wx, quad, G, [](auto ux, auto vx, auto /*x*/, const
    auto& face) {
    const auto  h = face.diameter;
    return 1/h * jump(ux).val * jump(vx).val;
});
// ... same for Y and Z
assemble_facets(mesh.interior_facets(), Q, quad, G, [](auto p, auto q, auto /*x*/,
    const auto& face) {
    const auto  h = face.diameter;
    return h * jump(p).val * jump(q).val;
});

// Compute edge integrals of the problem bilinear form
assemble_facets(mesh.facets(), Vx, Wx, quad, B, [eta](auto ux, auto vx, auto /*x*/,
    const auto& face) {
    const auto& n = face.normal;
    const auto  h = face.diameter;
    return - dot(grad(avg(vx)), n) * jump(ux).val
           - dot(grad(avg(ux)), n) * jump(vx).val
           + eta/h * jump(ux).val * jump(vx).val;
});
// ... same for (Vy, Wy) and (Vz, Wz)
assemble_facets(mesh.facets(), P, Wx, quad, B, [](auto p, auto vx, auto /*x*/, const
     auto& face) {
    const auto& n = face.normal;
    const auto  v = ads::point3_t{jump(vx).val, 0, 0};
    return avg(p).val * dot(v, n);
});
// ... same for Wy and Wz
assemble_facets(mesh.facets(), Vx, Q, quad, B, [](auto ux, auto q, auto /*x*/, const
     auto& face) {
    const auto& n = face.normal;
    const auto  u = ads::point3_t{jump(ux).val, 0, 0};
    return - dot(u, n) * avg(q).val;
});
// ... same for Vy and Vz
```

We also assemble the right-hand side

```
// Volume integral
assemble_rhs(Wx, quad, rhs, [&stokes](auto vx, auto x) {
    return vx.val * stokes.fx(x); // fx - problem data
});
// ... same for Wy and Wz

// Boundary integral
assemble_rhs(mesh.boundary_facets(), Wx, quad, rhs,
             [eta,&stokes](auto vx, auto x, const auto& face) {
    const auto& n = face.normal;
    const auto  h = face.diameter;
    const auto  g = stokes.vx(x); // vx - problem data
    return - dot(grad(vx), n) * g
           + eta/h * g * vx.val;
});
// ... same for Wy and Wz
```

5.5 Stationary advection-diffusion stabilized with residual minimization using Conjugate Gradients solver

Finally, we focus on the stationary advection-diffusion problem with Dirichlet boundary conditions. We seek the scalar concentration field $[0,1]^2 = \Omega \ni (x,y) \rightarrow u(x,y) \in \mathscr{R}$ such that

$$\begin{cases} -\varepsilon \Delta u + \beta \cdot \nabla u = f & \text{in } \Omega \\ v|_{\partial\Omega} = g & \text{in } \partial\Omega \end{cases} \tag{36}$$

We derive a weak formulation with weak enforcement of the boundary conditions using Nitsche's method

$$\begin{aligned} b(u_h, v_h) &= \varepsilon\,(\nabla u_h, \nabla v_h) + (\beta \cdot \nabla u_h, v_h) \\ &\quad - \langle \varepsilon \nabla u_h \cdot \hat{\mathbf{n}}, v_h \rangle - \langle u_h, \varepsilon \nabla v_h \cdot \hat{\mathbf{n}} \rangle \\ &\quad + \langle u_h, \beta \cdot \hat{\mathbf{n}} v_h \rangle - \langle \gamma_h u_h, v_h \rangle \\ l(v_h) &= (f, v_h) - \langle g, \varepsilon \nabla v_h \cdot \hat{\mathbf{n}} \rangle \\ &\quad + \langle g, \beta \cdot \hat{\mathbf{n}} v_h \rangle - \langle \gamma_h g, v_h \rangle \end{aligned} \tag{37}$$

On top of the problem formulation 37, we derive the residual minimization stabilization with the Gram matrix corresponding to the weighted H^1 norm, namely $G = M + \eta K$ where M, K corresponds to the mass and stiffness matrix, and η is the parameter.

$$\begin{bmatrix} G & B \\ B^T & 0 \end{bmatrix} \begin{bmatrix} r \\ u \end{bmatrix} = \begin{bmatrix} F \\ 0 \end{bmatrix} \tag{38}$$

We employ an iterative procedure to solve the system 38. We replace the Gram matrix G by an easy-to-factorize approximation $\widetilde{G}$ that has a Kronecker product structure $\widetilde{G} = (K_x + \eta M_x) \otimes (K_y + \eta M_y)$, and it is inexpensive to solve and only introduces an error of order η^2. We use an iterative algorithm

$$\begin{bmatrix} r^{k+1} \\ u^{k+1} \end{bmatrix} = \begin{bmatrix} r^k \\ u^k \end{bmatrix} + \begin{bmatrix} d \\ c \end{bmatrix} \tag{39}$$

where

$$\begin{bmatrix} \widetilde{G} & B \\ B^T & 0 \end{bmatrix} \begin{bmatrix} d \\ c \end{bmatrix} = \begin{bmatrix} F - Gr^k - Bu^k \\ -B^T r^k \end{bmatrix} \tag{40}$$

which is solved as

$$B^T \widetilde{G}^{-1} B c = B^T \widetilde{G}^{-1} \left(F - (G - \widetilde{G}) r^k - B u^k \right) \tag{41}$$

using conjugate gradients solver. This time we setup the problem in the `step` routine, implementing the solver algorithm

```
void step(int /*iter*/, double /*t*/) override {
    // buffers for auxiliary vectors
    auto dd = vector_type{{Vx.dofs(), Vy.dofs()}};
    auto dc = vector_type{{Ux.dofs(), Uy.dofs()}};
    auto Bc = vector_type{{Vx.dofs(), Vy.dofs()}};
```

```
    int i = 0;
    for (; i < cfg.max_outer_iters; ++i) {
        // first we compute the RHS of the Shur complement system
        // dd = A~ \ (F + Kr - Bu)
        compute_dd(Vx, Vy, dd);
        zero_bc(dd, Vx, Vy);
        solve_A(dd);

        // dc = B' dd
        apply_Bt(dd, dc);
        zero_bc(dc, Ux, Uy);

        // solve for c (this also updates u)
        auto c = cfg.use_cg ? substep_CG(dc) : substep_mumps();

        // use computed c to update r
        // Bc = A~ \ B c
        apply_B(c, Bc);
        zero_bc(Bc, Vx, Vy);
        solve_A(Bc);

        // r + d = dd - A~ \ B c
        update_residual(dd, Bc);

        // check stopping condition
        auto dimU = Ux.dofs() * Uy.dofs();
        auto cc = norm(c, Ux, Uy) / dimU;

        if (cc < cfg.tol_outer) {
            ++i;
            break;
        }
    }
}
```

The conjugate gradient solver is implemented in `substep_CG` routine

```
vector_view substep_CG(const vector_type& dc) {
    vector_type u_prev = u;
    auto p = dc;
    auto q = dc;
    auto theta = vector_type{{Vx.dofs(), Vy.dofs()}};
    auto delta = theta;
    auto Mp = vector_type{{Ux.dofs(), Uy.dofs()}};

    for (int i = 0; i < cfg.max_inner_iters; ++i) {
        // theta = Bp
        apply_B(p, theta);
        zero_bc(theta, Vx, Vy);
        // delta = A~ \ theta
        delta = theta;
        solve_A(delta);
        // alpha = (p, q) / (p, Mp)
        double alpha = dot(p, q, Ux, Uy) / dot(theta, delta, Vx, Vy);
        // Mp = B' delta
        apply_Bt(delta, Mp);
        zero_bc(Mp, Ux, Uy);

        double qnorm2_prev = norm_sq(q, Ux, Uy);

        // u := u + alpha p
        // q := q - alpha Mp
        for (auto i : dofs(Ux, Uy)) {
            u(i[0], i[1]) += alpha * p(i[0], i[1]);
            q(i[0], i[1]) -= alpha * Mp(i[0], i[1]);
        }
        // beta = |q_prev|^2 / |q|^2
        double qnorm2 = norm_sq(q, Ux, Uy);
        double beta = qnorm2 / qnorm2_prev;
        // p := q + beta * p
        for (auto i : dofs(Ux, Uy)) {
            p(i[0], i[1]) = q(i[0], i[1]) + beta * p(i[0], i[1]);
        }
```

```
        // check for convergence
        auto dimU = Ux.dofs() * Uy.dofs();
        double residuum = std::sqrt(qnorm2) / dimU;
        if (residuum < cfg.tol_inner)
            break;
    }
    vector_view du{full_rhs.data(), {Ux.dofs(), Uy.dofs()}};
    for (auto i : dofs(Ux, Uy)) {
        du(i[0], i[1]) = u(i[0], i[1]) - u_prev(i[0], i[1]);
    }
    return du;
}
```

The matrices from the system of equations are formulated in the following routines

```
// Interior part of the problem bilinear form
double B(value_type u, value_type v, point_type x) const {
    auto diff = diffusion(x);
    return diff * grad_dot(u, v) + dot(beta(x), u) * v.val;
}

// Boundary part of the problem bilinear form
double bdB(value_type u, value_type v, point_type x, point_type n) const {
    return - epsilon * v.val * dot(u, n)      // <eps |/u*n, v> -- consistency
           - epsilon * u.val * dot(v, n)      // <u, eps |/v*n> -- symmetry
           - u.val * v.val * dot(beta(x), n) // <u, v beta*n>  -- symmetry
           - u.val * v.val * gamma;           // <u, gamma v>   -- penalty
}

// Boundary part of the problem RHS
double bdL(value_type v, point_type x, point_type n) const {
    return - epsilon * g(x) * dot(v, n)      // <g, eps |/v*n>
           - g(x) * v.val * dot(beta(x), n) // <g, v beta*n>
           - g(x) * v.val * gamma;           // <u, gamma v> -- penalty
}

// Scalar product for iGRM stabilization
double A(value_type u, value_type v) const {
    return u.val * v.val + h * h * grad_dot(u, v);
}
```

Finally, the right-hand side vector is defined in the following routines

