TestLM, Test Lagrange Multiplier for Applying Dirichelet Conditions with MFEM library

Denis Lachapelle

November 2025

1 Introduction

This document explains using the Lagrange multiplier for applying Dirichlet Boundary Conditions. It start with ex1p.cpp from MFEM library. Example 1 solve the Poisson equation.

$$-\Delta u = 1.0 \tag{1}$$

In weakform

$$\int_\Omega -\Delta u \phi d\Omega = \int_\Omega \phi d\Omega \tag{2}$$

2 How Dirichlet Conditions are Normally Applied

Boundary conditions are applied by projecting the coefficient onto the grid function.

```
ParGridFunction x(&fespace);
x = 0.0;
Vector BCV(pmesh.bdr_attributes.Size());
BCV[0] = -1.0;
BCV[1] = -1.0;
```

```
   BCV[2] = -1.0;
   BCV[3] = -1.0;
   BCV[4] = 1.0;
   BCV[5] = 1.0;
   BCV[6] = 1.0;
   BCV[7] = 1.0;
   PWConstCoefficient BC(BCV);
   x.ProjectBdrCoefficient(BC, ess_bdr);  // <-- sets boundary dofs to g
```

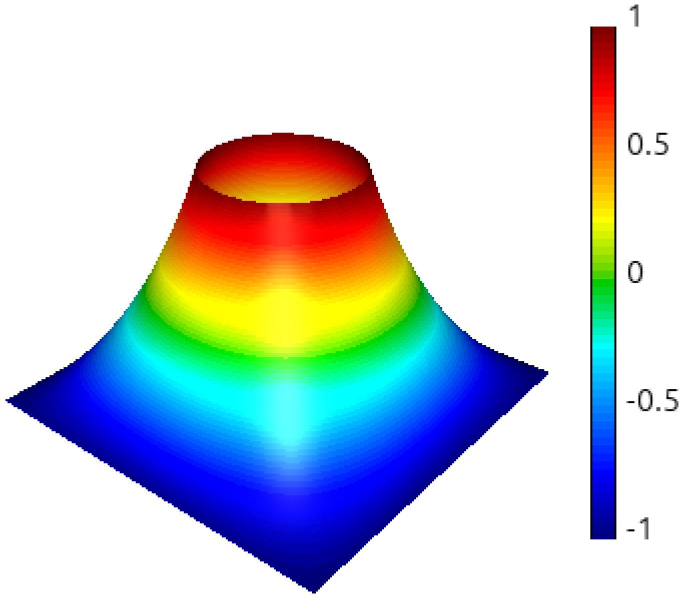
and then call formlinearsystem to adjust the rhs and eliminate row/col for the solver.

```
  FormLinearSystem(ess_tdof_list, x, b, A, X, B);
```

Later re-adjusted with recoversolution(...).

```
  RecoverFEMSolution(X, b, x);
```

The following figure shows boundary conditions -1.0 and 1.0 on the square and circular boundaries.



3 The Dirichlet Boundary Conditions applied with LM

The boundary condition on the square boundary and on the circular boundary: $u_s = BCs$ and $u_c = BCc$.
In weakform it becomes

$$\int_s u\phi = \int_s BCs\phi \tag{3}$$

$$\int_c u\phi = \int_s BCc\phi \tag{4}$$

$$\begin{bmatrix} BLFA & BLFs^T & BLFc^T \\ BLFs & 0 & 0 \\ BLFc & 0 & 0 \end{bmatrix} \begin{bmatrix} U \\ Ls \\ Lc \end{bmatrix} = \begin{bmatrix} LFA \\ LFs \\ LFc \end{bmatrix} \tag{5}$$

The LM will live on submesh, one for square boundary and one for the circular boundary.
Spaces will be made for u over each submesh and for the LM.
TransferMap will be needed between each submesh to the main mesh.
BLFA is a BilinearForm with DiffusionIntegrator with coefficient of +1.0.
BLFs is a MixedBilinearForm with MassIntegrator with coefficient of +1.0.
BLFc is a MixedBilinearForm with MassIntegrator with coefficient of +1.0.
LFA is a LinearForm with DomainLFIntegrator with coefficient +1.0.
LFs is a LinearForm with DomainLFIntegrator with coefficient -1.0.
LFc is a LinearForm with DomainLFIntegrator with coefficient +1.0.

