


Sorbonne Université

Project G-HPC1

Jan ESQUIVEL MARXEN, Thomas GANTZ

May 16, 2025

 github.com/gantz-thomas-gantz/SaddlePoint-SchurPETSc-FreeFEM

Abstract

Solving linear systems is a fundamental building block in computational mathematics. While the applications from which these systems arise are diverse, the systems themselves are often less varied than one might expect: they often tend to be symmetric, positive definite, sparse, or in saddle point form. However, the relentlessly growing size of these systems can pose significant challenges. Therefore, one must exploit their structure by applying the most suitable mathematical algorithms and, in addition, develop efficient computational methods (primarily parallelism) to ensure accurate and timely solutions. The goal of this project is to explore and present these potential solutions within the framework of the PETSc (Portable, Extensible Toolkit for Scientific Computation) library.

Contents

1	Saddle Point Problems	1
1.1	Optimization	1
1.1.1	Least Squares under Equality Constraints	1
1.1.2	Lagrange-Newton Method	1
1.2	Poisson Equation	2
1.2.1	Volume constraint	2
1.2.2	Finite Element Discretization	3
2	Mathematical Foundation	3
2.1	Solvers	3
2.1.1	Common Solvers	4
2.1.2	Using Solvers in PETSc	4
2.2	Preconditioners	4
2.2.1	Common Preconditioners	5
2.2.2	PCFIELDSPLIT	5
3	Implementation	7
3.1	Poisson Equation with PETSc	7
3.1.1	Goal	7
3.1.2	Assembling the Saddle Point Problem	7
3.1.3	Solving the Saddle Point Problem	7
3.1.4	Results	9
3.2	Stokes Equation with FreeFEM	10
3.2.1	FreeFEM	10
3.2.2	Goal	10
3.2.3	Assembling the Saddle Point Problem	12
3.2.4	Solving the Saddle Point Problem	13
3.2.5	Results	13
4	Conclusion	15

1 Saddle Point Problems

In this section, we aim to motivate saddle point problems as a frequently occurring type of linear system in various domains of applied mathematics. In particular, we will focus on optimization and the numerical solving of **P**artial **D**ifferential **E**quations (PDEs).

1.1 Optimization

The results of the following optimization part are taken from [Tré25].

1.1.1 Least Squares under Equality Constraints

Let $A \in M_{m,n}(\mathbb{R})$ with $m \geq n$ injective, $b \in \mathbb{R}^m$, and $C \in M_{k,n}(\mathbb{R})$ with $k \leq n$ surjective. We are interested in the problem

$$\min_x \frac{1}{2} \|Ax - b\|^2 \quad \text{subject to} \quad Cx = d.$$

This is equivalent to minimizing the function $f(x) = \frac{1}{2}x^T A^T A x - b^T A x$. Denote the rows of matrix C by $L_1, \dots, L_k \in M_{1,n}(\mathbb{R})$. These k affine constraints are

$$h_i(x) = L_i x - d_i = 0, \quad i = 1, \dots, k.$$

Since C is surjective, a qualification condition is satisfied: the gradients $\nabla h_i(x) = L_i^T$, $i = 1, \dots, k$, are linearly independent. Therefore we are in the normal case with unique Lagrange multipliers. The problem has at most one solution x since f is strictly convex (because the Hessian $Hess f(x) = A^T A$ is symmetric and positive definite). Furthermore, there exists at least one solution because f is coercive, and the constraint set $\bigcap_{i=1}^k h_i^{-1}(0)$ is closed. The unique solution x is characterized by the Lagrange multiplier condition (which is both necessary and sufficient in the qualified case of a convex objective function with affine equality constraints), i.e. there exists unique $\lambda = (\lambda_1, \dots, \lambda_k) \in \mathbb{R}^k$ such that

$$\nabla f(x) + \sum_{i=1}^k \lambda_i \nabla h_i(x) = 0 \quad \text{and} \quad Cx = d,$$

which can be written in matrix form as

$$\begin{pmatrix} A^T A & C^T \\ C & 0 \end{pmatrix} \begin{pmatrix} x \\ \lambda \end{pmatrix} = \begin{pmatrix} A^T b \\ d \end{pmatrix},$$

a saddle point problem.

1.1.2 Lagrange-Newton Method

Consider a well-posed equality-constrained optimization problem with $f : \mathbb{R}^n \rightarrow \mathbb{R}$ and $h : \mathbb{R}^n \rightarrow \mathbb{R}^k$:

$$\min_x f(x) \quad \text{subject to} \quad h(x) = 0.$$

Assume that this problem has a minimizer $x^* \in \mathbb{R}^n$ and a corresponding vector of Lagrange multipliers $\lambda^* \in \mathbb{R}^k$. The Lagrangian for this problem is defined as:

$$L(x, \lambda) = f(x) + \lambda^T h(x).$$

To find the solution, we need to solve the system of equations:

$$F(x, \lambda) = \begin{pmatrix} \frac{\partial L}{\partial x}(x, \lambda) \\ h(x) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

The Lagrange-Newton method consists of solving this system of equations using Newton's method:

$$\begin{pmatrix} x_{k+1} \\ \lambda_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ \lambda_k \end{pmatrix} - [dF(x_k, \lambda_k)]^{-1} F(x_k, \lambda_k),$$

where the Jacobian of $F(x, \lambda)$, denoted by $dF(x_k, \lambda_k)$, is given by:

$$dF(x_k, \lambda_k) = \begin{pmatrix} \frac{\partial^2 L}{\partial x^2}(x_k, \lambda_k) & \frac{dh(x_k)}{dx} \\ \left(\frac{dh(x_k)}{dx}\right)^T & 0 \end{pmatrix}.$$

Thus, the Newton step consists of solving a saddle point problem at each iteration.

1.2 Poisson Equation

The results of the following PDE parts are taken from [AP25]. As our toy problem we consider the Poisson equation with homogeneous Dirichlet boundary conditions:

$$-\Delta u = f \quad \text{in } \Omega, \quad (1)$$

$$u = 0 \quad \text{on } \partial\Omega, \quad (2)$$

- $\Omega \subset \mathbb{R}^n$ is a bounded domain with Lipschitz boundary $\partial\Omega$,
- $u : \Omega \rightarrow \mathbb{R}$ is the unknown function,
- $f \in L^2(\Omega)$ is a given source term,
- $\Delta u = \text{div}(\nabla u)$ is the Laplace operator.

We seek a weak solution in the Sobolev space $H^1(\Omega)$, defined as

$$H^1(\Omega) = \{v \in L^2(\Omega) \mid \nabla v \in (L^2(\Omega))^n\}. \quad (3)$$

Furthermore we impose the homogeneous Dirichlet boundary conditions directly in the search space. Multiplying by a test function $v \in H_0^1(\Omega)$ and integrating by parts, we obtain the weak formulation:

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} f v \, dx, \quad \forall v \in H_0^1(\Omega). \quad (4)$$

A unique solution $u \in H_0^1(\Omega)$ exists by the Lax-Milgram theorem. To introduce a volume constraint, we impose the additional condition:

$$\int_{\Omega} u \, dx = c, \quad (5)$$

where $c \in \mathbb{R}$ is a given constant.