```
void compute_dd(const dimension& Vx, const dimension& Vy, vector_type& dd) {
    zero(dd);
    // parallel loop over mesh elements
    executor.for_each(elements(Vx, Vy), [&](index_type e) {
        auto R = vector_type{{Vx.basis.dofs_per_element(),
                              Vy.basis.dofs_per_element()}};

        double J = jacobian(e);
        for (auto q : quad_points(Vx, Vy)) {
            double W = weight(q);
            double WJ = W * J;
            auto x = point(e, q);
            value_type uu = eval(u, e, q, Ux, Uy);
            // mixed xy derivative evaluation
            double rxy = eval_fun_dxy(r, e, q, Vx, Vy);

            for (auto a : dofs_on_element(e, Vx, Vy)) {
                auto aa = dof_global_to_local(e, a, Vx, Vy);
                value_type v = eval_basis(e, q, a, Vx, Vy);
                double vxy = eval_basis_dxy(e, q, a, Vx, Vy);
                double Lv = F(x) * v.val;

                // F + Kr - Bu
                double Kr = eta * eta * rxy * vxy;
                // B is the bilinear form of the problem
                double val = Lv + Kr - B(uu, v, x);
                R(aa[0], aa[1]) += val * WJ;
            }
        }
        executor.synchronized([&]() { update_global_rhs(dd, R, e, Vx, Vy); });
    });
```

```
    // Boundary terms of -Bu
    for (auto i : dofs(Vx, Vy)) {
        double val = 0;

        auto form = [&](auto v, auto u, auto x, auto n) {
            return this->bdB(u, v, x, n);
        };
        // loop over (subset) of boundary edges
        for_sides(~dirichlet, [&](auto side) {
            if (this->touches(i, side, Vx, Vy)) {
                val += integrate_boundary(side, i, Vx, Vy, u, Ux, Uy, form);
            }
        });
        dd(i[0], i[1]) -= val;
    }

    // Boundary terms of RHS
    for (auto i : dofs(Vx, Vy)) {
        double val = 0;

        auto form = [&](auto w, auto x, auto n) {
            return this->bdL(w, x, n);
        };
        // loop over (subset) of boundary edges
        for_sides(~dirichlet, [&](auto side) {
            if (this->touches(i, side, Vx, Vy)) {
                val += integrate_boundary(side, i, Vx, Vy, form);
            }
        });
        dd(i[0], i[1]) += val;
    }
}

template <typename U, typename Res>
void apply_B(const U& u, Res& result) {
    zero(result);
    executor.for_each(elements(Vx, Vy), [&](index_type e) {
        auto rhs = vector_type{{Vx.basis.dofs_per_element(), Vy.basis.
            dofs_per_element()}};
        double J = jacobian(e);
        for (auto q : quad_points(Vx, Vy)) {
            double W = weight(q);
            double WJ = W * J;
            auto x = point(e, q);
            value_type uu = eval(u, e, q, Ux, Uy);

