Fast Solvers for the Finite Element Method

Brian Muldoon & Mike Pozulp

May 9, 2022

Abstract

We solve arbitrarily ill-conditioned linear systems arising from a finite element discretization of a Poisson problem in linear solid mechanics theory using *hypre*. We compare iterative solver convergence and walltimes as condition number increases. We find that *hypre*'s algebraic multigrid preconditioner, used with *hypre*'s conjugate gradient solver, performs best.

Biography

Brian Muldoon and Mike Pozulp are PhD students at UC Berkeley in the Mechanical Engineering Department and the Applied Science & Technology Graduate Program, respectively.

1 Introduction

The elastic deformation of a solid under stress is a process that can be modeled using Poisson's equation. The linear system that arises from a finite element discretization can be ill-conditioned, especially if the body is composed of multiple dissimilar materials. A foam and aluminum body, as in Fig. 1, will be more ill-conditioned than one made of steel and aluminum. The large difference in stiffness between these two materials leads to the stiffness matrix arising in finite element discretizations to have a highly distributed eigenvalue spectrum. The conditioning of the stiffness matrix can have a significant effect on the performance of iterative solvers for systems of algebraic equations.

To solve the linear system of equations for the elastic deformation problem, we use Lawrence Livermore National Laboratory's *hypre*, an open-source library of fast solvers and preconditioners for sparse linear systems [1]. The objective of the project is to explore the iterative solvers available within *hypre* and quantify their performance on the aforementioned computational mechanics problem.

2 FEM Discretization of Elastic Deformation

Consider a 1 dimensional body $\Omega = (0, L)$ with boundary $\partial\Omega = \Gamma_u$ where Γ_u is a portion of the boundary where Dirichlet boundary conditions are applied. The strong form of the 1D Poisson equation for modeling uni-axial beam extension is stated as follows: given Dirichlet boundary

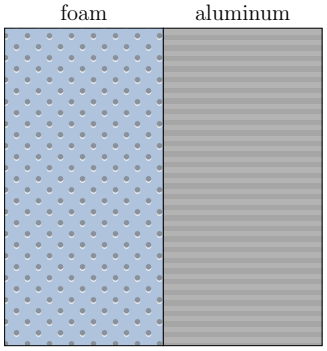


Figure 1: Adjacent foam and aluminum. The elastic deformation of this body will be ill-conditioned relative to a body made of a single material, or two similar materials, like steel and aluminum.

conditions $\bar{u}$ over part of the boundary and material property data EA find $u(x)$ such that,

$$\begin{aligned} EA\frac{d^2u}{dx^2} &= f \quad \text{for} \quad x \in (0, L) \\ u(L) &= \bar{u} \\ u(0) &= 0 \end{aligned} \tag{1}$$

Given the strong form, a weak counterpart is derived by integrating over the domain of a finite element Ω^e and integrating by parts to arrive at

$$\int_{\Omega^e} EA\frac{dw}{dx}\frac{du}{dx}d\Omega = \int_{\Omega_e} wf d\Omega. \tag{2}$$

The weak form objective is then to find $u \in H^1(\Omega)$ in which $u(0) = 0$, $u(L) = \bar{u}$ given $w \in H^1_0(\Omega)$ such that (2) is satisfied. The function $w \in H^1_0(\Omega)$ is a test function with square integrable first derivative which vanishes on the boundary of the domain. The weak form requires that the admissible function u be complete up to polynomial order $p = 1$. Therefore, linear finite element interpolation functions have sufficient approximating power of the solution in Ω. We represent the solution of (2) using a finite basis of linear interpolation functions. The solution with an element domain Ω^e using linear interpolation functions takes the form,

$$u^e \doteq u^e_h = \sum_{i=1}^{2} N^e_i u^e_i \tag{3}$$

where N^e_i is the finite element interpolation function for node i of the element and u^e_i the corresponding nodal degree of freedom. The interpolation functions satisfy the orthogonality condition of a nodal basis such that

$$N_i(x_j) = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases} \tag{4}$$

Assuming a Bubnov-Galerkin approximation, we take the same finite element interpolation functions to construct a basis for the admissible test functions $w \doteq w^e_h = \sum_{i=1}^{2} N^e_i w^e_i$. Substituting these results into the weak form gives,

$$\sum_{i=1}^{2} w_i \left[\int_{\Omega^e} \Big(\sum_{j=1}^{2} EA\frac{dN_i}{dx}\frac{dN_j}{dx}u_j - N_i f \Big) d\Omega \right] = 0 \tag{5}$$