1.2.1 Volume constraint

Introducing a Lagrange multiplier $\lambda \in \mathbb{R}$ to enforce the volume constraint, the weak form's associated Lagrangian functional becomes

$$L(u, \lambda) = \frac{1}{2} \int_{\Omega} |\nabla u|^2 \, dx - \int_{\Omega} f u \, dx + \lambda \left(\int_{\Omega} u \, dx - c \right) \quad (6)$$

By taking the variation with respect to u and then λ we get the following variational formulation: Find $(u, \lambda) \in H_0^1(\Omega) \times \mathbb{R}$ such that:

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx + \lambda \int_{\Omega} v \, dx = \int_{\Omega} f v \, dx, \quad \forall v \in H_0^1(\Omega), \quad (7)$$

$$\int_{\Omega} u \, dx = c. \quad (8)$$

Note that u is the saddle point problem's primal variable and λ acts as the dual variable enforcing the constraint. A unique solution still exists since the constraint operator $u \mapsto \int_{\Omega} u \, dx$ is surjective.

1.2.2 Finite Element Discretization

We approximate u using a finite-dimensional subspace $V_h \subset H_0^1(\Omega)$, spanned by basis functions $\{\phi_i\}_{i=1}^N$ (Galerkin's method). The discrete solution is sought in the form:

$$u_h = \sum_{i=1}^N U_i \phi_i. \quad (9)$$

The Lagrange multiplier λ_h remains a scalar. Testing with basis functions ϕ_j for $j = 1, \dots, N$, we obtain the discrete system:

$$\sum_{i=1}^N U_i \int_{\Omega} \nabla \phi_i \cdot \nabla \phi_j dx + \lambda_h \int_{\Omega} \phi_j dx = \int_{\Omega} f \phi_j dx, \quad \forall j = 1, \dots, N, \quad (10)$$

$$\sum_{i=1}^N U_i \int_{\Omega} \phi_i dx = c. \quad (11)$$

In matrix form, this can be written as:

$$\begin{bmatrix} A & C \\ C^T & 0 \end{bmatrix} \begin{bmatrix} U \\ \lambda_h \end{bmatrix} = \begin{bmatrix} F \\ c \end{bmatrix}, \quad (12)$$

where:

$$A_{ij} = \int_{\Omega} \nabla \phi_i \cdot \nabla \phi_j dx, \quad C_i = \int_{\Omega} \phi_i dx, \quad F_j = \int_{\Omega} f \phi_j dx. \quad (13)$$

This system represents the variational formulation of the saddle point problem. Choosing a very simple 1D example on a boundary $\Omega = (0, L) = (0, 1)$ with homogeneous Dirichlet boundary conditions

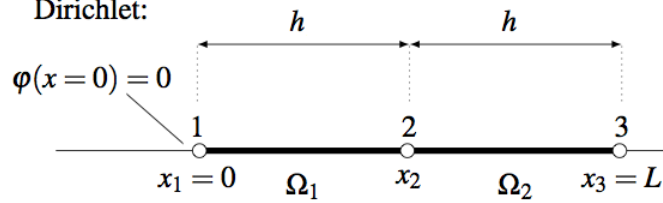


Figure 1: Discretized domain.

and $f(x) = 1$, $c = 1$, we have (see 1D Example Derivation):

$$A = \frac{1}{h} \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}, \quad C = h \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}, \quad F = h \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}.$$

2 Mathematical Foundation

Since our goal is to solve linear systems efficiently, we must focus on two essential components: the solver and the preconditioner.

2.1 Solvers

To solve a linear system of equations of the form $Ax = b$ one can employ various solvers depending on the properties of the matrix A and the trade-off between the computational cost and the accuracy of the solution desired.

2.1.1 Common Solvers

Solvers can be broadly categorized into direct and iterative methods, with Krylov subspace methods forming the most important subclass of iterative techniques. [Dup24] [Gra25]

Direct Methods. LU or Cholesky factorization, are standard for small to moderately sized problems. They provide exact solutions in finite arithmetic and are particularly effective when the matrix is dense or structured. However, they scale poorly for large, sparse systems. This is due to fill-in: even if A is sparse, its LU factors often are not. While reordering strategies can reduce fill-in, they rarely eliminate it. In contrast, iterative methods are designed to take advantage of sparsity, often with significantly lower memory and computational cost. For example, a direct LU decomposition typically costs $\mathcal{O}(n^3)$ operations, whereas Krylov subspace methods can solve a sparse system in $\mathcal{O}(kn)$, where $k \ll n$ is the number of iterations. However, as a good practice, we use direct methods to test the coherence of the code with the original problem or to compare them to iterative solutions.

Krylov Subspace Methods These form a class of iterative solvers that construct solutions within a sequence of nested subspaces generated by repeated applications of A to the residual. Notably, these methods are theoretically exact in at most n steps (for an $n \times n$ matrix) in exact arithmetic. In practice, they are truncated much earlier and paired with preconditioners to achieve fast convergence. The most common Krylov solvers include:

- **Conjugate Gradient (CG):** Best suited for symmetric positive definite (SPD) matrices. It minimizes the error in the A -norm and profits from a short recurrence sequence.
- **MINRES (Minimum Residual):** Designed for symmetric, possibly indefinite matrices. It minimizes the residual in the A -norm.
- **GMRES (Generalized Minimal Residual):** Applicable to general non-symmetric matrices. It minimizes the residual over a Krylov subspace, but at the cost of increasing memory due to its growing basis. Therefore, restarted versions (e.g., GMRES(30)) are typically used.

Stationary Iterative Methods. Besides Krylov methods, simpler stationary iterative methods like Jacobi, Gauss-Seidel, or SOR exist. These are rarely used on their own for large-scale systems, as they not always converge and if then slowly, but they are important as building blocks in more complex structures.

2.1.2 Using Solvers in PETSc

PETSc is a library for solving large-scale scientific problems and our main project’s dependency [BAA⁺25]. In PETSc, the solver context is managed using the KSP object (even for non-Krylov solvers). Below is a generic example of how to set up a linear solver in PETSc:

```
1 KSP ksp;
2 // Create linear solver context
3 PetscCallVoid(KSPCreate(PETSC_COMM_WORLD, &ksp));
4 PetscCallVoid(KSPSetOperators(ksp, A, A));
5
6 // Choose the solver type
7 PetscCallVoid(KSPSetType(ksp, KSPCG)); // For SPD matrices
8 PetscCallVoid(KSPSetType(ksp, KSPGMRES)); // For general non-symmetric matrices
9 PetscCallVoid(KSPSetType(ksp, KSPMINRES)); // For symmetric indefinite matrices
10
11 // Set additional solver options from command line or options database
12 PetscCallVoid(KSPSetFromOptions(ksp));
13
14 // Solve the linear system Ax = b
15 PetscCallVoid(KSPSolve(ksp, b, x));
```

Listing 1: Creating and configuring a PETSc Krylov solver

These commands initialize the solver and associate it with the matrix A . The function `KSPSetFromOptions` allows for command-line options.