            for (auto a : dofs_on_element(e, Vx, Vy)) {
                auto aa = dof_global_to_local(e, a, Vx, Vy);
                value_type v = eval_basis(e, q, a, Vx, Vy);
                double val = B(uu, v, x);
                rhs(aa[0], aa[1]) += val * WJ;
            }
        }
        executor.synchronized([&]() { update_global_rhs(result, rhs, e, Vx, Vy); });
    });
    // Boundary terms of Bu
    for (auto i : dofs(Vx, Vy)) {
        double val = 0;
        auto form = [&](auto v, auto u, auto x, auto n) { return this->bdB(u, v, x,
            n); };
        for_sides(~dirichlet, [&](auto side) {
            if (this->touches(i, side, Vx, Vy)) {
                val += integrate_boundary(side, i, Vx, Vy, u, Ux, Uy, form);
            }
        });
        result(i[0], i[1]) += val;
    }
} // apply_Bt - analogous
```

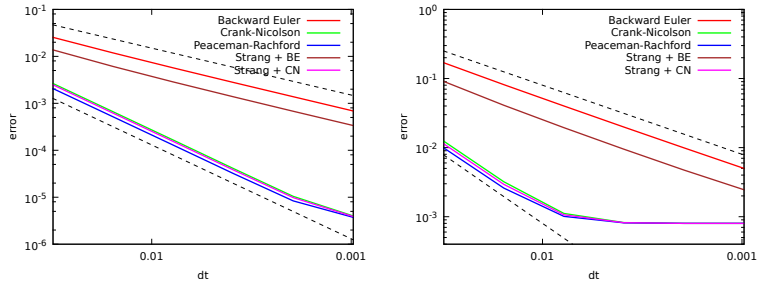


Figure 3: Horizontal axis denotes time step size. The vertical axis denotes the L2 and H1 norms of the solution for different time integration schemes on 32×32 mesh.

6 Numerical examples

The numerical examples section aims to illustrate the correctness of our code and show how it can be used to stabilize difficult computational problems. The first numerical result illustrates the application of different time-marching schemes for the non-stationary advection-dominated diffusion problem with a manufactured solution. The second numerical example concerns the stabilized simulations of the pollution propagation from a chimney with the Douglass-Gunn time integration scheme and residual minimization method. The third example concerns the cavity problem with Navier-Stokes equations. This example aims to show that the standard Galerkin formulation does not work, and thus, we can stabilize it by adding residual minimization on top of the time marching schemes. The fourth numerical example concerns the manufactured solution for the Stokes problem solved by using the Discontinuous Galerkin method with residual minimization. The goal of our simulation is to measure the numerical errors to show the high accuracy of our solutions. The fifth numerical example is the advection skew to the mesh, a stationary advection-dominated diffusion problem solved with residual minimization and conjugate gradient solver.

6.1 Manufactured solution for advection-dominated diffusion with different time marching schemes

We focus on time-dependent advection-diffusion problem

$$\frac{\partial u}{\partial t} - \varepsilon \nabla \cdot (\nabla u) + \beta \cdot \nabla u = f,$$

with $\varepsilon = 10^{-2}$, $\beta = (1,0)$, solved on $[0,1]^2$ domain. We set up the forcing $f(x,y,t)$ to deliver the manufactured solution $u(x,y,t) = \sin(\Pi x)\sin(\Pi y)\sin(\Pi t)$ for $t \in [0,2]$ We investigate the error for different time step sizes dt varying from 0.1 to 0.001. We investigate the Backward-Euler, Crank-Nicolson, Peaceman-Reachford, and Strang method with Backward-Euler and Strang method with Crank-Nicolson schemes. The results are summarized in Figure 3.

From the numerical experiments, we can conclude that Peacemen-Rachford, together with Crank-Nicolson and Strang methods coupled with Crank-Nicolson schemes, are of the second order with respect to time. This means that decreasing the time step one in order of magnitude results in a two-order increase in numerical accuracy.

6.2 Three-dimensional advection-diffusion simulation with Douglass-Gunn time marching scheme and residual minimization method

We describe the numerical simulation of three-dimensional model advection-diffusion problem over a 3D cube shape domain with dimensions $5000 \times 5000 \times 5000$ meters.

$$\frac{du}{dt} - \nabla \cdot (K \nabla u) + \beta \cdot \nabla u = f \tag{42}$$

In our equation we have

$$K = \begin{bmatrix} 50 & 0 & \\ 0 & 50 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}, \tag{43}$$

the β is given by

$$\beta = (\beta^x(t), \beta^y(t), \beta^z(t)) = (\cos a(t), \sin a(t), v(t)), \tag{44}$$

where

$$a(t) = \frac{\pi}{3}(\sin(s) + \frac{1}{2}\sin(2.3s)) + \frac{3}{8}\pi, \tag{45}$$

and

$$v(t) = \frac{1}{3}\sin(s), \tag{46}$$

with $s = \frac{t}{150}$. The source is given by

$$f(p) = (r-1)^2(r+1)^2, \tag{47}$$

where $r = \min(1, (|p-p_0|/25)^2)$, and p represents the distance from the source, and p_0 is the location of the source $p_0 = (3,3,2)$. The initial state is defined as the constant concentration of the order of 10^{-6} in the entire domain (numerical zero). The physical meaning of this setup is the following. We model the propagation of the pollutant generated by a chimney as modeled by the f function. The pollution is distributed by the wind as modeled by β function and by the diffusion phenomena described by the diffusion matrix K. For this three-dimensional problem, we need the Douglas-Gunn time integration scheme that is second-order accurate in time

$$\begin{cases} (1+\frac{\tau}{2}\mathscr{L}_1)u^{n+1/3} = \tau f^{n+1/2} + (1-\frac{\tau}{2}\mathscr{L}_1 - \tau\mathscr{L}_2 - \tau\mathscr{L}_3)u^n, \\ (1+\frac{\tau}{2}\mathscr{L}_2)u^{n+2/3} = u^{n+1/3} + \frac{\tau}{2}\mathscr{L}_2 u^n, \\ (1+\frac{\tau}{2}\mathscr{L}_3)u^{n+1} = u^{n+2/3} + \frac{\tau}{2}\mathscr{L}_3 u^n. \end{cases} \tag{48}$$

which translates into