Since w_i are arbitrary, each equation above must independently be zero, yielding the system of equations for each element,

$$[\mathbf{K}^e][\mathbf{u}^e] = [\mathbf{F}^e] \tag{6}$$

where $[\mathbf{K}^e]$ is the element stiffness matrix, $[\mathbf{u}^e]$ is the element degree of freedom vector, and $[\mathbf{F}^e]$ is the applied force vector for an element. The stiffness matrix is given by,

$$[\mathbf{K}^e] = \int_{\Omega^e} EA[\mathbf{B}^e]^T[\mathbf{B}^e]dx \tag{7}$$

where

$$[\mathbf{B^e}] = \begin{bmatrix}[\mathbf{B}_1^e] & [\mathbf{B}_2^e]\end{bmatrix}, \quad [\mathbf{B}_i^e] = \left[\frac{dN_i}{dx}\right] \tag{8}$$

Integration of the element arrays is performed exactly using 2 point Gaussian quadrature. Assembly of the element arrays within the finite element mesh establishes the global system of equations,

$$[\mathbf{K}][\mathbf{u}] = [\mathbf{F}]. \tag{9}$$

Assume that active degrees of freedom are given by u_a and prescribed Dirichlet boundary condition degrees of freedom are given by $\bar{u}$. The system of equations then admits a block decomposition of the form,

$$\begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \begin{bmatrix} u_a \\ \bar{u} \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix}. \tag{10}$$

Solving for the active degrees of the freedom in the system subject to Dirichlet boundary conditions requires solution of the following linear system of equations,

$$K_{11}u_a = F_1 - K_{12}\bar{u}. \tag{11}$$

The conditioning of the K_{11} matrix in equation (11) is dependent on the problem size and material properties of the mechanical system. Also, the dimensionality of the stiffness matrix depends upon the number of finite elements used to construct a piecewise representation of the solution over the domain.

3 Condition Number Analysis

We are interested in understanding the behavior of the stiffness matrix condition number as a function of various parameters. Referring to Fig. 2, as the number of elements in the finite element mesh for the 1D example is increased, the condition number of the stiffness matrix also increases. Additionally, consider the case in which the left side of an elastic bar is given material properties EA_1 and the right side material properties EA_2. Define the material ratio $\alpha = EA_2/EA_1$ as the ratio of the right side stiffness with respect to the left side stiffness. The condition number of the active part of the stiffness matrix adjusted for step-size $K_{11} \cdot h$ is provided in Fig. 2 as a function of element count and material ratio. As expected, the condition number of the stepsize adjusted stiffness matrix increases monotonically with both number of elements and material ratio. However, the condition number of the system increases more rapidly as the problem size grows with increasing number of elements.

Performing the same analysis as above on the 2D Poisson equation gives analogous relationships for the condition number as a function of material ratio and problem size. The stiffness matrix emanating from the 2D Poisson finite element discretization is given by,

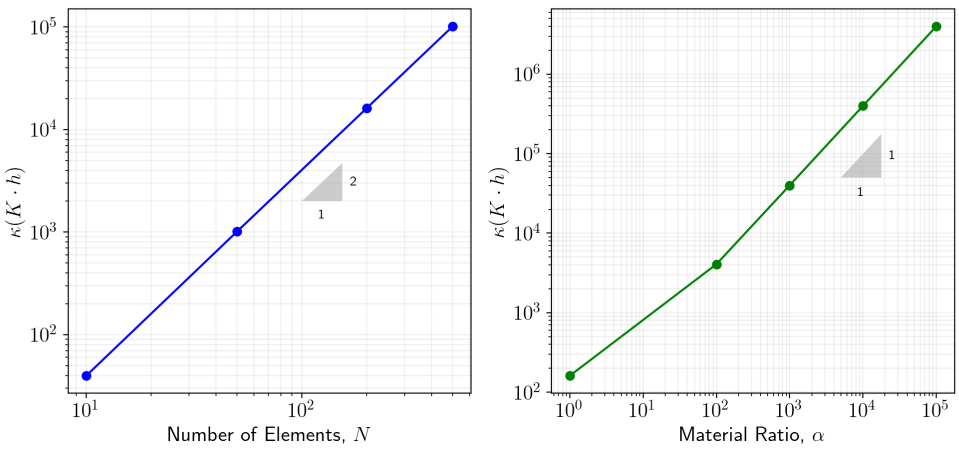


Figure 2: Condition number of the step-size adjusted stiffness matrix $\kappa(K \cdot h)$ from the 1D problem as a function of number of elements (left) and material ratio (right).