2.2 Preconditioners

Given a linear system

$$Ax = b, \tag{14}$$

we solve the modified system

$$P^{-1}Ax = P^{-1}b, \quad (15)$$

where P is a preconditioner chosen to make $P^{-1}A$ better conditioned than A itself. One can note that $P = A$ would be the optimal preconditioner, leading to the minimal condition number of 1 for $I = A^{-1}A$. However, this would require solving the system in order to solve the system. [She94]

Note. The inverse of a matrix is a mathematical notation. In practice, computing it explicitly is inefficient. Instead, the implementation involves solving a linear system.

Goals. The condition number of a matrix affects both the convergence rate and numerical stability of iterative methods. For example, the error $e_k = x_k - x^*$ at iteration k of the CG algorithm satisfies the bound:

$$\|e_k\|_A \leq 2\|e_0\|_A \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k, \quad (16)$$

where $\|e\|_A = \sqrt{e^T A e}$ is the A -norm, and $\kappa(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}$ is the condition number of A . This result shows that a smaller condition number $\kappa(A)$ leads to faster convergence. The numerical stability of solving $Ax = b$ is also influenced by the condition number $\kappa(A)$, which controls the sensitivity of the solution to perturbations in b or round-off errors:

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A) \frac{\|\delta b\|}{\|b\|}. \quad (17)$$

By choosing a preconditioner such that $\kappa(P^{-1}A) \ll \kappa(A)$, one reduces the impact of numerical errors, improving both stability and accuracy of the solution in an inexact floating-point arithmetic setting.

Interpretation. A preconditioner can be seen as:

- An approximation of a matrix that has an easier-to-calculate inverse.
- A one-step application of an iterative solver.
- A scaling of the eigenvalues.

2.2.1 Common Preconditioners

Below are some commonly used preconditioners:

- **Jacobi:** Works similar to the respective stationary iterative method and utilizes only the diagonal elements of the matrix A , making it simple and inexpensive to implement. It's most effective when A is diagonally dominant. However, its simplicity often results in limited improvement in convergence rates.
- **Gauss-Seidel:** Works similar to the respective stationary iterative method and incorporates both the diagonal and lower triangular parts of A , leading to better convergence properties than the Jacobi method. It updates solution components sequentially, using the latest available values.
- **Incomplete LU (ILU) Factorization:** Approximates the LU decomposition of A without computing the exact factors, allowing for controlled fill-in and preserving sparsity. ILU is particularly useful for general sparse matrices.
- **Incomplete Cholesky (ICC) Factorization:** Similar to ILU but tailored for symmetric positive definite matrices. It approximates the Cholesky decomposition, maintaining sparsity.

2.2.2 PCFIELDSPLIT

The idea behind the PCFIELDSPLIT preconditioner in PETSc is inspired by classical matrix factorization techniques, such as LU or Cholesky, but applied at the block level. This is particularly useful for systems arising from coupled PDEs, like the Stokes problem, where the global system matrix has a natural 2×2 block structure. By treating different physical variables (e.g., velocity and pressure) as separate fields, PCFIELDSPLIT allows targeted solver and preconditioner strategies for each block. We consider the system of block matrices:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Block Composition Types. The function `PCFieldSplitSetType()` selects how the blocks are combined in the preconditioner:

```
1 #include "petscpc.h"
2 PetscErrorCode PCFieldSplitSetType(PC pc, PCCompositeType type)
```

PC_COMPOSITE_ADDITIVE

This corresponds to a block Jacobi preconditioner (i.e., inverting the diagonal blocks of the matrix):

$$\begin{bmatrix} A^{-1} & 0 \\ 0 & D^{-1} \end{bmatrix}$$

PC_COMPOSITE_MULTIPLICATIVE (default)

This corresponds to a block Gauss-Seidel preconditioner (i.e., inverting the lower triangular part of the block matrix)

$$\begin{bmatrix} I & 0 \\ 0 & D^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} A^{-1} & 0 \\ 0 & I \end{bmatrix}$$

which can also be written as:

$$\begin{bmatrix} I & 0 \\ 0 & D^{-1} \end{bmatrix} \left(\begin{bmatrix} 0 & 0 \\ 0 & I \end{bmatrix} + \begin{bmatrix} I & 0 \\ -C & -D^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} \right) \begin{bmatrix} A^{-1} & 0 \\ 0 & I \end{bmatrix}.$$

PC_COMPOSITE_SYMMETRIC_MULTIPLICATIVE

This corresponds to a symmetric block Gauss-Seidel preconditioner:

$$\begin{bmatrix} A^{-1} & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} I & -B \\ 0 & I \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & D^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} A^{-1} & 0 \\ 0 & I \end{bmatrix}.$$

This is particularly useful for the Conjugate Gradient solver, which requires a symmetric preconditioner.

PC_COMPOSITE_SCHUR

A block matrix can be factorized as follows:

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} I & 0 \\ CA^{-1} & I \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I & A^{-1}B \\ 0 & I \end{bmatrix} = L\Sigma U,$$

where the **Schur complement** of A in M is given by:

$$S = D - CA^{-1}B.$$

Control Over Schur Factorization. To choose which parts of the Schur factorization to use in the preconditioner, use:

```
1 PetscErrorCode PCFieldSplitSetSchurFactType(PC pc, PCFieldSplitSchurFactType ftype)
```

The available options are:

1. `PC_FIELDSPLIT_SCHUR_FACT_DIAG` – The preconditioner applies Σ^{-1} .
2. `PC_FIELDSPLIT_SCHUR_FACT_LOWER` – The preconditioner applies $(L\Sigma)^{-1}$.
3. `PC_FIELDSPLIT_SCHUR_FACT_UPPER` – The preconditioner applies $(\Sigma U)^{-1}$.
4. `PC_FIELDSPLIT_SCHUR_FACT_FULL` (default) – The full factorization corresponds to a direct solver.

The `DIAG` option uses S with its sign flipped to ensure a positive definite preconditioner. This sign flipping can be disabled using:

```
1 PetscErrorCode PCFieldSplitSetSchurScale(PC pc, PetscScalar scale)
```

Approximating the Schur Complement. Since explicitly constructing S is expensive, PETSc allows specifying how to approximate it:

```
1 PetscErrorCode PCFieldSplitSetSchurPre(PC pc, PCFieldSplitSchurPreType ptype, Mat pre)
```

The available options are:

1. **a11** (default) – $S \approx D$, and the preconditioner is obtained from D .
2. **user** – $S \approx \mathbf{pre}$, where **pre** is user-provided.
3. **selfp** – $S \approx D - C(\text{diag}(A))^{-1}B$ (effective only when $\text{diag}(A)$ is a good preconditioner for A).
4. **full** – PCFIELDSPLIT computes S exactly (computationally expensive, used for testing purposes).

3 Implementation

This section describes the implementation of the saddle point problems using PETSc. The framework for assembling the problem, solving it, and analyzing the results is constant, while the preconditioners and solvers should be adapted to the problem at hand. The implementation is modular, with separate utilities for matrix and vector assembly, as well as for solving the system.

3.1 Poisson Equation with PETSc

This project uses PETSc’s MATNEST feature to assemble and solve the saddle point problem. MATNEST allows for the creation of block matrices, which is useful for problems involving coupled systems, such as the following saddle point problem.