$$\begin{cases} (u^{n+1/3}, v) + \frac{\tau}{2}\left(\alpha\frac{\partial u^{n+1/3}}{\partial x}, \frac{\partial v}{\partial x}\right) + \frac{\tau}{2}\left(\beta_x\frac{\partial u^{n+1/3}}{\partial x}, v\right) = \\ \quad (u^n, v) - \frac{\tau}{2}\left(\alpha\frac{\partial u^n}{\partial x}, \frac{\partial v}{\partial x}\right) - \frac{\tau}{2}\left(\beta_x\frac{\partial u^n}{\partial x}, v\right) - \tau\left(\alpha\frac{\partial u^n}{\partial y}, \frac{\partial v}{\partial y}\right) \\ \quad - \tau\left(\beta_y\frac{\partial u^n}{\partial y}, v\right) - \tau\left(\alpha\frac{\partial u^n}{\partial z}, \frac{\partial v}{\partial z}\right) - \tau\left(\beta_z\frac{\partial u^n}{\partial z}, v\right) + \tau(f^{n+1/2}, v), \\ (u^{n+2/3}, v) + \frac{\tau}{2}\left(\alpha\frac{\partial u^{n+2/3}}{\partial y}, \frac{\partial v}{\partial y}\right) + \frac{\tau}{2}\left(\beta_y\frac{\partial u^{n+2/3}}{\partial y}, v\right) = \\ \quad (u^{n+1/3}, v) + \frac{\tau}{2}\left(\alpha\frac{\partial u^n}{\partial y}, \frac{\partial v}{\partial y}\right) + \frac{\tau}{2}\left(\beta_y\frac{\partial u^n}{\partial y}, v\right), \\ (u^{n+1}, v) + \frac{\tau}{2}\left(\alpha\frac{\partial u^{n+1}}{\partial z}, \frac{\partial v}{\partial z}\right) + \frac{\tau}{2}\left(\beta_z\frac{\partial u^{n+1}}{\partial z}, v\right) = \\ \quad (u^{n+2/3}, v) + \frac{\tau}{2}\left(\alpha\frac{\partial u^n}{\partial z}, \frac{\partial v}{\partial z}\right) + \frac{\tau}{2}\left(\beta_z\frac{\partial u^n}{\partial z}, v\right), \end{cases}$$

where $(\cdot,\cdot)$ denotes the inner product of $L^2(\Omega)$. The weak form is the following

$$\begin{cases} \left[M^x + \frac{\tau}{2}(K^x + G^x)\right] \otimes M^y \otimes M^z u^{n+1/3} \\ = \left[M^x - \frac{\tau}{2}(K^x + G^x)\right] \otimes M^y \otimes M^z u^n \\ - \tau M^x \otimes (K^y + G^y) \otimes M^z u^n - \tau M^x \otimes M^y \otimes (K^z + G^z) u^n + \tau F^{n+1/2} \\ M^x \otimes \left[M^y + \frac{\tau}{2}(K^y + G^y)\right] \otimes M^z u^{n+2/3} \\ = M^x \otimes M^y \otimes M^z u^{n+1/3} + M^x \otimes \frac{\tau}{2}(K^y + G^y) \otimes M^z u^n, \\ M^x \otimes M^y \otimes \left[M^z + \frac{\tau}{2}(K^z + G^z)\right] u^{n+1} \\ = M^x \otimes M^y \otimes M^z u^{n+2/3} + M^x \otimes M^y \otimes \frac{\tau}{2}(K^z + G^z) u^n, \end{cases}$$

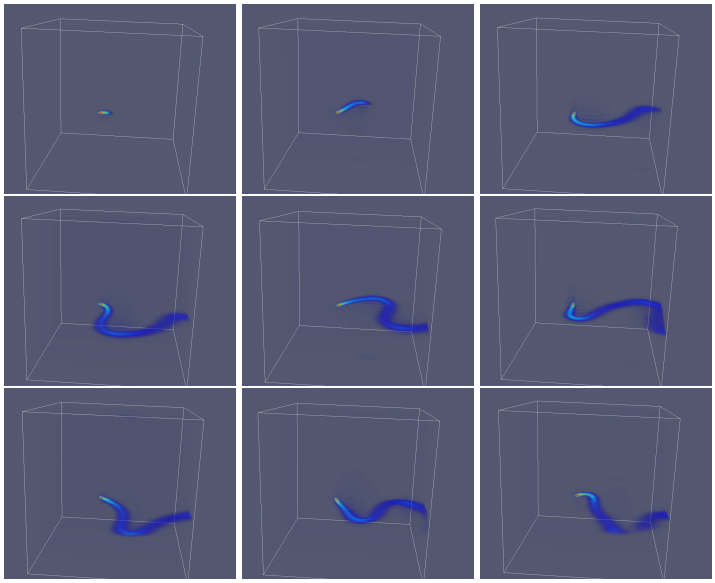


Figure 4: Simulation of pollution propagation from a chimney with Douglass-Gunn time integration scheme and residual minimization method. Snapshots for time steps 100, 200, 300, 400, 500, 600, 700, 800 and 900 with quadratic B-splines with C^1 continuity for trial and cubic B-splines with C^2 continuity for test over $50 \times 50 \times 50$ mesh.

where $M^{x,y,z}$, $K^{x,y,z}$ and $G^{x,y,z}$ are the 1D mass, stiffness and advection matrices, respectively. The numerical results from time steps 100,200,300,400,500,600,700,800, and 900 are presented in Figure 4.

We can observe the propagation of the pollution generated from a single source, the central chimney, with the non-constant wind varying in time. We do not have the exact solution for this problem, but we can observe that the 900-time steps of the simulation are stable, do not oscillate, and do not explode.

6.3 Navier-Stokes with residual minimization for cavity flow problem

We consider the non-stationary cavity flow problem over a 2D domain $\Omega = [0,1]^2$

$$\begin{cases} \partial_t v - (v \cdot \nabla) v - \dfrac{1}{Re} \Delta v + \nabla p = 0 \\ \nabla \cdot v = 0 \end{cases}$$

$$\begin{aligned} v_x(1,y) &= 1 \text{ for } y \in (0,1) \qquad & v_x(0,y) &= 0 \text{ for } y \in (0,1) \\ v_x(x,0) &= 0 \text{ for } x \in (0,1) \qquad & v_x(x,1) &= 0 \text{ for } x \in (0,1) \\ & & v_y(x,y) &= 0 \text{ for } (x,y) \in \partial\Omega \end{aligned}$$

This problem developes singularities of the pressure as illustrated in Figure 5. First, we employ the time integration scheme without the residual minimization. We show that for high Reynolds number $Re = 1000$ it implies an unexpected oscillations of the solution. This is illustrated in Table 1. Next, we incorporate the residual minimization stabilization, and we show numerical results for $Re = 100$ and $Re = 1000$ in Figure 6.

The cavity flow problem can be naturally extended to three-dimensions, by defining the problem on a cube $[0,1]^3$, using zero Dirichlet boundary condition on bottom and side faces, and adding $v = (1,0,0)$ Dirichlet boundary condition on the top face. The three-dimensional results are summarized in Figure 8.

In the cavity flow problem, the fluid located inside the cavity starts rotating due to the flow on the upper boundary. The velocity of the rotating fluid is directly related to the velocity of the flow on the boundary condition. The Reynolds number grows when we increase the velocity. The location of the vortex moves to the right-hand side corner of the domain when we increase the Reynolds number. For Reynolds number Re=100, both the Galerkin and residual minimization method works. However, for Re=1000, the Galerkin method provides unstable numerical results, while the residual minimization results in a correct solution.

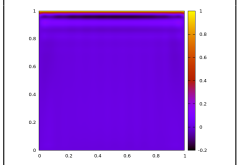
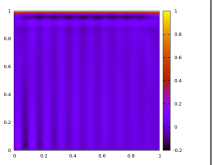
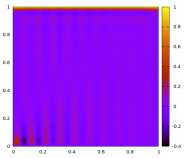
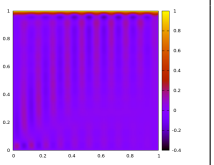
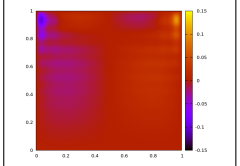
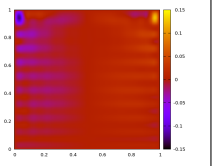
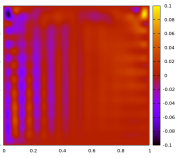
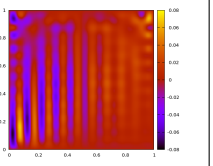
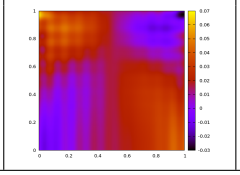
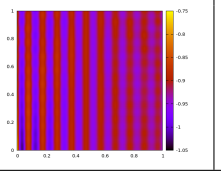
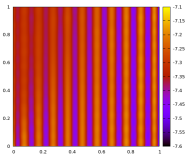
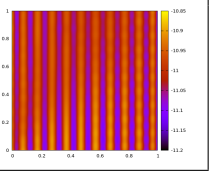
	t=0.2	t=0.5	t=0.8	t=1.0
v_1				
v_2				
p				

Table 1: Galerkin method for $Re = 1000$ for mesh 80×80. Components v_1, v_2 of the velocity and p pressure scalar field at time steps 20, 50, 80, 100 of the solution of the non-stationary cavity flow problem.

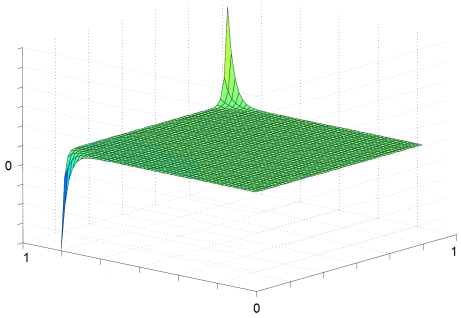


Figure 5: The singularities at the pressure solution.

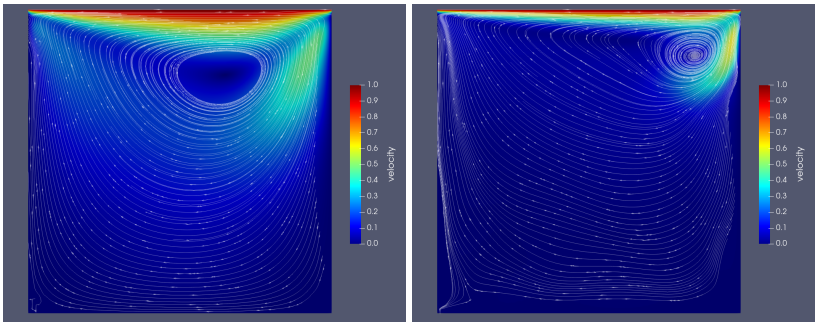


Figure 6: **Left panel:** Residual minimization method with cubic B-splines of C^1 continuity for trial and quartic B-splines of C^1 continuity for test. Mesh size of 40×40 elements with for $Re = 100$.
Middle panel: Residual minimization method with cubic B-splines of C^1 continuity for trial and quartic B-splines of C^1 continuity for test. Mesh size of 40×40 elements with for $Re = 1000$.

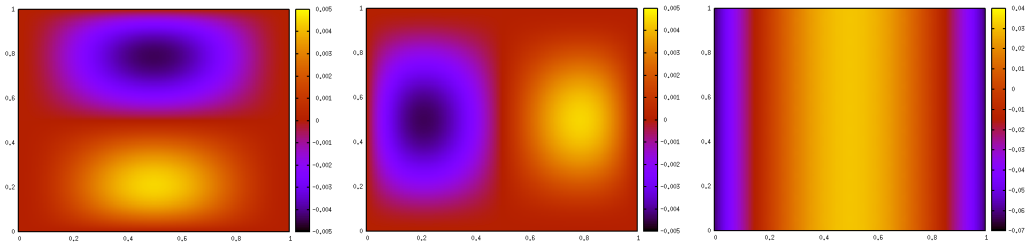