$$[\mathbf{K}^e] = \int_{\Omega^e} [\mathbf{B}^e]^T [\mathbf{D}^e][\mathbf{B}^e] d\Omega \tag{12}$$

where the material constitutive matrix $[\mathbf{D}^e]$ arises from a linearly elastic material model. The components of the constitutive matrix for each element are governed by the Lamé parameters λ, μ such that,

$$[\mathbf{D}^e] = \begin{bmatrix} \lambda + 2\mu & \lambda & 0 \\ \lambda & \lambda + 2\mu & 0 \\ 0 & 0 & \mu \end{bmatrix}. \tag{13}$$

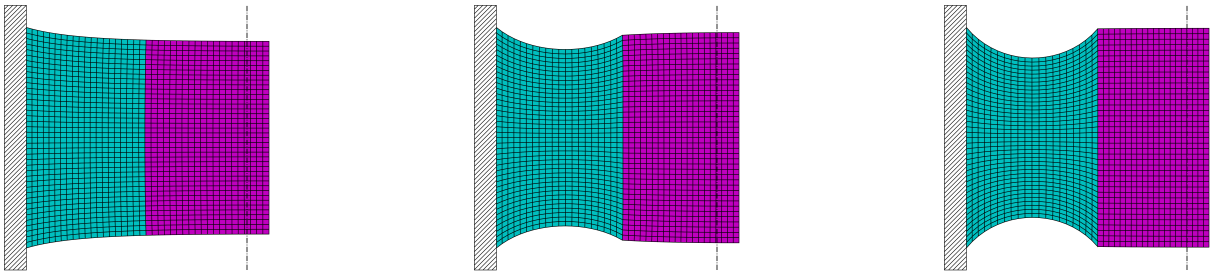
Thus in the 2D case, we define the material ratio α such that for the Lamé parameters of the left (λ_1, μ_1) and right (λ_2, μ_2) parts of the domain,

$$\alpha = \frac{\lambda_2}{\lambda_1}, \qquad \alpha = \frac{\mu_2}{\mu_1}. \tag{14}$$

This material ratio scaling leads to bisected material properties in the body with flexible and stiff properties on the left and right halves of the domain, respectively. The array $[\mathbf{B}^e]$ for the 4 node quadrilateral element is given by,

$$[\mathbf{B}^e] = \begin{bmatrix} \mathbf{B}_1^e & \mathbf{B}_2^e & \mathbf{B}_3^e & \mathbf{B}_4^e \end{bmatrix}, \qquad [\mathbf{B}_i^e] = \begin{bmatrix} \frac{\partial N_i}{\partial x_1} & 0 \\ 0 & \frac{\partial N_i}{\partial x_2} \\ \frac{\partial N_i}{\partial x_2} & \frac{\partial N_i}{\partial x_1} \end{bmatrix} \tag{15}$$

The results in Fig. 3 demonstrates how the deformed configuration of the 2D elastic body changes with respect to increasing material ratio. The material ratio is defined as the ratio of the material properties in the right half (magenta) of the domain with respect to the left half (cyan). Increasing the material ratio monotonically increases the condition number of the stiffness matrix, similar to the 1D example in Fig. 2. The deformed mesh for systems with large material ratios exhibit small deformations in the right half of the domain, relative to the soft material on the left. We use the material ratio to arbitrarily increase the condition number of the linear system of equations that we solve using *hypre*.



(a) $\alpha = 1$ (b) $\alpha = 10$ (c) $\alpha = 100$

Figure 3: Extension of an elastic body for various material ratio subject to boundary conditions $u(0, y) = 0$ and $u(1, y) = \bar{u} = 0.1$. The dashed line indicates the reference length of the material prior to extension. $N = 40$ elements per side for each system. Left half Lamé parameters $\lambda_1 = 1.59\text{E}11$, $\mu_1 = 80\text{E}6$.