3.1.1 Goal

The goal is to solve the Poisson equation with a constraint on the average value over the domain as described in section 1.1.2. The problem is discretized using finite elements, and the resulting system is formulated as a saddle point problem. A flowchart explaining the implementation can be found in figure 2.

3.1.2 Assembling the Saddle Point Problem

The saddle point problem is assembled in the `assembleSaddlePointProblem` function (see ‘PETSCMatUtilities.cpp’, lines 42–79). The steps are:

1. **Poisson Matrix Assembly:** The Poisson matrix A is assembled in the `assemblePoissonMatrix` function (lines 3–20). The matrix is tridiagonal, with values corresponding to the discretized Laplacian operator.
2. **Constraint Matrix Assembly:** The constraint matrix C , which enforces the average value constraint, is assembled in the `assembleConstraintMatrix` function (lines 35–45).
3. **Nested Matrix and Vector:** The block matrix G is created as a MATNEST object using `MatCreateNest`, combining A , C , and C^T . Similarly, the right-hand side vector b is created as a nested vector using `VecCreateNest` combining F (source term) and c (constraint value).

The modular design ensures that we may apply different preconditioners and solvers for each MATNEST block.

3.1.3 Solving the Saddle Point Problem

A diagram depicting the project structure is given in figure 3.1.3. The solution of the saddle point problem is implemented in the `solveSaddleSystemSchur` function (see ‘PETSCSolveUtilities.cpp’, lines 3–73). The function uses PETSc’s Krylov Subspace Solver (KSP) and an outer Schur complement preconditioner. The steps are:

1. Solver and Preconditioner Setup:

- A KSP solver is created and associated with the matrix G (lines 14–15). The Solver type is set to PREONLY (no outer solver).
- The preconditioner is set to PCFIELDSPLIT, which is designed for block matrices (line 20). The Schur complement decomposition is configured with PC_COMPOSITE_SCHUR (line 22) and a full Schur factorization (PC_FIELDSPLIT_SCHUR_FACT_FULL, line 24). PCFeldSplitSetSchur-Pre is set to "full", since computing S in this case is cheap ($D = 0$ and $CA^{-1}B = B^T A^{-1}B$ is one SPD solve and one matrix-vector product).

2. Sub-Solver Configuration:

- The solver for the A -block is set to Conjugate Gradient (CG) (line 37), the recommended solver for a SPD matrix.
- The preconditioner for the A -block is configured to Incomplete Cholesky (ICC). (lines 41–42).
- The solver for the Schur block is set to none (using preonly), since the system is solved directly by the preconditioner.
- The preconditioner for the Schur block is set to Cholesky (CC), whereby any preconditioner could be chosen (since S is 1x1).

3. Solving the System:

- The linear system $Gx = b$ is solved using KSPSolve (line 52).

4. Results and Cleanup:

- The number of iterations and the residual norm are retrieved and printed (line 69).
- All PETSc objects are destroyed to free memory.