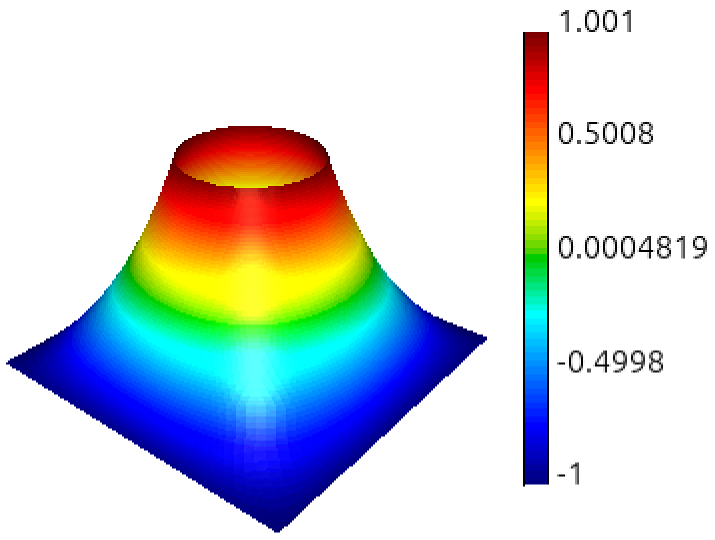
The block matrix above can be written in standard way for LM block matrix.

$$\begin{bmatrix} A & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} U \\ L \end{bmatrix} = \begin{bmatrix} LFA \\ LFL \end{bmatrix} \tag{6}$$

I convert the code from $\underline{\text{parallel to serial}}$ and get a result that is better after some code correction, especially the boundary defining the submesh that were not having the proper attribute.

The following graph was obtain with **"ex01_03.cpp"** without a preconditioner. The result looks quite good even if the convergence was not fully completed after 50000 iterations. FGMRES and GMRES solvers both offer similar results. If I reduce the number of element refinement to less the 1000 instead of 10000 it converge. I try with order 2 and 3 the results are ok but the converge did not reach 10^{-6} after 100000 iterations.

I also try with various order for the field and the Lagrange multiplier.



4 Next step is to make a preconditioner.

4.1 Simple DSmoother diagonal block based preconditioner

That is the code ”ex01_04.cpp”.
I implement a class doing a block diagonal preconditioner using three DSmoother.

DSmoother can be call with various parameters and I try to optimize the parameters to reduce the number of iterations.

849 (0, 0, 0)
351 (1, 0, 10)
126 (1, 0, 100)
70 (1, 0, 1000)

61 (1, 0, 10000)

The class for the simple preconditioner is shown below:

```
/// Block-diagonal preconditioner for
/// [ A   B1^T  B2^T ]
/// [ B1   0     0   ]
/// [ B2   0     0   ]
///
/// P^{-1}  diag(A^{-1}, S1^{-1}, S2^{-1}),
/// using DSmoother.
class ThreeBlockDiagonalPreconditioner : public Solver
{
 private:
 Array<int> block_offsets;
 const SparseMatrix &A, &M1, &M2;
 DSmoother *precA, *precM1, *precM2;
 public:
 ThreeBlockDiagonalPreconditioner(const Array<int> &offsets,
 const SparseMatrix &A_,
 const SparseMatrix &M1_,
 const SparseMatrix &M2_)
 : Solver(offsets.Last()),
 block_offsets(offsets),
 A(A_), M1(M1_), M2(M2_)
 {
  height = width = block_offsets.Last();

  precA = new DSmoother(1, 1.0, 100);
  precM1 = new DSmoother(1, 1.0, 100);
  precM2 = new DSmoother(1, 1.0, 100);

  MFEM_VERIFY(A.Finalized(), "Matrix A must be finalized.");
  MFEM_VERIFY(M1.Finalized(), "Matrix M1 must be finalized.");
  MFEM_VERIFY(M2.Finalized(), "Matrix M2 must be finalized.");

  // --- Build diagonals and DSmoothers (inside the class) ---
  precA->SetOperator(A);
  precM1->SetOperator(M1);
  precM2->SetOperator(M2);
 }

 virtual ~ThreeBlockDiagonalPreconditioner() {}

 virtual void SetOperator(const Operator &op) override
 {
  MFEM_ASSERT(op.Height() == op.Width(), "Operator must be square.");
  MFEM_ASSERT(op.Height() == height, "Operator size must match block offsets.");
 }

 virtual void Mult(const Vector &x, Vector &y) const override
 {
  MFEM_ASSERT(x.Size() == Size(), "Preconditioner input size mismatch.");
  MFEM_ASSERT(y.Size() == Size(), "Preconditioner output size mismatch.");

  // remove const only locally, we won't modify x through xx
  Vector &xx = const_cast<Vector &>(x);
  Vector x0(xx, 0, block_offsets[1]); // primal
  Vector x1(xx, block_offsets[1], block_offsets[2] - block_offsets[1]); // LM1
  Vector x2(xx, block_offsets[2], block_offsets[3] - block_offsets[2]); // LM2

  Vector y0(y, 0, block_offsets[1]); // primal
  Vector y1(y, block_offsets[1], block_offsets[2] - block_offsets[1]); // LM1
  Vector y2(y, block_offsets[2], block_offsets[3] - block_offsets[2]); // LM2

  precA->Mult(x0, y0);
  precM1->Mult(x1, y1);
  precM2->Mult(x2, y2);
 }
};
```