4 Stiffness Matrix Analysis

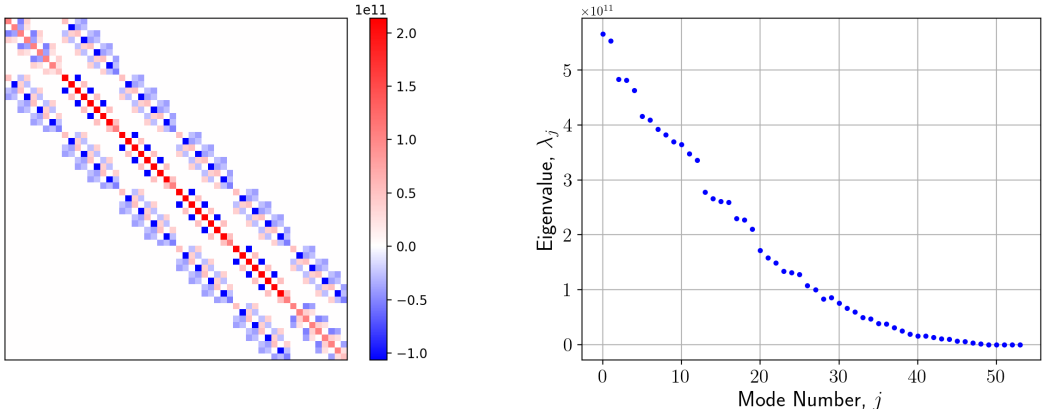
The stiffness matrix arising from the finite element discretization is symmetric and positive semi-definite. The symmetry of the stiffness matrix is apparent from the symmetry of the element array emanating from (12), considering that the constitutive matrix $[\mathbf{D}^e]$ is symmetric for a linearly elastic isotropic material. Additionally, for illustrative purposes, consider a 5×5 finite element mesh composed of 25 bilinear quadrilateral elements. Referring to Fig. 4, the sparsity structure colormap arising from this discretization shows that the system is symmetric. A half bandwidth of 11 was measured for the 54×54 stiffness matrix arising from the 5×5 mesh. Approximately 23% of the entries are non-zero. The stiffness matrix is highly sparse with a narrow bandwidth. The positive semi-definite quality of the stiffness matrix is apparent from the eigenvalue spectrum, shown in Fig. 4b. Upon removal of the rows and columns associated with the Dirichlet boundary conditions, the rigid body modes resulting in zero eigenvalues have been suppressed. Thus, all eigenvalues of the stiffness matrix are non-negative and the matrix is positive semi-definite.

5 Iterative Solvers Using *hypre*

We use *hypre*, a library of fast solvers and preconditioners for sparse linear systems [1]. *hypre* has four different problem description interfaces: Structured-Grid, Semi-Structured-Grid, Finite Element, and Linear-Algebraic ("IJ"). Each *hypre* solver and preconditioner supports a subset of the four interfaces. We use the IJ interface to run multiple *hypre* solvers and preconditioners, which we describe below.

(a) **Conjugate Gradient (cg)**: The CG algorithm [2] chooses x_k which minimizes $||r_k||_{A^{-1}}$, the A^{-1} norm of the residual at iteration k. CG assumes that A is symmetric and positive definite and so A defines a norm $||r||_{A^{-1}} = (r^T A^{-1} r)^{1/2}$.

(b) **Biconjugate Gradient Stabilized (bicgstab)**: The BiCGStab algorithm [3] is a variation on CG which does not require A to be symmetric.



(a) Sparsity diagram for 5 x 5 F.E. mesh (b) Eigenvalue spectrum of the stiffness matrix

Figure 4: Stiffness matrix sparsity plot and eigenvalue spectrum for the 5×5 element mesh. Note: the colormap in (a) indicates the value of the matrix entries.

(c) **Algebraic Multigrid (amg)**: The AMG algorithm [4] is a variation on the multigrid algorithm which does not require information about the discretization grid. Both algorithms use coarsening to convert high-frequency noise in the solution estimate to low-frequency noise, which iterative methods like Jacobi iteration can smooth more efficiently. Multigrid needs the solution grid for interpolation whereas AMG interpolates using only the equations in the linear system.

(d) **Generalized Minimum Residual (gmres)**: The GMRES algorithm [5] chooses x_k which minimizes $||r_k||_2$. GMRES is a variation on MINRES which does not require A to be symmetric.