Figure 7: **Left panel:** Horizontal component v_x of the velocity scalar field of the solution of the model Stokes problem with isogeometric residual minimization method with trial = cubic B-splines of C^1 continuity, test = quartic B-splines of C^1 continuity, on a uniform mesh 20×20 elements. **Middle panel:** Vertical component v_y of the velocity of the solution of the model Stokes problem with isogeometric residual minimization method with trial = cubic B-splines of C^1 continuity, test = quartic B-splines of C^1 continuity on a uniform mesh 20×20 elements. **Right panel:** Pressure p scalar field of the solution of the model Stokes problem with isogeometric residual minimization method with trial = cubic B-splines, test = quartic B-splines on a uniform mesh 20×20 elements.

6.4 Discontinuous Galerkin with residual minimization for manufactured solution Stokes problem

We consider the Stokes equations a 2D domain $\Omega = [0,1]^2$ with no-slip boundary conditions: find $v = (v_1, v_2)$ and p such that

$$\begin{cases} -\Delta v + \nabla p = F \\ \nabla \cdot v = 0 \\ v|_{\partial\Omega} = 0 \end{cases} \tag{49}$$

where $\mathbf{f} = (f_1, f_2)$ is given by

$$\begin{aligned} f_1(x,y) &= (12-24y)x^4 + (-24+48y)x^3 + (-48y+72y^2+12)x^2 \\ &\quad + (-2+24y-72y^2+48y^3)x + 1 - 4y + 12y^2 - 8y^3 \\ f_2(x,y) &= (8-48y+48y^2)x^3 + (-12+72y-72y^2)x^2 \\ &\quad + (4-24y+48y^2-48y^3+24y^4)x - 12y^2 + 24y^3 - 12y^4 \end{aligned} \tag{50}$$

The resulting v_x, v_y, and p scalar fields are presented in Figure 7. We consider cubic B-splines of C^1 continuity as trial basis and quartic B-splines of C^1 continuity as test basis. We run the experiment on a uniform mesh 20×20 elements mesh. The numerical errors are summarized in Table 2.

Trial	Test	L2 vx	L2 vy	L2 p	L2 div$u$	H1 vx	H1 vy	H1 p	H1 div$u$
quadratic	cubic	0.0179	0.0179	0.0171	0.0127	0.31	0.31	0.31	1.95
cubic	quartic	0.00033	0.00033	0.0018	0.000328	0.0057	0.0057	0.0057	0.0444
quartic	quintic	3.76e-11	4.26e-11	1.21e-10	1.44e-11	3.46e-10	4.18e-10	2.94e-09	2.1e-09

Table 2: Convergence of the numerical errors while increasing the orders of both trial and test spaces with C^1 continuity for the Stokes model problem.

This problem has the manufactured exact solution, and our numerical experiments show that our solver can solve this problem with high numerical accuracy of the order of $10^{-9} - 10^{-11}$ as denoted in the last row of Table 2.

We also compare in Table 3 the residual minimization approach with the Galerkin method, including the computational costs expressed by the number of floating-point operations and the resulting numerical error.

6.5 Advection skew to the mesh solved with residual minimization and conjugate gradient solver

We focus on stationary advection-diffusion problem

$$-\varepsilon \nabla \cdot (\nabla u) + \beta \cdot \nabla u = 0,$$

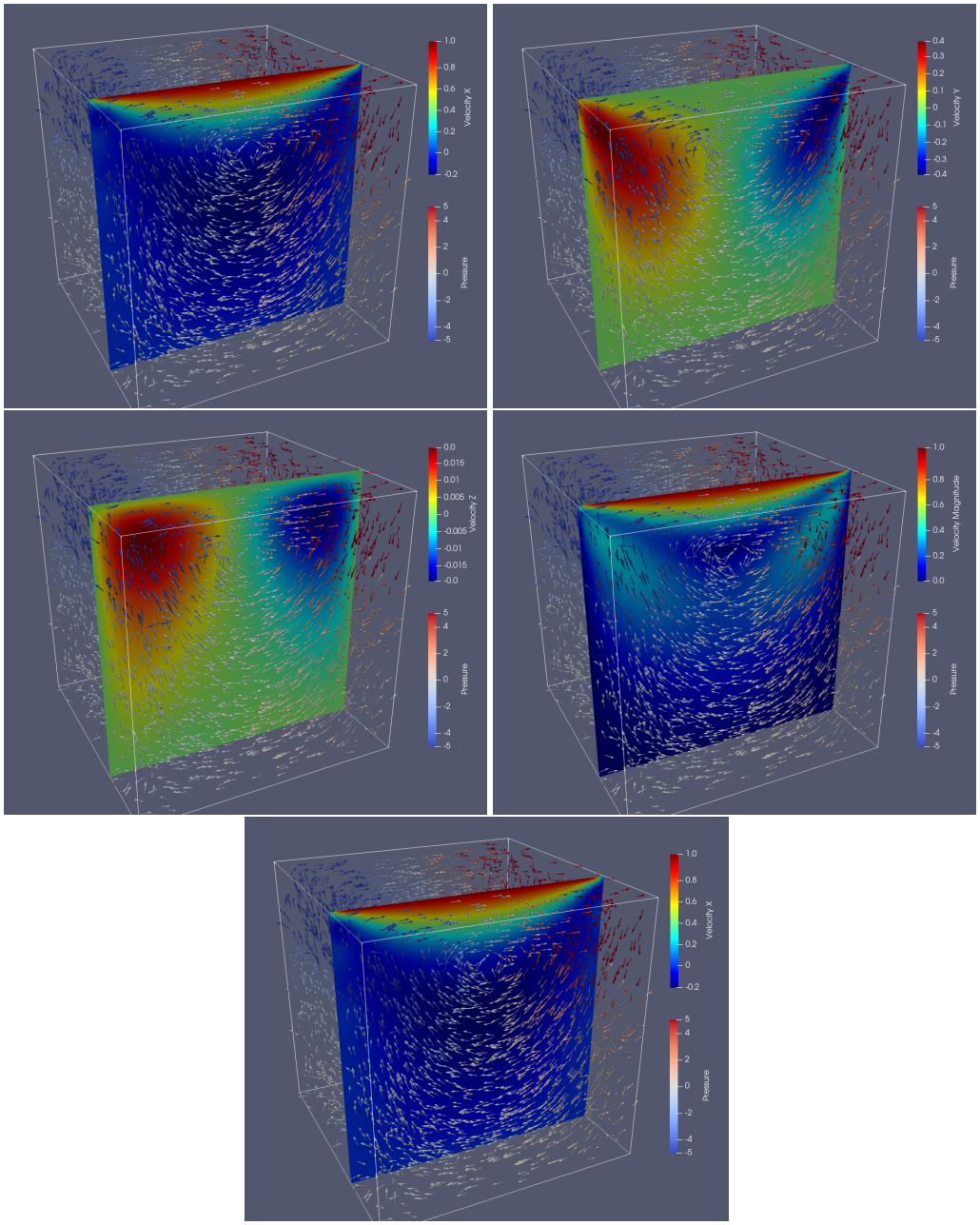


Figure 8: Cavity flow problem with isogeometric residual minimization method with trial = quartic B-splines of C^2 continuity, test = broken quartic B-splines, on a uniform mesh $8 \times 8 \times 8$ elements. **Left top panel:** Horizontal component v_x of the velocity field. **Right top panel:** Vertical component v_y of the velocity field. **Left middle panel:** Vertical component v_z of the velocity. **Right middle panel:** Magnitude of the vvelocity field. **Bottom panel:** Velocity vectors.

Trial	Test	L2 *vx*	L2 *vy*	L2 *p*	L2 div*u*	H1 *vx*	H1 *vy*	H1 *p*	H1 div*u*
quadratic	cubic	0.0179	0.0179	0.0171	0.0127	0.31	0.31	0.31	1.95
cubic	quartic	0.00033	0.00033	0.0018	0.000328	0.0057	0.0057	0.0057	0.0444
quartic	quintic	3.76e-11	4.26e-11	1.21e-10	1.44e-11	3.46e-10	4.18e-10	2.94e-09	2.1e-09

Table 3: Computational cost expressed as the number of floating-point operations and the numerical errors for the Galerkin method (trial space equal to test space) and residual minimization method (trial space different from the test space) applied for the Stokes model problem.

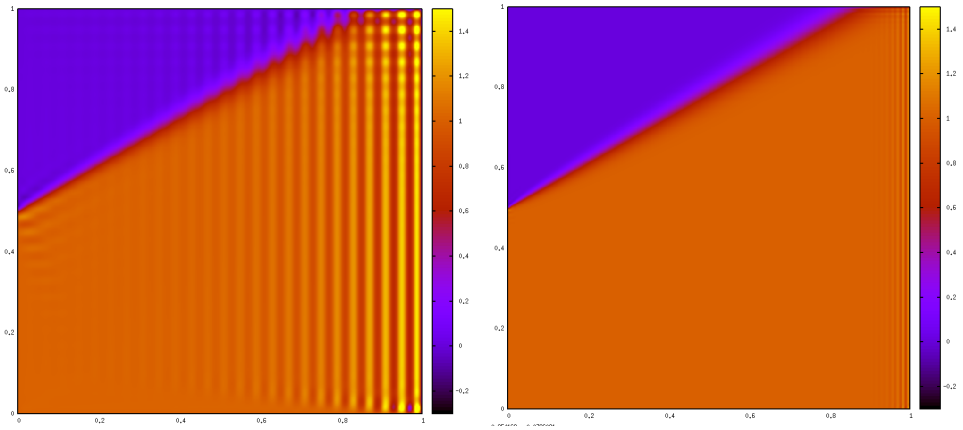


Figure 9: Advection skew to the mesh. **Left panel:** Unstable solution with Galerkin method. **Right panel:** Stabilized solution with residual minimization method.

Trial	Test	10×10	20×20	50×50	100×100	200×200
linear	cubic	3.88e+6	3.46e+6	1.75e+6	0.91e+6	0.46e+6
linear	quartic	22.5e+6	7.37e+6	2.97e+6	1.50e+6	0.74e+6
quadratic	cubic	15.6e+6	5.07e+6	1.20e+6	0.63e+6	0.32e+6
quadratic	quartic	15.6e+6	5.07e+6	2.04e+6	1.03e+6	0.51e+6

Table 4: Condition numbers of matrices in advection skew to the mesh problem.

with $\varepsilon = 10^{-6}$, $\beta = (\sqrt{3}, 1)$, solved on $[0,1]^2$ domain, with Dirichlet b.c.

$$v(x,1) = 0 \text{ for } x \in (0,1) \qquad v(x,0) = 1 \text{ for } x \in (0,1)$$
$$v(0,y) = 0 \text{ for } y \in (0.5,1) \qquad v(0,y) = 1 \text{ for } y \in (0,0.5)$$
$$v(1,y) = 0 \text{ for } y \in (0,1) \in \partial\Omega$$

Figure 9 presents the numerical results obtained with the Galerkin method (without stabilization) and with residual minimization stabilization. We employ a high-fidelity mesh of 200×200 elements, with 100 elements spread uniformly between $[0, 0.9]$ and 100 elements spread uniformly between $[0.9, 1.0]$. We use quadratic B-splines with C^0 separators between elements as a trial basis and quartic B-splines with C^0 separators as a test basis.

The advection is the driving force to propagate the substance from the bottom half of the left side of the domain, as well as from the bottom side of the domain, in the direction of the β vector. The exact solution exhibits the boundary layer at the opposite sides of the domain. This problem cannot be solved with the Galerkin method, as it is illustrated in the left panel in Figure 9. The residual minimization method can solve this problem well.

Table 4 presents the impact of the discrete spaces choice and mesh size on the condition number of the problem matrix. It seems that increasing the mesh size has a positive effect on the matrix conditioning. On the other hand, a large gap between the polynomial order of trial and test space leads to a high condition number.

Finally, we compare the computational costs of the Galerkin and residual minimization methods. We employ uniform grids. For example, for the mesh 10×10, for the Galerkin method we define