4.2 A better preconditioner

That is the code ”ex01_05.cpp”.

The block matrix is:

$$\begin{bmatrix} A & B1^T & B2^T \\ B1 & 0 & 0 \\ B2 & 0 & 0 \end{bmatrix} \tag{7}$$

The Schur complement of matrix A for the second row $SC1 = -B1A^{-1}B1^T$ and for the third row $SC2 = -B2A^{-1}B2^T$.
The preconditioner will take the form

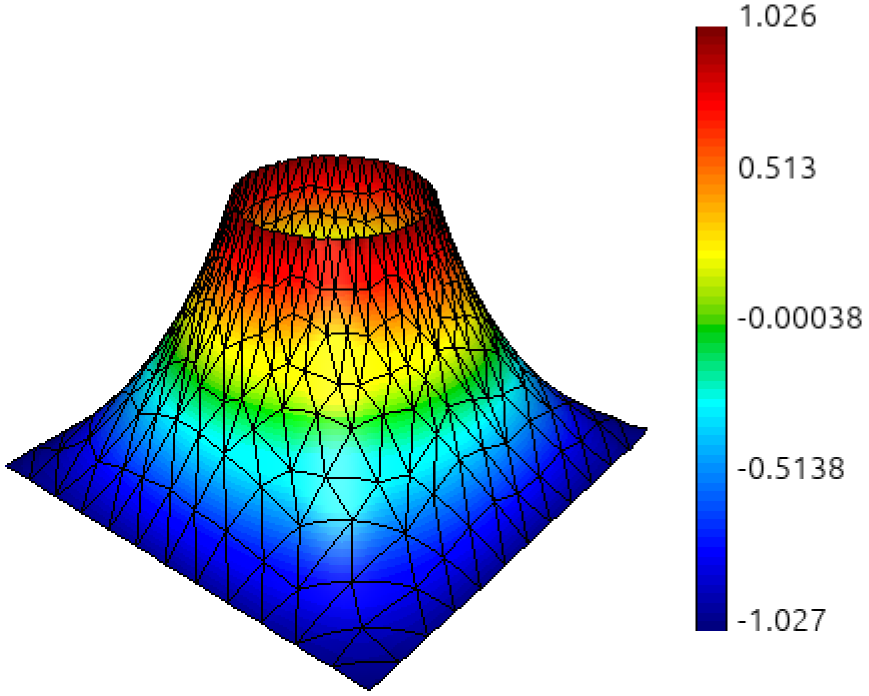
$$\begin{bmatrix} A^{-1} & 0 & 0 \\ 0 & -SC1^{-1} & 0 \\ 0 & 0 & -SC2^{-1} \end{bmatrix} \tag{8}$$

A^{-1} will be approximated with DSmoother or other smoother.