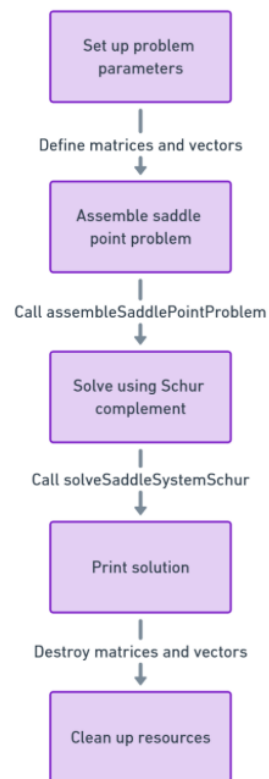


Figure 2: flowchart

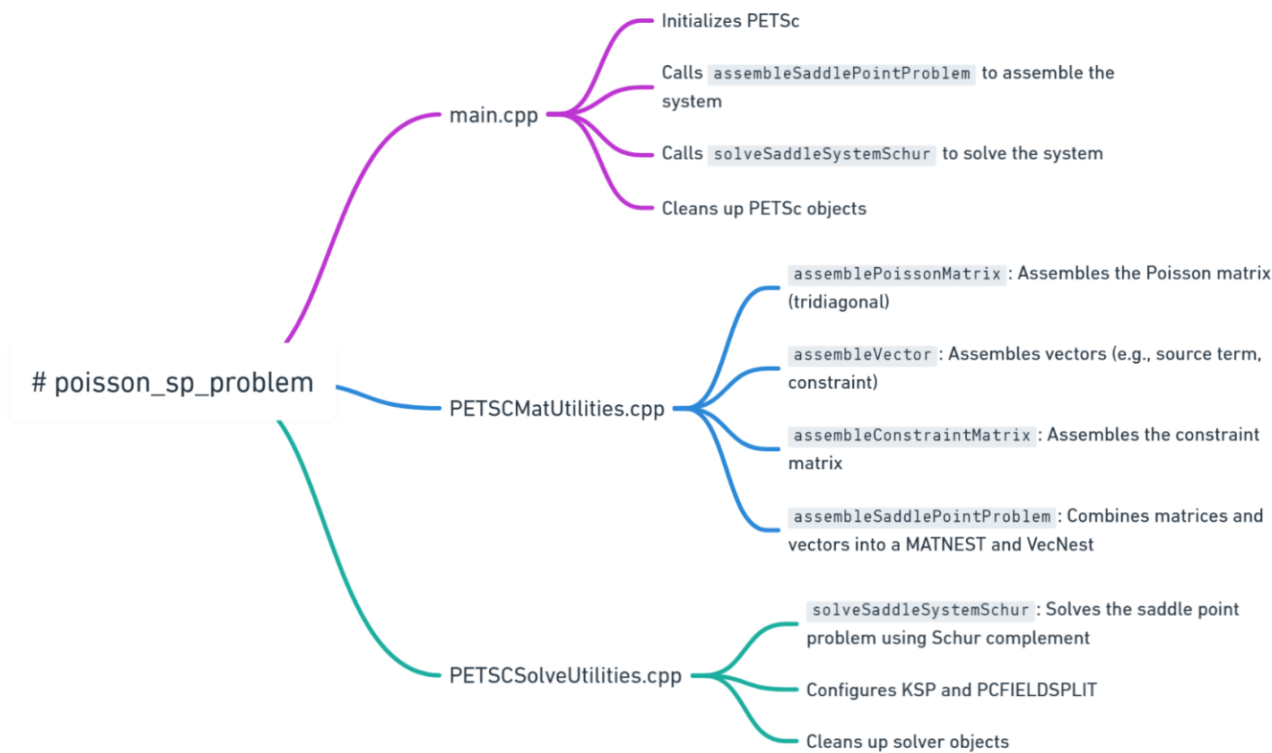


Figure 3: project structure

3.1.4 Results

```

1 >>> mpirun -np 1 ./poisson_sp_problem -ksp_view
2 KSP Object: 1 MPI process
3   type: preonly
4   maximum iterations=10000, initial guess is zero
5   tolerances: relative=1e-05, absolute=1e-50, divergence=10000.
6   left preconditioning
7   using NONE norm type for convergence test
8 PC Object: 1 MPI process
9   type: fieldsplit
10  FieldSplit with Schur preconditioner, blocksize = 1, factorization FULL
11  Preconditioner for the Schur complement formed from the exact Schur complement
12  Split info:
13  Split number 0 Fields 0
14  Split number 1 Fields 1
15  KSP solver for A00 block
16    KSP Object: (fieldsplit_0_) 1 MPI process
17    type: cg
18    maximum iterations=10000, initial guess is zero
19    tolerances: relative=1e-05, absolute=1e-50, divergence=10000.
20    ...
21  PC Object: (fieldsplit_0_) 1 MPI process
22  type: icc
23  ...
24  Mat Object: (fieldsplit_0_) 1 MPI process
25  type: seqaij
26  rows=10, cols=10
27  ...
28  KSP solver for S = A11 - A10 inv(A00) A01
29  KSP Object: (fieldsplit_1_) 1 MPI process
30  type: preonly
31  maximum iterations=10000, initial guess is zero
32  tolerances: relative=1e-05, absolute=1e-50, divergence=10000.
33  ...
34  PC Object: (fieldsplit_1_) 1 MPI process
35  type: cholesky
36  out-of-place factorization
37  tolerance for zero pivot 2.22045e-14
38  ...
39  Factored matrix follows:
40  Mat Object: (fieldsplit_1_) 1 MPI process
41  type: seqdense
42  rows=1, cols=1
43  package used to perform factorization: petsc
44  total: nonzeros=1, allocated nonzeros=1
45  ...
46  type: schurcomplement
47  rows=1, cols=1
48  Schur complement A11 - A10 inv(A00) A01
49  ...
50 Mat Object: (fieldsplit_1_explicit_operator_) 1 MPI process
51 type: seqdense
52 -1.0999999999999999e-01
53 Norm of error: 0., Iterations: 1
54 Vec Object: 1 MPI process
55 type: nest
56 VecNest, rows=2, structure:
57 (0) : type=seq, rows=10
58   Vec Object: 1 MPI process
59   type: seq
60   0.454545
61   0.818182
62   1.09091
63   1.27273
64   1.36364
65   1.36364
66   1.27273
67   1.09091
68   0.818182
69   0.454545
70 (1) : type=seq, rows=1
71   Vec Object: 1 MPI process
72   type: seq
73   -8.09091

```

Summary of KSP View Output The `ksp_view` output confirms the expected configuration explained earlier:

Solver Configuration: The outer solver is configured as `preonly` (line 3), meaning the preconditioner is applied directly without iterations since its already exact. The preconditioner type is set to `fieldsplit` with a Schur complement decomposition (`PC_FIELDSPLIT_SCHUR_FACT_FULL`, line 11), ensuring efficient handling of the block matrix.

Field Splitting: The block size of 1 (line 10) indicates the system is split into two fields:

- **Field 0 (A-block):** Solved using the Conjugate Gradient method (line 17). The preconditioner used is Incomplete Cholesky (ICC) (line 22).
- **Field 1 (Schur complement block):** Solved directly using `preonly` (line 35), with Cholesky as the preconditioner (line 34), since the Schur complement matrix S is scalar (1x1).

Schur Complement Factorization: The Schur complement is explicitly formed as $S = A_{11} - A_{10}A_{00}^{-1}A_{01}$ (line 28), and the chosen full factorization is appropriate because the computation of S is inexpensive.

Results: The system converges in just one iteration (line 53), as expected when using `preonly` with an exact preconditioner. The error norm is reported as 0, indicating a successful and accurate solution. The final result vector (line 54) shows the computed solution for the Poisson problem and the volume constraint. It was matched with MATLABs inbuilt functions' results.

3.2 Stokes Equation with FreeFEM

So far, we have constructed the mesh and assembled the linear system ourselves, relying on external software such as PETSc only to solve the resulting linear system. However, there also exist tools that automate the earlier steps of the PDE pipeline. In this project, we make use of FreeFEM, developed at the Laboratoire Jacques-Louis Lions (Sorbonne University). This software enables us to handle more complex systems, such as the Stokes equation in 2D.

3.2.1 FreeFEM

FreeFEM is a partial differential equation solver designed for nonlinear multiphysics systems in 1D, 2D, and 3D. It is widely used in areas such as fluid dynamics. It comes with its own C++-like scripting language and interpreter, focusing on specifying the what of a PDE problem rather than the how. This declarative style is also found in other mathematical modeling languages. [Hec12]

3.2.2 Goal

We consider the stationary incompressible Stokes equations:

$$\begin{aligned} -\nu\Delta\mathbf{u} + \nabla p &= \mathbf{f} && \text{in } \Omega, \\ \operatorname{div} \mathbf{u} &= 0 && \text{in } \Omega, \\ \mathbf{u} &= \mathbf{g} && \text{on } \partial\Omega, \end{aligned}$$

where $\mathbf{u} = (u_1, u_2)$ denotes the velocity field, p the pressure, $\nu > 0$ the kinematic viscosity, and $\mathbf{f} = (f_1, f_2)$ a given body force (e.g. gravity). This system is a fundamental model for viscous, incompressible flows at low Reynolds numbers. The first equation represents conservation of momentum, while the second enforces mass conservation. The latter is often referred to as the incompressibility constraint. The key modeling assumption is that the flow is sufficiently slow so that convective (nonlinear) effects can be neglected. Such flows arise when the fluid is highly viscous. Examples include the sedimentation of dust in oil, blood flow in capillaries, or the motion of toothpaste. The Stokes equations can be viewed as the linearized, steady-state limit of the more general Navier-Stokes equations. [ESW14]

To derive the variational formulation, we multiply the momentum equation by a test function $\mathbf{v} = (v_1, v_2) \in [H_0^1(\Omega)]^2$ and integrate over the domain:

$$\int_{\Omega} (-\nu\Delta\mathbf{u} + \nabla p) \cdot \mathbf{v} \, dx = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \, dx.$$

We treat each term individually using integration by parts:

Diffusion term:

$$\int_{\Omega} (-\nu \Delta \mathbf{u}) \cdot \mathbf{v} \, dx = \nu \int_{\Omega} \nabla \mathbf{u} \cdot \nabla \mathbf{v} \, dx - \nu \int_{\partial\Omega} (\nabla \mathbf{u} \cdot \mathbf{n}) \cdot \mathbf{v} \, ds,$$

where the boundary term vanishes due to the homogeneous Dirichlet condition on the test function \mathbf{v} .

Pressure term:

$$\int_{\Omega} (\nabla p) \cdot \mathbf{v} \, dx = - \int_{\Omega} p \operatorname{div} \mathbf{v} \, dx + \int_{\partial\Omega} p (\mathbf{v} \cdot \mathbf{n}) \, ds,$$

and again, the boundary integral vanishes because $\mathbf{v} = 0$ on $\partial\Omega$.