```
knot_x = [0 0 0 1 2 3 4 5 6 7 8 9 10 10 10];
points_x = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1];
knot_y = [0 0 0 1 2 3 4 5 6 7 8 9 10 10 10];
points_y = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1];
```

For the residual minimization method, we employ the same trial space as for the Galerkin method, but for the test space we reduce the continuity between the elements

```
knot_trial_x = [0 0 0 1 2 3 4 5 6 7 8 9 10 10 10];
knot_test_x = [0 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 10];
points_x = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1];
knot_trial_y = [0 0 0 1 2 3 4 5 6 7 8 9 10 10 10];
knot_test_y = [0 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 10];
```

```
points_y = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1];
```

We compare the total computational costs, including the integration and generation of the right-hand side, and the cost of solving the system of linear equations.

	mesh size	Galerkin method	Residual minimization method
Execution time [s]	5×5	0.063	0.122
Execution time [s]	10×10	0.141	0.348
Execution time [s]	20×20	0.526	1.325
Execution time [s]	40×40	1.956	5.645
Execution time [s]	80×80	8.82	21.96

Table 5: Advection skew to the mesh. Computational cost of direct solver solution with the Galerkin and residual minimization method on uniform grids.

7 Automatic mesh refinements on tensor product grids

In this section we present the algorithm for automatic refinements of tensor product grids with the residual minimization method. The input to our algorithm are the know vectors along x and y axis, ξ_i^x, $i = 1,\ldots,N_x$, and ξ_j^y, $j = 1,\ldots,N_y$.

1. We solve the residual minimization problem

$$\begin{bmatrix} G & B \\ B^T & 0 \end{bmatrix} \begin{bmatrix} r \\ u \end{bmatrix} = \begin{bmatrix} F \\ 0 \end{bmatrix} \tag{51}$$

   where B is the matrix related to the advection-skew to the mesh problem, and G is the corresponding Gram matrix.

2. We loop through elements E and we compute $res\_error(E) = \int_E r^2 + \varepsilon \nabla r \cdot \nabla r$ which represents the element local contribution to the weighted H^1 norm of the residuum

3. We loop through pairs of two consecutive non-equal knot points $\xi_k < \xi_{k+1}$ from the knot vector along x axis, and we sum up all the $res\_error(E)$ for elements E located between them. The sums represent contributions of the vertical strips of the mesh to the residual error.

4. We refine the knot vector along x axis in the intervals fulfilling the Dörfler criterion [12].

5. We repeat the last two steps along y axis.

We tested the algorithm on the advection-skew to the mesh problem with $\varepsilon = 0.01$. The resulting sequence of meshes is presented in Figure 10. The sequence of obtained solutions is summarized in Figure 11. The convergence of the global residual error is illustrated in Figure 12. We have shown that this simple mesh refinement algorithm allows us to construct a computational mesh that provides a numerical solution with a residual error of less than 0.1.

8 Quadrature improvements

Some methods of speeding up the integration in FEM and IGA have been proposed [20]. Since when using an efficient linear solver, the integration time becomes a major portion of the total computational cost; these strategies seem to provide an attractive avenue to reduce it. That said since these approaches focus on evaluating integrals in the matrix, which in transient problems is typically only computed once, their applicability and usefulness in this context is limited. Below, we briefly comment on the applicability of three integration speedup strategies described in [20] in the context of integrating the right-hand side.

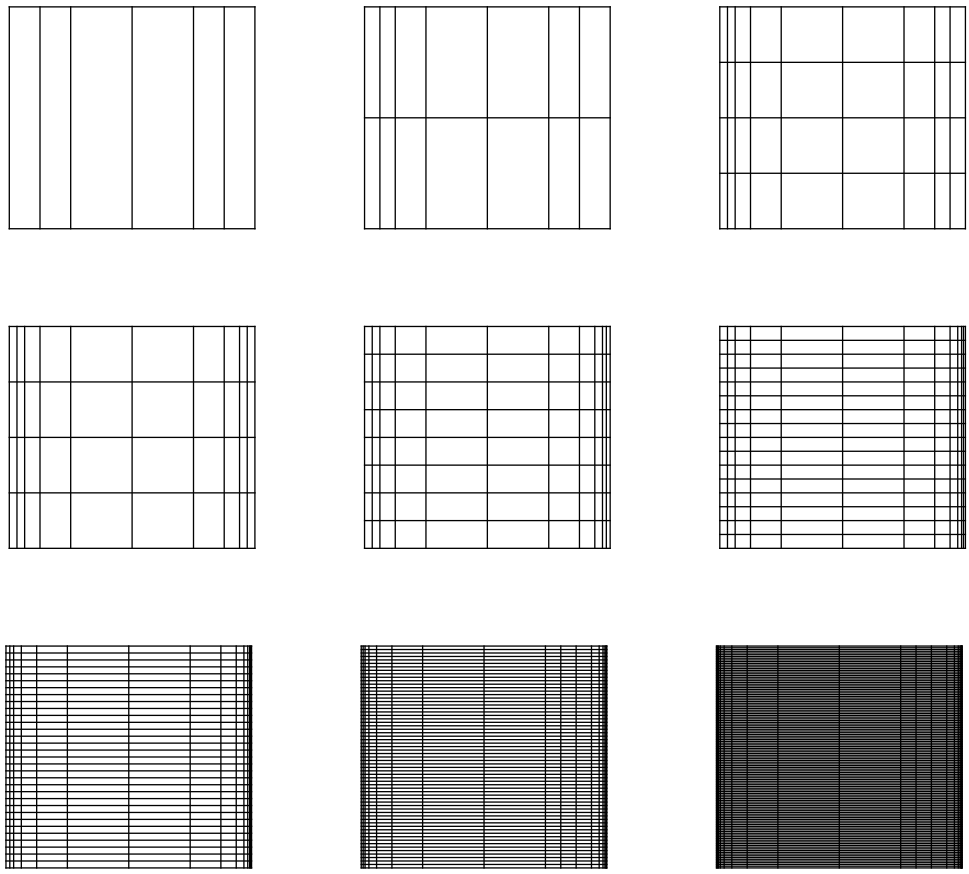


Figure 10: Advection skew to the mesh. A sequence of tensor product meshes refined using the adaptive algorithm with Dörfler criterion along the strips of the mesh using the weighted residual norm.

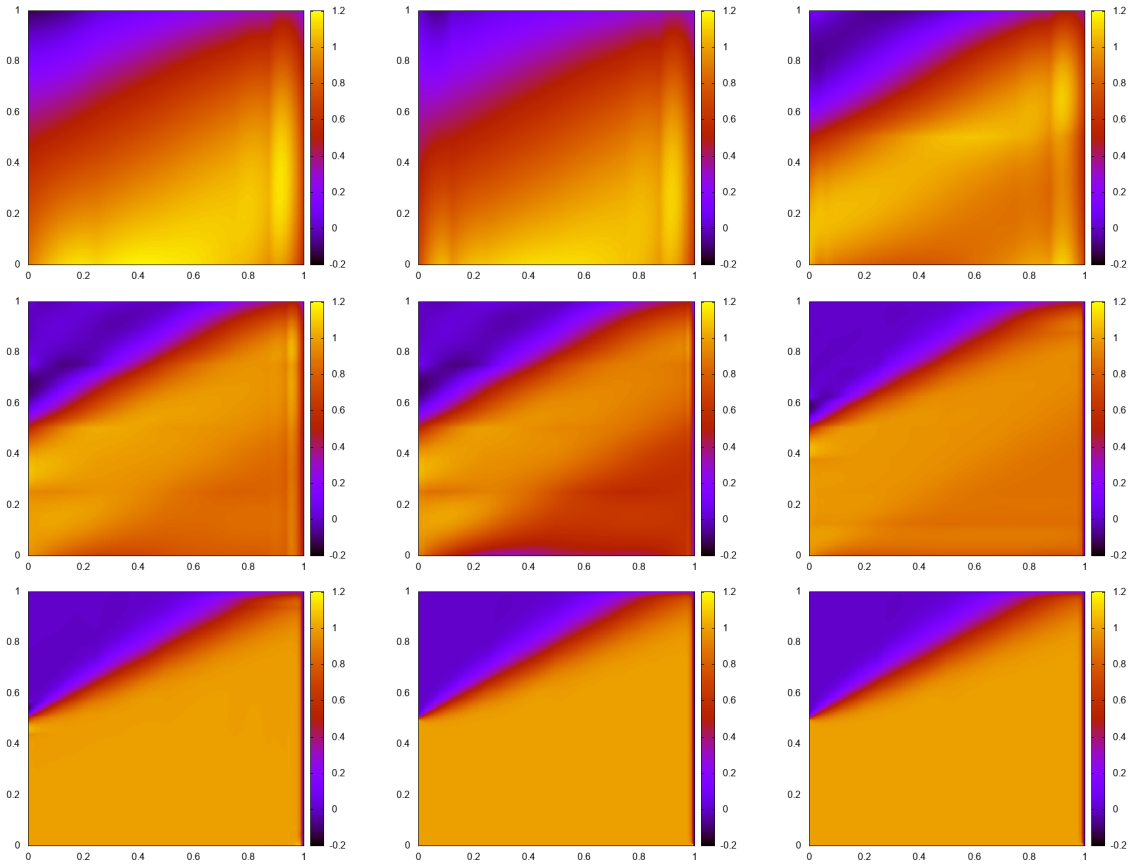


Figure 11: Advection skew to the mesh. A sequence of solution obtained on refined tensor product meshes using the adaptive algorithm.

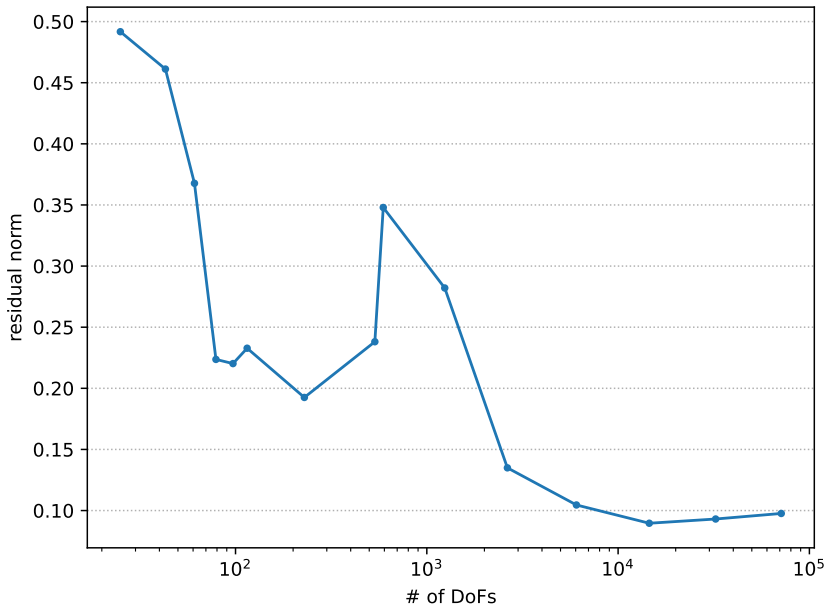


Figure 12: Advection skew to the mesh. Convergence of the residual error. The horizontal axis denotes degrees of freedom, the vertical exis denotes the norm of the residual error.

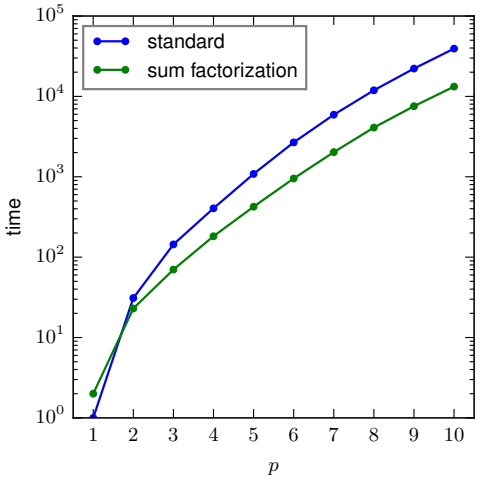


Figure 13: RHS integration time comparison

8.1 Sum factorization

Sum factorization is a technique for speeding up summations involving terms that have a tensor product structure, i.e., terms that can be represented as a product of quantities depending on a single summation index (or a subset of summation indices), by means of clever reuse of partial results. For a detailed description, we refer to [20]. This idea can be applied to integrating the right-hand side and reducing the computational cost per element from $\mathcal{O}\left(p^6\right)$ to $\mathcal{O}\left(p^4\right)$ for a 3D simulation, p being the discrete space polynomial order.

The big caveat is that this estimate assumes that the right-hand side consists of integrals of the form $\int_\Omega f e_i \, dx$, where e_i are the test functions, and f is a function with a constant evaluation cost. In practice, the right-hand side terms of the time discretization involve the solution at the previous time step. Computing its value at a single quadrature point requires evaluating a linear combination of $\mathcal{O}\left(p^3\right)$ terms since there are $(p+1)^3$ B-spline basis functions are supported on each mesh element. As a result, the asymptotic cost remains $\mathcal{O}\left(p^6\right)$ whether sum factorization is used or not, and while some efficiency gains are indeed possible to obtain (Figure 13), they are much less pronounced than in the case of computing the matrix entries.

8.2 Weighted quadratures

Replacing standard (say, Gauss-Legendre) quadrature rules with special, dedicated quadratures maintaining accuracy while using fewer quadrature points can improve efficiency sof right-hand side integration of both the matrix and the right-hand side. That said, while general-purpose quadratures integrate simple functions (pure polynomials of a fixed order) exactly and offer well-understood error bounds for other functions. These special quadratures tend to be tailored to the specific form of the integrand [2, 3, 20], which makes it less suitable for integrating the right-hand side, which can often contain less regular and well-behaved terms.

Finally, we compare the computational costs of the Galerkin and residual minimization methods. We employ uniform grids, namely for the Galerkin method