The class InvSchurComp implement $-(BA^{-1}B^T)^{-1}$ as an operator and the class CompleteSchurPreconditioner implement the two Schur based on the approximated A^{-1} and implement the block diagonal operator.

```
/* This class implement an operator applying the inverse of Schur complement.
Ainv_: Operator applying the aproximative inverse of the system matrix.
B_: Operator applying Lagrange multiplier matrix.
*/
class InvSchurComp : public Solver
{
 private:
 Solver &Ainv;
 SubmeshOperator &B;
 TransposeOperator *BT;
 GMRESSolver *solver;
 ProductOperator *AM1BT, *BAM1BT;
 public:
 InvSchurComp(Solver &Ainv_, SubmeshOperator &B_)
 : Solver(B_.Height()), Ainv(Ainv_), B(B_)
 {

  BT = new TransposeOperator(B);
  AM1BT = new ProductOperator(&Ainv, BT, false, false);
  BAM1BT = new ProductOperator(&B, AM1BT, false, false);
  solver = new GMRESSolver();
  solver->SetAbsTol(0);
  solver->SetRelTol(1e-2);
  solver->SetMaxIter(200);
  solver->SetPrintLevel(0);
  solver->SetKDim(20);
  solver->SetOperator(*BAM1BT);

 }

 virtual void Mult(const Vector &b, Vector &x) const override
 {
  x = 0.0;
  solver->Mult(b, x);
 }

 virtual void SetOperator(const Operator &op)
 {

 }
};

class CompleteSchurPreconditioner : public Solver
{
 private:
 InvSchurComp *SC1, *SC2;
 Solver &Ainv;
 SubmeshOperator &B1, &B2;
 int Size;
 const Array<int> &block_offsets;
 public:
 CompleteSchurPreconditioner(const Array<int> &block_offsets_, Solver &Ainv_, Sub
 : Solver(Ainv_.Height() + B1_.Height() + B2_.Height()), Ainv(Ainv_), B1(B1_), B2
 {
  SC1 = new InvSchurComp(Ainv, B1);
  SC2 = new InvSchurComp(Ainv, B2);
  Size = Ainv_.Height() + B1_.Height() + B2_.Height();
 }

 virtual void Mult(const Vector &x, Vector &y) const override
 {
  MFEM_ASSERT(x.Size() == Size(), "Preconditioner input size mismatch.");
  MFEM_ASSERT(y.Size() == Size(), "Preconditioner output size mismatch.");

  // remove const only locally, we won't modify x through xx
  Vector &xx = const_cast<Vector &>(x);
  Vector x0(xx, 0, block_offsets[1]); // primal
  Vector x1(xx, block_offsets[1], block_offsets[2] - block_offsets[1]); // LM1
  Vector x2(xx, block_offsets[2], block_offsets[3] - block_offsets[2]); // LM2

  Vector y0(y, 0, block_offsets[1]); // primal
  Vector y1(y, block_offsets[1], block_offsets[2] - block_offsets[1]); // LM1
  Vector y2(y, block_offsets[2], block_offsets[3] - block_offsets[2]); // LM2

  Ainv.Mult(x0, y0);
  SC1->Mult(x1, y1);
  SC2->Mult(x2, y2);
 }

 virtual void SetOperator(const Operator &op)
 {

 }
};
```

The next figure is produced with command "./ex1_05 -fo 2 -lmo 2 -pt 2"



preconditioner type	**command**	**iterations**	**cpu time**
no preconditioner	./ex1_05 -fo 1 -lmo 1 -pt 0	434	0.018
DSmoother diagonal	./ex1_05 -fo 1 -lmo 1 -pt 1	56	0.021
Schur diagonal	./ex1_05 -fo 1 -lmo 1 -pt 2	39	0.018
no preconditioner	./ex1_05 -fo 2 -lmo 2 -pt 0	2468	0.18
DSmoother diagonal	./ex1_05 -fo 2 -lmo 2 -pt 1	88	0.080
Schur diagonal	./ex1_05 -fo 2 -lmo 2 -pt 2	67	0.13

5 Conclusion

The simple PDE equation systems including the Lagrange Multiplier is converging even without a preconditioner; with the preconditioners the number of iteration get reduced but the CPU time did not reduce much. But at least we prove the preconditioner works, at least they did not destroy the stability.