Hence, the weak form of the momentum equation becomes:

$$\nu \int_{\Omega} \nabla \mathbf{u} \cdot \nabla \mathbf{v} \, dx - \int_{\Omega} p \operatorname{div} \mathbf{v} \, dx = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \, dx.$$

To weakly impose the incompressibility constraint, we multiply the divergence condition by a test function $q \in L_0^2(\Omega)$ and integrate:

$$\int_{\Omega} q \operatorname{div} \mathbf{u} \, dx = 0.$$

Variational formulation. Find $\mathbf{u} \in \mathbf{g} + [H_0^1(\Omega)]^2$ and $p \in L_0^2(\Omega)$ (unique pressure) such that:

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) &= \langle \mathbf{f}, \mathbf{v} \rangle \quad \forall \mathbf{v} \in [H_0^1(\Omega)]^2, \\ b(\mathbf{u}, q) &= 0 \quad \forall q \in L_0^2(\Omega), \end{aligned}$$

with:

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}) &= \nu \int_{\Omega} \nabla \mathbf{u} : \nabla \mathbf{v} \, dx, \\ b(\mathbf{v}, p) &= - \int_{\Omega} p \operatorname{div} \mathbf{v} \, dx, \\ b(\mathbf{u}, q) &= - \int_{\Omega} q \operatorname{div} \mathbf{u} \, dx. \end{aligned}$$

Saddle-point system. After finite element discretization and change of variables, this leads to the linear system:

$$\begin{bmatrix} A & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u}_h \\ p_h \end{bmatrix} = \begin{bmatrix} \mathbf{f}_h \\ 0 \end{bmatrix},$$

where:

- A is the stiffness matrix associated with the bilinear form $a(\cdot, \cdot)$,
- B is the discrete divergence operator from $b(\cdot, \cdot)$.

The existence, uniqueness, and stability of a solution to the Stokes problem may be proven by the *Babuška-Brezzi theorem* (also known as the *inf-sup condition* or *Ladyzhenskaya-Babuška-Brezzi (LBB) condition*), which provides the necessary framework for analyzing saddle-point systems.

Symmetric Positive-Definite A . The matrix A in the Stokes system arises from the bilinear form

$$a(\mathbf{u}, \mathbf{v}) = \nu \int_{\Omega} \nabla \mathbf{u} \cdot \nabla \mathbf{v} \, dx,$$

which is clearly symmetric, and hence so is A . We show that A is symmetric positive definite (SPD) when restricted to the velocity space $\mathbf{V}_h := [H_0^1(\Omega)]^2$. To prove positive definiteness, we evaluate $a(\mathbf{u}, \mathbf{u})$ for $\mathbf{u} \in \mathbf{V}_h$:

$$a(\mathbf{u}, \mathbf{u}) = \nu \int_{\Omega} |\nabla \mathbf{u}|^2 \, dx.$$

This is zero if and only if $\nabla \mathbf{u} = 0$ almost everywhere in Ω . Since $\mathbf{u} \in [H_0^1(\Omega)]^2$, the Poincaré inequality implies that

$$\|\mathbf{u}\|_{L^2(\Omega)} \leq C_P \|\nabla \mathbf{u}\|_{L^2(\Omega)},$$

where C_P is a constant depending on the domain Ω . Hence, if $\nabla \mathbf{u} = 0$, then $\mathbf{u} = 0$ in $L^2(\Omega)$, implying that $\mathbf{u} = 0$ in the Sobolev space $H_0^1(\Omega)$. Therefore, A is positive definite on \mathbf{V}_h . Although A is SPD, the full Stokes system is indefinite due to the zero block in the pressure-pressure coupling. Nonetheless, the SPD property of A is essential for block preconditioners.

Not Symmetric Positive-Definite S . The matrix S with $\text{selfp} - S \approx D - C(\text{diag}(A))^{-1}B$ – is symmetric but not symmetric-positive definite in general, for the following reasons:

- $D = 0$, $C = B^T$.
- The diagonal entries of a SPD matrix are all positive; therefore, the matrix $\text{diag}(A)^{-1}$ is SPD.
- However, the operator B is not injective in general: the discrete divergence-free subspace

$$\ker B = \{\mathbf{u}_h \in \mathbf{V}_h \mid \text{div } \mathbf{u}_h = 0\}$$

can contain nonzero functions. For example, a velocity field like $\mathbf{u}_h = (1, 1)$ is divergence-free but nonzero, and thus lies in the kernel of B .

3.2.3 Assembling the Saddle Point Problem

The saddle point problem is constructed by first defining the mesh and the finite element spaces for the velocity and pressure fields. The variational formulations are then defined. The matrix system is assembled as a block matrix consisting of the velocity and pressure terms.

We consider a Poiseuille flow with Dirichlet boundary conditions imposed as follows: a parabolic velocity profile is prescribed at the right boundary (outflow), while no-slip conditions (zero velocity) are enforced at the top and bottom boundaries. The left boundary remains unspecified in our configuration, effectively modeling flow through a channel or pipe. The body force \mathbf{f} is set to zero, and the kinematic viscosity ν is taken as 1, simplifying the problem to focus on the effects of boundary-driven flow. [Ric11]

We use the standard Taylor–Hood elements for this problem, which employ \mathbb{P}_2 (quadratic) finite elements for the velocity field and \mathbb{P}_1 (linear) finite elements for the pressure field. This choice is well-established in the context of incompressible flow problems and satisfies the inf-sup condition, which is a fundamental stability criterion for mixed finite element methods.

```

1 load "PETSc"
2 macro dimension()2
3
4 // -- Mesh and finite element spaces
5 mesh Th = square(10, 10);
6 fespace Vh(Th, [P2, P2]); // velocity: vector-valued P2
7 fespace Qh(Th, P1);       // pressure: scalar P1
8
9 // -- Unknowns and test functions
10 Vh [u1, u2], [v1, v2];
11 Qh p, q;
12
13 // -- Variational forms
14 varf aStokes([u1, u2], [v1, v2]) =
15   int2d(Th)(
16     dx(u1)*dx(v1) + dy(u1)*dy(v1)
17     + dx(u2)*dx(v2) + dy(u2)*dy(v2)
18   )
19   + on(1, 3, u1 = 0, u2 = 0)
20   + on(4, u1 = y*(1.0 - y), u2 = 0);
21
22 // -- Variational form for divergence
23 varf bDiv([q], [u1, u2]) = int2d(Th)(-q*(dx(u1) + dy(u2)));
24
25 // -- Assemble system matrices
26 Mat A(Vh.ndof);
27 A = aStokes(Vh, Vh);
28 Mat B(Vh.ndof, Qh.ndof);
29 B = bDiv(Qh, Vh);
30
31 // -- Build block matrix
32 Mat M = [[A, B],
33          [B', 0]];
34
35 // -- Assemble RHS
36 real[int] rhs(Vh.ndof + Qh.ndof);
37 real[int] rhsV = aStokes(0, Vh); // f = 0, only B.C.
38 rhs(0:rhsV.n - 1) = rhsV;       // div(u) = 0

```

Listing 2: Assembling the saddle point problem for the Stokes equation

3.2.4 Solving the Saddle Point Problem

The system is solved using PETSc via its command-line interface, integrated into the FreeFEM script. The code configures the block Schur preconditioner in its full factorization variant (see PCFIELDSPLIT). However, for the interior blocks we use iterative solvers and an inexact Schur complement. As a result, the overall preconditioner is neither linear nor exact, which disqualifies the use of `preonly` as the Krylov solver for the outer system. Instead, we employ `fgmres`, a flexible GMRES variant capable of handling nonlinear preconditioners.