(e) **Loose Generalized Minimum Residual (lgmres)**: The LGMRES algorithm [6] is a variation on GMRES that attempts to accelerate GMRES convergence by disrupting the cyclic pattern of directions of the residual vectors at the end of each restart cycle of restarted GMRES.

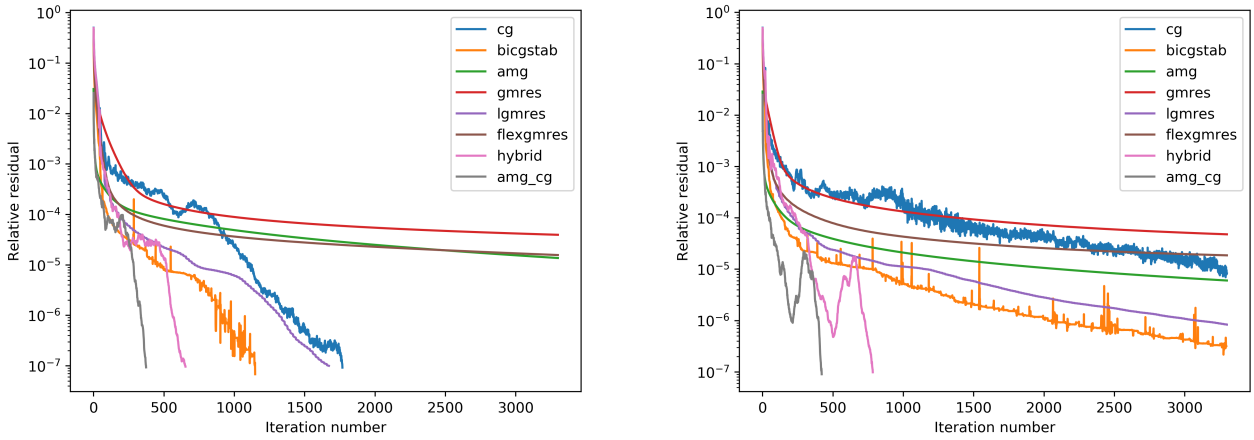
(f) **Flexible Generalized Minimum Residual (flexgmres)**: The FlexGMRES algorithm [7] is a variation on GMRES that allows changes in the preconditioner at every step.

(g) **Hybrid (hybrid)**: *hypre*'s "hybrid" solver assumes a strongly diagonally dominant system, and begins iterating a diagonally scaled Krylov solver without preconditioning. If the convergence rate of the solver falls below a threshold, the algorithm switches to a preconditioned Krylov solver. The solver and preconditioner are arbitrary, but we used *hypre*'s default, which is CG preconditioned using AMG.

(h) **Algebraic Multigrid Preconditioned Conjugate Gradient (amg-cg)**: Use *hypre*'s CG solver with *hypre*'s AMG as the preconditioner.

6 Results

Convergence for a 40-by-40 element system subject to two different material ratios $\alpha = 1, 100$ may be seen in Fig. 5. The walltimes for each of the methods is given in Table 1.



(a) $\alpha = 1$ (b) $\alpha = 100$

Figure 5: Convergence of *hypre* solvers for 40-by-40 element system with material ratio $\alpha = 1$ and $\alpha = 100$. AMG-preconditioned CG converged in the fewest iterations. GMRES, AMG, and FlexGMRES did not converge after 3300 iterations for $\alpha = 1$. Only AMG-preconditioned CG and Hybrid converged for $\alpha = 100$.

solver	setup_walltime	solve_walltime	total_walltime	iterations	time_per_iter
lgmres	0.000371	21.876	21.876371	1673	0.013076
cg	0.000035	25.726	25.726035	1769	0.014543
bicgstab	0.000067	31.747	31.747067	1150	0.027606
hybrid	0.000007	38.315	38.315007	654	0.058586
amg_cg	0.647910	54.632	55.279910	374	0.147807

solver	setup_walltime	solve_walltime	total_walltime	iterations	time_per_iter
hybrid	0.000006	64.036	64.036006	784	0.081679
amg_cg	0.930680	78.745	79.675680	421	0.189253
lgmres	0.000226	99.137	99.137226	6119	0.016202
cg	0.000045	115.690	115.690045	7298	0.015852
bicgstab	0.000055	143.580	143.580055	4440	0.032338

Table 1: Solver walltimes in seconds. The top table is for $\alpha = 1$ and the bottom for $\alpha = 100$. The tables are sorted by "total_walltime" which is the sum of "setup_walltime" and "solve_walltime". The "iterations" column is the number of iterations required to converge. "time_per_iter" is "total_walltime" divided by "iterations". We ran on an Intel Xeon E5-2695 v4 machine. All calculations were serial. We used `MPI_Wtime()` to collect walltimes.