```
knot_x = [0 0 0 1 2 3 4 5 6 7 8 9 10 10 10];
points_x = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1];
knot_y = [0 0 0 1 2 3 4 4 4];
points_y = [0 0.25 0.5 0.75 1];
```

For the residual minimization method, we employ

```
knot_x = [0 0 0 1 2 3 4 5 6 7 8 9 10 10 10];
points_x = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1];
knot_y = [0 0 0 1 2 3 4 4 4];
points_y = [0 0.25 0.5 0.75 1];
```

We compare the total computational costs, including the integration and generation of the right-hand side, and the cost of solving the system of linear equations.

	mesh size	Galerkin method	Residual minimization method
Execution time [s]	5×5	0.063	0.122
Execution time [s]	10×10	0.141	0.348
Execution time [s]	20×20	0.526	0.1325
Execution time [s]	40×40	1.956	0.5645

Table 6: Advection skew to the mesh. Computational cost of solution with the Galerkin and residual minimization method on uniform grids.

8.3 Assembly by row

The idea of assembly by row is is to reorder the integration loop from the outer loop iterating over mesh elements, to the outer loop iterating over test functions. This change provides a benefit only in combination with sum factorization, effectively reducing the per-element cost of computing the matrix from $\mathcal{O}\left(p^6\right)$ (just sum factorization) to $\mathcal{O}\left(p^4\right)$ (sum factorization with assembly by row) for 3D simulation. As mentioned earlier, sum factorization for right-hand side matrix reduces the cost from $\mathcal{O}\left(p^6\right)$ to $\mathcal{O}\left(p^4\right)$, but a similar cost analysis reveals that assembling by row does not reduce it any further, which renders it not applicable to integrating right-hand side, even disregarding the fact that the cost is dominated by evaluating the previous time-step solution.

9 Parallel scalability

In this section we present the measurements of the parallel scalability for the three-dimensional simulations of the advection-diffusion problem. To measure the parallel scalability, we have moved to a shared memory machine with four Intel®Xeon® CPU E7-4860 processors, each possessing ten cores with hyperthreading (for a total of forty cores, while using more than twenty cores requires hyperthreading).

The speedup and efficiency of the code are presented in Figures 14-19. Different plots correspond to different mesh dimensions and different polynomial orders of approximation. Namely, by n we denote the mesh size in one dimension. We test our solver on a very small mesh of size $8 \times 8 \times 8$, through the mesh of size $16 \times 16 \times 16$, the mesh of size $32 \times 32 \times 32$ up to the large mesh of size $64 \times 64 \times 64$. We present the plots of the speedup and efficiency for quadratic and cubic B-splines. The speedup is defined as $S = \frac{T_1}{T_p}$, where T_1 stands for the execution time of the sequential algorithms and T_p stands for the execution time of the parallel algorithm using p cores. The perfect parallel algorithm delivers p speedup using p cores. The efficiency is defined as $E = \frac{T_1}{pT_p}$. The perfect parallel algorithm delivers efficiency equal to 1.0.

We can draw the following conclusions from the presented plots:

- For quadratic B-splines presented in Figures 14-15 and for small grids $8 \times 8 \times 8$ and $16 \times 16 \times 16$ the speedup grows up to 8 cores. The corresponding efficiency for 8 threads is of the order 0.9, and then it decreases rapidly.
- For quadratic B-splines and large grids $32 \times 32 \times 32$ and $64 \times 64 \times 64$ the speedup grows up to 16 cores. It is around 10-11 for 16 cores. The corresponding efficiency for 16 cores is around 0.7. Then, for 32 cores the speedup went down since for more than 20 cores used the hyperthreading is utilized.
- For cubic B-splines presented in Figures 16-17 and for small grid of $8 \times 8 \times 8$ the speedup grows up to 8 cores, and the corresponding efficiency is close to 1.0. Then both speedup and efficiency decrease.
- For cubic B-splines and large grids $32 \times 32 \times 32$ and $64 \times 64 \times 64$ the speedup grows up to 16 cores. It is around 12-14 for 16 cores. The corresponding efficiency for 16 cores is around 0.8-0.9. Then, for 32 cores and $32 \times 32 \times 32$ mesh the speedup grows up to 17, and for $64 \times 64 \times 64$ mesh is decreases slightly since for more than 20 cores the hyperthreading is used.
- For quartic B-splines presented in Figures 18-19 and for small grids of $8 \times 8 \times 8$ and $16 \times 16 \times 16$ the speedup grows up to 16 cores, and the corresponding efficiency is between 0.8-1.0. Then both speedup and efficiency decrease slightly.
- For quartic B-splines and large grids $32 \times 32 \times 32$ and $64 \times 64 \times 64$ the speedup grows up to 32 cores. It is around 15 for 16 cores (near perfect speedup) and around 20 for 32 cores, where we use the hyperthreading (more than 20 cores). The corresponding efficiency for 16 cores is around 0.9-1.0. Then, for 32 cores the efficiency decreases slightly down to 0.6-0.7.

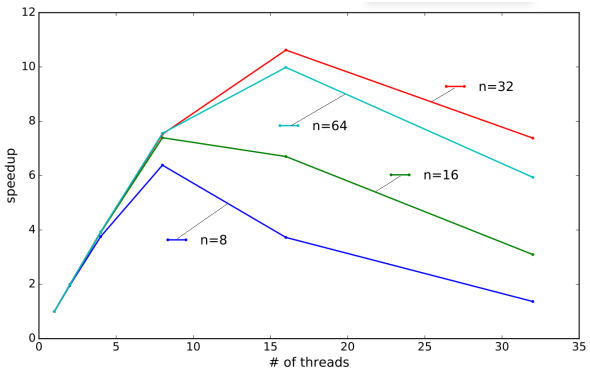


Figure 14: Speedup $p = 2$

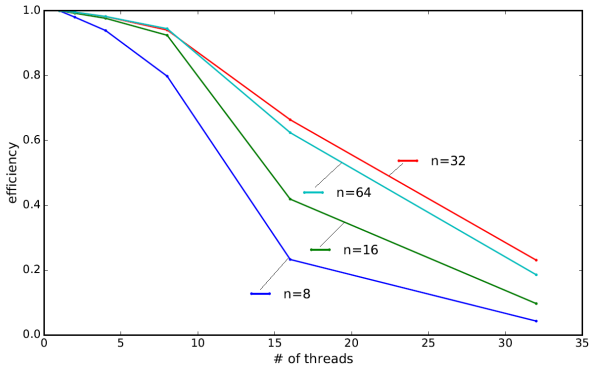


Figure 15: Efficiency $p = 2$

- Increasing the mesh size increases the parallel scalability up to $32 \times 32 \times 32$ mesh. Larger mesh, $64 \times 64 \times 64$ performs slightly worse than $32 \times 32 \times 32$ mesh.
- The most interesting observation is that while increasing the B-splines order we observe improvement of the parallel scalability.

The plots depict the time of the integration, i.e. assembling of the system, which constitutes the vast majority of the computational time. Cost of rest of each single time step – solving the system using ADS – was found to be negligible in comparison, and preparatory steps (array allocation, etc.) become insignificant given a large number of time steps computed in a full simulation.

We measure the speedup for 8, 16, and 32 cores. The speedup of execution for a computational mesh of 32x32x32 elements is better than the speedup for a computational mesh of 64x64x64 for the number of cores larger than the number of physical cores. The computational nodes have 10 physical cores and a total of 40 cores using the Hyper-threading (HT) technology. Hyper-threading allows a single physical core to execute multiple threads

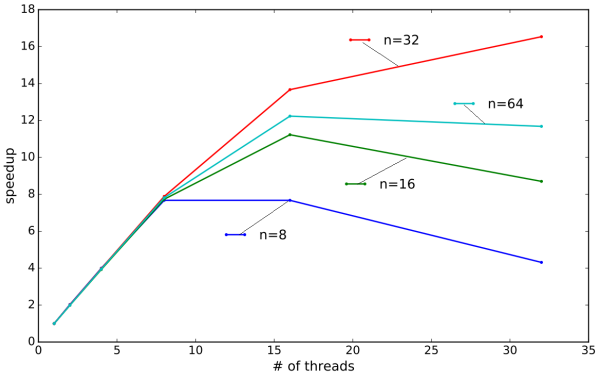


Figure 16: Speedup $p = 3$

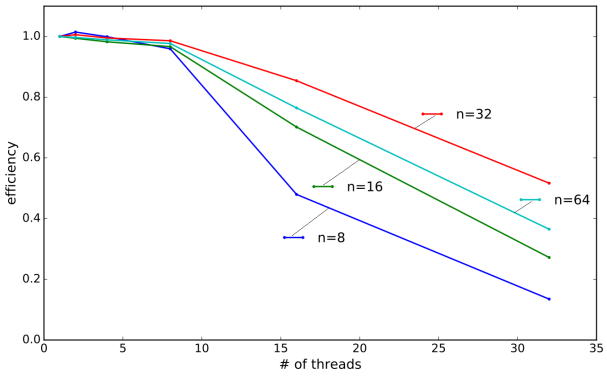


Figure 17: Efficiency $p = 3$

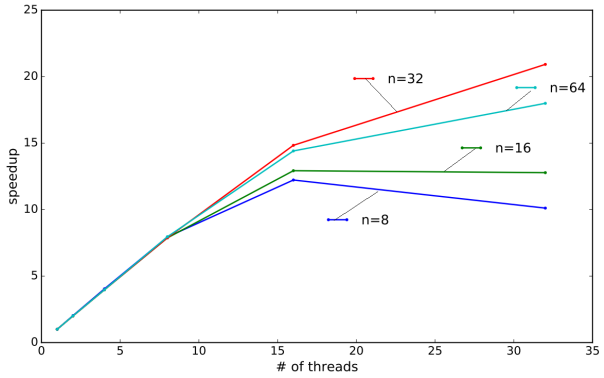


Figure 18: Speedup $p = 4$

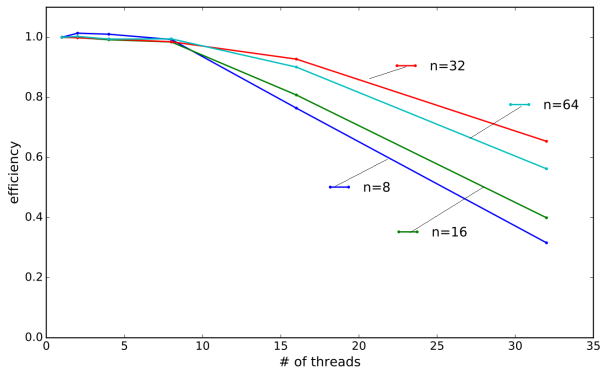


Figure 19: Efficiency $p = 4$

simultaneously. The physical core can only execute one instruction at a time. HT allows the CPU core to switch between multiple threads quickly. This is highly beneficial when one thread waits for data from RAM or another resource. The core can switch to another thread that is ready to execute. Please note that the speedup achieved by Hyper-threading depends on the nature of the workload. Some applications and tasks benefit significantly from HT, while others may see slight improvement or deterioration

10 Conclusions

We presented an open-source parallel shared-memory C++ software for simulations of time-dependent phenomena. It supported IGA discretizations on tensor product grids and employs three solvers: the alternating-directions (ADS) linear cost $O(\mathcal{N})$ solver for problems suitable for direction splitting of the differential operators, iterative conjugate gradients solver, as well as an interface to the MUMPS direct solver. We implemented implicit time-integration schemes suitable for direction splitting, including Peaceman-Reachford, Douglass-Gunn, and leapfrog methods with Crank-Nicolson or explicit Euler. We provided support for scalar and vector fields and systems of PDEs in two and three dimensions. We also supported the interface to ParaView and Gnuplot. The main novelty of our software was the incorporation of the residual minimization and Discontinuous Galerkin stabilization methods. We presented several numerical examples, including two and three-dimensional advection-diffusion problems with different time integration schemes using residual minimization stabilization, two-dimensional Navier-Stokes problem with residual minimization stabilization, and two and three-dimensional Stokes problem with Discontinuous Galerkin stabilization. In our examples, we employed the alternating directions solver, direct solver, and the conjugate gradients iterative solver.

Several mature and high-quality numerical libraries exist for simulations with B-spline basis functions, including PetIGA and GeoPDE. They support arbitrary geometries and provide interfaces to different direct and iterative solvers. These libraries are generally dedicated to standard Galerkin method formulations. Our RM-IGA-ADS software employs the residual minimization problem to automatically stabilize difficult problems for which the Galerkin method results in unstable solutions. It also supports DG method discretization. RM-IGA-ADS works on tensor product grids, and thus it often enables the use of an ultrafast alternating directions solver for time-dependent problems. RM-IGA-ADS is an extension of IGA-ADS into residual minimization and DG methods.

11 Acknowledgments

Research project supported by the program “Excellence initiative - research university" for the AGH University of Krakow.

Appendix

A Installation of the code

Installation of IGA-ADS solver on Linux server.

1. Install cmake (version 3.13 or later)
   ```
   sudo apt install cmake