For the A -block (i.e., the (0,0) block corresponding to the velocity unknowns), we use the `cg` method. This is possible because A is SPD. `cg` benefits from a short recurrence relation and avoids the need to store a large Krylov basis. We precondition this system with `icc`, which is a SPD preconditioner, as required by `cg`, and requires the matrix itself to be SPD to be well-defined. Since A is already SPD, no Manteuffel shift is necessary, and we explicitly disable it in the configuration using `pc_factor_shift_type none`.

The Schur complement, on the other hand, is not computed exactly. In fact, attempting to compute the exact Schur complement was not computationally feasible on a laptop with a discretization of (100,100). This matrix is typically dense and no longer SPD, although it remains symmetric in this special setup. The recommended Krylov solver in this case is `minres`, which is tailored for symmetric (but not necessarily definite) matrices. However, `minres` still requires an SPD preconditioner. We therefore apply `icc` again but since the Schur complement is not SPD, we rely on a Manteuffel shift to modify it into a matrix for which `icc` is well-defined.

PETSc applies this shift automatically for non-block matrices when using `icc`, but in this case, we explicitly enabled the shift for clarity, and disabled it where not needed.

```
1 set(M, sparams =
2   "-pc_type fieldsplit -pc_fieldsplit_type schur -pc_fieldsplit_schur_fact_type full -ksp_type fgmres " +
3   "-fieldsplit_0_pc_type icc -fieldsplit_0_pc_factor_shift_type none -fieldsplit_0_ksp_type cg " +
4   "-pc_fieldsplit_schur_precondition selfp " +
5   "-fieldsplit_1_pc_type icc -fieldsplit_1_pc_factor_shift_type positive_definite -fieldsplit_1_ksp_type minres
6   " +
7   "-ksp_view"
8 );
9 // -- Solve with PETSc
10 real[int] sol(rhs.n);
11 KSPSolve(M, rhs, sol);
```

Listing 3: Solving the saddle point problem using PETSc

3.2.5 Results

After solving the system, the velocity and pressure components are extracted from the solution vector. The following code shows how the solution is split and how the plots of the velocity and pressure are generated:

```
1 // -- Extract solution
2 Vh [u1sol, u2sol];
3 Qh psol;
4
5 real[int] VelocityPart(Vh.ndof);
6 real[int] PressurePart(Qh.ndof);
7
8 // Extract the velocity and pressure parts from the solution
9 VelocityPart = sol(0 : Vh.ndof - 1);
10 PressurePart = sol(Vh.ndof : Vh.ndof + Qh.ndof - 1);
11
12 // Separate u1, u2 and p from the solution vector
13 u1sol[] = VelocityPart;
14 psol[] = PressurePart;
15
16 // -- Plot
17 if (mpirank == 0) {
18   plot(Th, [u1sol, u2sol], cmm = "Velocity", WindowIndex=0);
19   plot(Th, psol, cmm = "Pressure", WindowIndex=1);
20 }
```

Listing 4: Extracting and plotting the solution of the saddle point problem

One can observe the characteristic pipe flow behavior: the velocity is maximal in the center of the domain and decreases towards the top and bottom walls due to the no-slip boundary condition. The flow proceeds from left to right, driven by a pressure gradient. Correspondingly, the pressure decreases along the horizontal axis, from the inflow to the outflow boundary.

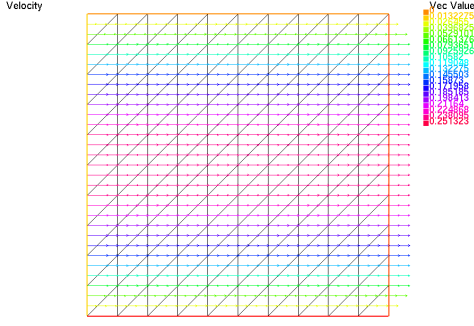


Figure 4: Velocity magnitude of the Stokes solution.

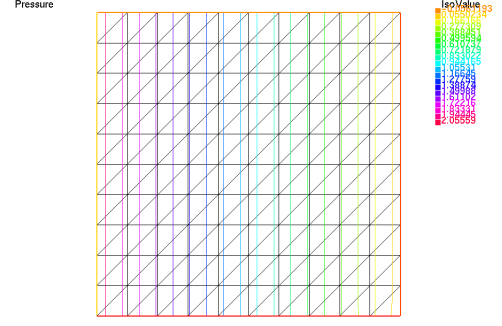


Figure 5: Pressure distribution of the Stokes solution.

We ran the code with:

```
>> /home/FreeFem/bin/ff-mpirun -np 1 stokes.edp -wg -v 0 -malloc_dump -ksp_view_final_residual -
    fieldsplit_0_ksp_converged_reason -fieldsplit_1_ksp_converged_reason
```

Listing 5: Command-line invocation of the FreeFEM script with PETSc options

As a good practice, we always check with `-ksp_view` whether the solver and preconditioners are configured as intended. We also verify convergence of all iterative solvers using `-fieldsplit_0_ksp_converged_reason` and `-fieldsplit_1_ksp_converged_reason`. The final residual is displayed with `-ksp_view_final_residual`. To detect memory leaks, we use `-malloc_dump`, which acts as a Valgrind-like utility for PETSc. FreeFEM's built-in memory leak checker is known to produce false positives, so we suppress its output with `-v 0`. The graphical output is enabled via `-wg`.

```
1 Linear fieldsplit_0_ solve converged due to CONVERGED_RTOL iterations 13
2 ...
3 Linear fieldsplit_1_ solve converged due to CONVERGED_RTOL iterations 44
4 KSP final norm of residual 1.41833e+14
5 ...
```

Listing 6: KSP configuration and solver output from PETSc

We observed that the residual norm is relatively large, which is expected when enforcing homogeneous Dirichlet boundary conditions weakly via a penalty method (default in FreeFEM). However, the relative residual at the order of machine precision:

```
1 // -- Compute relative residual
2 real[int] residual(rhs.n);
3 MatMult(M, sol, residual);          // residual := M * sol
4 residual -= rhs;                    // residual := M*sol - rhs
5
6 real residualNorm = sqrt(residual' * residual); // ||residual||
7 real rhsNorm = sqrt(rhs' * rhs);              // ||rhs||
8 real relativeResidual = residualNorm / rhsNorm;
9
10 cout << "Relative residual: " << relativeResidual << endl;
```

Listing 7: Computing the relative residual

```
1 Relative residual: 1.73709e-16
```

Listing 8: Relative residual of the solution

Penalty Method for Homogeneous Dirichlet Boundary Conditions Let u be the solution of a PDE, and suppose we wish to enforce the condition $u = 0$ on part of the boundary Γ_D . Rather than eliminating the degrees of freedom (DOFs) corresponding to these boundary nodes, the penalty method modifies the system matrix A and right-hand side b as follows: for each DOF i on Γ_D ,

$$A_{ii} \leftarrow A_{ii} + \mathbf{tgv}, \quad b_i \leftarrow b_i + \mathbf{tgv} \cdot 0 = b_i.$$

This strongly penalizes any deviation from zero at the boundary, driving the solution close to the desired value. It is simple to implement, however, if \mathbf{tgv} is chosen too large, the resulting system becomes increasingly ill-conditioned, which may impair the convergence of iterative solvers. Moreover, the method is inherently inexact and can introduce approximation errors. One may also use alternative strategies to enforce Dirichlet boundary conditions by directly modifying the linear system. However, care must be taken to preserve the SPD structure of the system, which is required to use CG.