   ```
2. Install gfortran
   ```
   sudo apt install gfortran
   ```
3. Install LAPACK
   ```
   sudo apt install liblapack-doc
   sudo apt install liblapack-dev
   ```
4. Install BLAS
   ```
   sudo apt install libblas-doc
   sudo apt install libblas-dev
   ```

5. Install MUMPS

   ```
   sudo apt install libmumps-dev
   ```

6. Install boost (version 1.58 or later)

   ```
   sudo apt install libboost-all-dev
   ```

7. Install LLVM (necessary for Galois)

   ```
   sudo apt install llvm-12 llvm-12-dev
   ```

8. Download IGA-ADS-RM code

   ```
   git clone https://github.com/marcinlos/iga-ads-rm
   cd iga-ads-rm
   git checkout develop
   ```

9. Install other necessary libraries (including Galois)

   ```
   cd ..
   DEPS=$(realpath deps)
   iga-ads-rm/scripts/install-dependencies.sh $(realpath deps-build) "${DEPS}"
   ```

   As described in https://github.com/IntelligentSoftwareSystems/Galois/issues/401 Galois does not compile with GCC compiler version $\geq$ 11. In other to fix this problem it is necessary to add to install-dependencies.sh the following line after line 82

   ```
   sed -i '36i #include <optional>' Galois/tools/graph-convert/graph-convert.cpp
   # Install Galois GALOIS_VER=6.0
   git clone -branch release-${GALOIS_VER} -depth=1 -quiet
      https://github.com/IntelligentSoftwareSystems/Galois
   sed -i '36i #include <optional>' Galois/tools/graph-convert/graph-convert.cpp
   ```

10. Compile IGA-ADS-RM

    ```
    cd iga-ads-rm
    mkdir build
    cmake -S . -B build -D CMAKE_BUILD_TYPE=Release -D ADS_USE_GALOIS=ON -D USE_MUMPS=ON
    - D CMAKE_PREFIX_PATH="${DEPS}"
    cmake -build build -j $(nproc)
    ```

11. Exemplary run

    ```
    cd build/examples
    ./heat_2d
    ```

12. Generation of plots from out_* data files

    ```
    gnuplot
    gnuplot > plot "out_100.data" with imag
    ```

References

[1] *Galois framework* http://iss.ices.utexas.edu/?p=projects/galois.

[2] R. AIT-HADDOU, M. BARTOŇ, AND V. M. CALO, *Explicit gaussian quadrature rules for c1 cubic splines with symmetrically stretched knot sequences*, Journal of Computational and Applied Mathematics, 290 (2015), pp. 543–552.

[3] M. BARTOŇ AND V. M. CALO, *Gauss–galerkin quadrature rules for quadratic and cubic spline spaces and their application to isogeometric analysis*, Computer-Aided Design, 82 (2017), pp. 57–67. Isogeometric Design and Analysis.

[4] G. BIRKHOFF, R. VARGA, AND D. YOUNG, *Alternating Direction Implicit Methods*, Advances in Computers, 3 (1962), pp. 189–273.

[5] V. CALO, M. ŁOŚ, Q. DENG, I. MUGA, AND M. PASZYŃSKI, *Isogeometric residual minimization method (igrm) with direction splitting preconditioner for stationary advection-dominated diffusion problems*, Computer Methods in Applied Mechanics and Engineering, 373 (2021), p. 113214.

[6] J. CHAN AND J. EVANS, *A minimal-residual finite element method for the convection-diffusion equations*, ICES-REPORT, The University of Texas at Austin, USA, (2013).

[7] J. A. COTTRELL, T. HUGHES, AND Y. BAZILEVS, *Isogeometric analysis: toward integration of CAD and FEA*, John Wiley & Sons, 2009.

[8] R. COURANT, K. FRIEDRICHS, AND H. LEWY, *On the partial difference equations of mathematical physics*, IBM Journal of Research and Development, 11 (1967), pp. 215–234.

[9] J. CRANK AND P. NICOLSON, *A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type*, Mathematical Proceedings of the Cambridge Philosophical Society, 43 (1947), p. 50–67.

[10] L. DALCIN, N. COLLIER, P. VIGNAL, A. CÔRTES, AND V. CALO, *Petiga: A framework for high-performance isogeometric analysis*, Computer Methods in Applied Mechanics and Engineering, 308 (2016), pp. 151–181.

[11] D. A. DI PIETRO AND A. ERN, *Mathematical aspects of discontinuous Galerkin methods*, vol. 69, Springer Science, 2012.

[12] W. DÖRFLER, *A convergent adaptive algorithm for poisson's equation*, SIAM Journal on Numerical Analysis, 33 (1996), pp. 1106–1124.

[13] J. DOUGLAS AND H. RACHFORD, *On the Numerical Solution of Heat Conduction Problems in Two and Three Space Variables*, Transactions of American Mathematical Society, 82 (1956), pp. 421–439.

[14] I. S. DUFF AND J. K. REID, *The multifrontal solution of indefinite sparse symmetric linear*, ACM Transactions on Mathematical Software (TOMS), 9 (1983), pp. 302–325.

[15] ——, *The multifrontal solution of unsymmetric sets of linear equations*, SIAM Journal on Scientific and Statistical Computing, 5 (1984), pp. 633–641.

[16] K. ERIKSSON AND C. JOHNSON, *Adaptive finite element methods for parabolic problems I: A linear model problem*, SIAM Journal on Numerical Analysis, 28 (1991), pp. 43–77.

[17] E. ERTURK, T. C. CORKE, AND C. GÖKÇÖL, *Numerical solutions of 2-d steady incompressible driven cavity flow at high reynolds numbers*, International Journal for Numerical Methods in Fluids, 48 (2005), pp. 747–774.

[18] J. GUERMOND AND P. MINEV, *A new class of massively parallel direction splitting for the incompressible navier-stokes equations*, Computer Methods in Applied Mechanics and Engineering, 200 (2011), p. 2083 – 2093. Cited by: 40.

[19] M. A. HASSAAN, M. BURTSCHER, AND K. PINGALI, *Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms*, Acm Sigplan Notices, 46 (2011), pp. 3–12.

[20] R. R. HIEMSTRA, G. SANGALLI, M. TANI, F. CALABRÒ, AND T. J. HUGHES, *Fast formation and assembly of finite element matrices with application to isogeometric linear elasticity*, Computer Methods in Applied Mechanics and Engineering, 355 (2019), pp. 234–260.

[21] T. HUGHES, J. COTTRELL, AND Y. BAZILEVS, *Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement*, Computer methods in applied mechanics and engineering, 194 (2005), pp. 4135–4195.

[22] M. KULKARNI, K. PINGALI, B. WALTER, G. RAMANARAYANAN, K. BALA, AND L. P. CHEW, *Optimistic parallelism requires abstractions*, in Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007, pp. 211–222.

[23] A. LENHARTH, D. NGUYEN, AND K. PINGALI, *Priority queues are not good concurrent priority schedulers*, in European Conference on Parallel Processing, Springer, 2015, pp. 209–221.

[24] M. ŁOŚ, S. ROJAS, M. PASZYŃSKI, I. MUGA, AND V. CALO, *Discontinuous galerkin based isogeometric residual minimization for the stokes problem (DGIRM)*, invited to the special issue of Journal of Computational Science on 20th anniversary of ICCS conference, (2020).

[25] M. ŁOŚ, M. WOŹNIAK, M. PASZYŃSKI, A. LENHARTH, M. A. HASSAAN, AND K. PINGALI, *IGA-ADS: Isogeometric analysis FEM using ADS solver*, Computer Physics Communications, 217 (2017), pp. 99–116.

[26] K. MISAN, M. KOZIEJA, M. ŁOŚ, D. GRYBOŚ, J. LESZCZYŃSKI, P. MACZUGA, M. WOŹNIAK, A. O. SERRA, AND M. PASZYŃSKI, *The first scientiffic evidence for the hail cannon*, in Computational Science – ICCS 2023, J. Mikyška, C. de Mulatier, M. Paszynski, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. Sloot, eds., Cham, 2023, Springer Nature Switzerland, pp. 177–190.

[27] K. MISAN, W. ORMANIEC, A. KANIA, M. KOZIEJA, M. ŁOŚ, D. GRYBOŚ, J. LESZCZYŃSKI, AND M. PASZYŃSKI, *Fast isogeometric analysis simulations of a process of air pollution removal by artificially generated shock waves*, in Computational Science – ICCS 2022, D. Groen, C. de Mulatier, M. Paszynski, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, eds., Cham, 2022, Springer International Publishing, pp. 298–311.

[28] D. PEACEMAN AND H. RACHFORD, *The Numerical Solution of Parabolic and Elliptic Differential Equations*, Journal of Society of Industrial and Applied Mathematics, 3 (1955), pp. 28–41.

[29] D. W. PEACEMAN AND H. H. RACHFORD, JR., *The numerical solution of parabolic and elliptic differential equations*, Journal of the Society for Industrial and Applied Mathematics, 3 (1955), pp. 28–41.

[30] D. PIETRO AND A. ERN, *Mathematical Aspects of Discontinuous Galerkin Methods*, Springer, 2011.

[31] K. PINGALI, D. NGUYEN, M. KULKARNI, M. BURTSCHER, M. A. HASSAAN, R. KALEEM, T.-H. LEE, A. LENHARTH, R. MANEVICH, M. MÉNDEZ-LOJO, ET AL., *The tao of parallelism in algorithms*, in Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, 2011, pp. 12–25.

[32] G. STRANG, *On the construction and comparison of difference schemes*, SIAM Journal on Numerical Analysis, 5 (1968), pp. 506–517.

[33] R. VÁZQUEZ, *A new design for the implementation of isogeometric analysis in octave and matlab: Geopdes 3.0*, Computers & Mathematics with Applications, 72 (2016), pp. 523–554.

[34] E. WACHSPRESS AND G. HABETLER, *An Alternating-Direction-Implicit Iteration Technique*, Journal of Society of Industrial and Applied Mathematics, 8 (1960), pp. 403–423.

[35] M. ŁOŚ, I. MUGA, J. MUÑOZ-MATUTE, AND M. PASZYŃSKI, *Isogeometric residual minimization (igrm) for non-stationary stokes and navier–stokes problems*, Computers & Mathematics with Applications, 95 (2021), pp. 200–214. Recent Advances in Least-Squares and Discontinuous Petrov–Galerkin Finite Element Methods.

[36] M. ŁOŚ, J. MUÑOZ-MATUTE, I. MUGA, AND M. PASZYŃSKI, *Isogeometric residual minimization method (igrm) with direction splitting for non-stationary advection–diffusion problems*, Computers & Mathematics with Applications, 79 (2020), pp. 213–229.

[37] M. ŁOŚ, S. ROJAS, M. PASZYŃSKI, I. MUGA, AND V. M. CALO, *Dgirm: Discontinuous galerkin based isogeometric residual minimization for the stokes problem*, Journal of Computational Science, 50 (2021), p. 101306.

[38] M. ŁOŚ, M. WOŹNIAK, I. MUGA, AND M. PASZYNSKI, *Three-dimensional simulations of the airborne covid-19 pathogens using the advection-diffusion model and alternating-directions implicit solver*, Bulletin of the Polish Academy of Sciences Technical Sciences, 69 (2021), p. e137125.