4 Conclusion

In this project, we investigated efficient methods for solving saddle point problems using PETSc, focusing on the Poisson and Stokes equations. We formulated the weak forms, assembled the systems using MATNEST, and applied the Schur complement preconditioner via PCFIELDSPLIT. To validate our implementation, we compared the Poisson equation results with MATLAB, achieving consistent outcomes. For the Stokes problem, we assembled the system in FreeFEM and used PETSc for solving, employing FGMRES as the outer solver, CG for the velocity block, and MINRES for the Schur complement block.

Throughout the project, we gained hands-on experience with professional numerical software and observed both the conveniences and challenges that arise when one is not fully in control of the computational pipeline, as it is often the case when interfacing with large, modular software systems rather than writing a self-contained Python script.

A natural next step is to move from the position of an end user to that of a contributor by looking inside the “black box.” In particular, we discussed and analyzed potential improvements to the part of PETSc we primarily used, namely PCFIELDSPLIT. PETSc supports solving with multiple right-hand sides simultaneously (i.e., solving $AX = B$ where B is a matrix with several column vectors). The routine `KSPMatSolve` enables this, provided that the preconditioner is also capable of processing multiple right-hand sides.

Currently, however, PCFIELDSPLIT applies its logic independently to each column, without exploiting any potential gains from batched operations. Thus, there is no performance benefit from using the matrix version of the solve. The proposed direction is to examine every operation within the field split logic and identify which ones can be lifted to their matrix equivalents, e.g., replacing `KSPSolve` with `KSPMatSolve`, `MatMult` with `MatMatMult`, and `VecScale` with `MatScale`.

This perspective opens the door for further development and contributions toward optimizing PETSc’s capabilities for block and batched computations in the context of saddle point problems.

1D Example Derivation

We consider the domain $\Omega = (0, 1)$ and discretize it using a uniform grid with $N + 1$ nodes at

$$x_i = ih, \quad i = 0, \dots, N, \quad \text{where } h = \frac{1}{N}.$$

The finite element space V_h consists of piecewise linear basis functions $\{\phi_i\}_{i=1}^{N-1}$, where each ϕ_i is supported on the interval $[x_{i-1}, x_{i+1}]$ and satisfies the nodal conditions:

$$\phi_i(x_j) = \delta_{ij}, \quad 1 \leq i, j \leq N-1.$$

Each function $\phi_i(x)$ is defined as:

$$\phi_i(x) = \begin{cases} \frac{x-x_{i-1}}{h}, & x \in [x_{i-1}, x_i], \\ \frac{x_{i+1}-x}{h}, & x \in [x_i, x_{i+1}], \\ 0, & \text{otherwise.} \end{cases}$$

Taking derivatives:

$$\phi_i'(x) = \begin{cases} \frac{1}{h}, & x \in [x_{i-1}, x_i], \\ -\frac{1}{h}, & x \in [x_i, x_{i+1}], \\ 0, & \text{otherwise.} \end{cases}$$

The entries of the stiffness matrix A are given by:

$$A_{ij} = \int_0^1 \phi_i'(x) \phi_j'(x) dx.$$

Since $\phi_i(x)$ is supported only on $[x_{i-1}, x_{i+1}]$, we compute:

1. Diagonal terms ($i = j$)

$$A_{ii} = \int_{x_{i-1}}^{x_i} \left(\frac{1}{h}\right)^2 dx + \int_{x_i}^{x_{i+1}} \left(-\frac{1}{h}\right)^2 dx.$$

Evaluating:

$$A_{ii} = \frac{1}{h^2}(h + h) = \frac{2}{h}.$$

2. Off-diagonal terms ($i = j \pm 1$)

$$A_{i,i+1} = A_{i+1,i} = \int_{x_i}^{x_{i+1}} \left(-\frac{1}{h}\right) \left(\frac{1}{h}\right) dx.$$

Evaluating:

$$A_{i,i+1} = -\frac{1}{h^2} \int_{x_i}^{x_{i+1}} dx = -\frac{1}{h}.$$

Thus, the stiffness matrix is tridiagonal:

$$A = \frac{1}{h} \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}.$$

The constraint vector C has entries:

$$C_i = \int_0^1 \phi_i(x) dx.$$

Since $\phi_i(x)$ is supported on $[x_{i-1}, x_{i+1}]$, we compute:

$$C_i = \int_{x_{i-1}}^{x_i} \frac{x - x_{i-1}}{h} dx + \int_{x_i}^{x_{i+1}} \frac{x_{i+1} - x}{h} dx.$$

Evaluating both integrals:

$$C_i = \frac{h}{2} + \frac{h}{2} = h.$$

Thus, the constraint vector is:

$$C = h \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}.$$

The load vector entries are given by:

$$F_i = \int_0^1 f(x) \phi_i(x) dx.$$

For $f(x) = 1$:

$$F_i = \int_{x_{i-1}}^{x_i} \frac{x - x_{i-1}}{h} dx + \int_{x_i}^{x_{i+1}} \frac{x_{i+1} - x}{h} dx.$$

From the previous computation:

$$F_i = \frac{h}{2} + \frac{h}{2} = h.$$

Thus, the load vector is:

$$F = h \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}.$$

References

- [AP25] Nathalie Ayi and Emile Parolin. *Approximation des EDP elliptiques et simulation numérique (MU4MA029)*. Sorbonne Université, Paris, France, 2025.
- [BAA⁺25] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Scott J. Benson, Jed Brown, Paul Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, et al. PETSc/TAO Users Manual. Technical report, Argonne National Laboratory (ANL), Argonne, IL, United States, 2025. ANL-21/39 – Revision 3.23.
- [Dup24] Mi-Song Dupuy. *Calcul scientifique pour les grands systèmes linéaires (MU4MA053)*. Sorbonne Université, Paris, France, 2024.
- [ESW14] Howard C. Elman, David J. Silvester, and Andrew J. Wathen. *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*. Oxford University Press, 2014.
- [Gra25] Stef Graillat. *Algorithmique numérique avancé (MU4IN920)*. Sorbonne Université, Paris, France, 2025.
- [Hec12] Frédéric Hecht. New developments in FreeFEM++. *Journal of Numerical Mathematics*, 20(3–4):251–265, 2012.
- [Ric11] S. M. Richardson. Poiseuille flow, 2011. A-to-Z Guide to Thermodynamics, Heat & Mass Transfer, and Fluids Engineering. Begell House. Article added: 2 February 2011. Last modified: 16 March 2011.
- [She94] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. 1994.
- [Tré25] Emmanuel Trélat. *Optimisation numérique et science des données (MU4MA066)*. Sorbonne Université, Paris, France, 2025.