7 Discussion

Referring to Fig. 5, the convergence rates of *BICGSTAB*, *LGMRES*, and *CG* were negatively impacted when the material ratio was increased for the system due to increased ill-conditioning. The *GMRES*, *AMG*, and *FLEXGMRES* methods underperformed their counterpart methods by converging to a relative residual on the order of 10^{-4} after 3300 iterations for both material ratio

cases. The relative residual achieved by the *CG* method after 1000 iterations was on the order of 10^{-5} for both $\alpha = 1, 100$, however the $\alpha = 100$ case required an additional 1500 iterations to achieve a relative residual of similar order as the $\alpha = 1$ system. The *AMGCG* and *HYBRID* methods had superior convergence compared to all other methods, but *AMGCG* slightly outperformed *HYBRID* in the sense of achieving the lowest residual in the fewest number of iterations. Using *AMG* to precondition improved the convergence of *CG* for both material ratio cases by reducing the total number of iterations required to convergence.

Table 1 shows the solver walltimes. For $\alpha = 1$, the un-preconditioned solvers required more iterations to converge, but less walltime. At $\alpha = 100$ the preconditioned solvers not only required fewer iterations to converge, but also less walltime. The walltime per iteration is higher for the preconditioned solvers.

8 Software and Hardware

We used the following software

- CPython 3.10.1
  - NumPy 1.21.5
  - Matplotlib 3.5.1
- Hypre 2.24.0
  - Clang 13.0.0

We wrote the FEM discretization and plotting code in Python. We wrote the code that calls *hypre* to solve the linear system in C. We compiled our C code and *hypre* at `-O2`. The C code reads A using `HYPRE_IJMatrixRead` and b using `HYPRE_IJVectorRead` from files written by our Python code. The C code writes the solution x of the linear system $Ax = b$ using `HYPRE_IJVectorPrint`. Our Python code reads x from the file and plots the solution, which is the deformed mesh. We ran on an Intel Xeon E5-2695 v4 machine. All calculations were serial.

References

[1] R D Falgout and U M Yang. hypre: A library of high performance preconditioners. *European Conference on Parallel Processing*, 2331 LNCS(PART 3):632–641, 2002.

[2] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J Res NIST*, 49(6):409–436, 1952.

[3] H. A. van der Vorst. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

[4] A. Brandt, S. McCormick, and J. Ruge. Algebraic multigrid (AMG) for sparse matrix equations. In D. J. Evans, editor, *Sparsity and its Applications*, pages 257–284 (of x + 338), pub-CAMBRIDGE:adr, 1984. pub-CAMBRIDGE.

[5] Youcef Saad and Martin H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[6] A. H. Baker, E. R. Jessup, and T. Manteuffel. A technique for accelerating the convergence of restarted gmres. *SIAM Journal on Matrix Analysis and Applications*, 26(4):962–984, 2005.

[7] Youcef Saad. A flexible inner-outer preconditioned gmres algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469, 1993.

9 Project Contributions

Brian Muldoon developed the 1D and 2D Poisson equation finite element discretization codes, and created methods for input/output of data from Python to *hypre* formats. Mike Pozulp developed the C code that calls *hypre*. Both added content to the report.

```
#include "boom.hpp"

#include <stdio.h>
#include <string.h>
#include <sys/stat.h>

#include "HYPRE.h"
#include "HYPRE_IJ_mv.h"
#include "HYPRE_parcsr_ls.h"
#include "mpi.h"

#define MAX_NUM_ITER 10000

typedef void (*SolveFunction)(const HYPRE_ParCSRMatrix *,
                              const HYPRE_ParVector *, HYPRE_ParVector *,
                              double, uint8_t, double *, double *);

void strip_suffix(const char *filename, char *truncated) {
    // REMOVE the .00000 suffix, so A.mij.00000 becomes A.mij, which is
    // what the hypre's read function use.
    // For more details see the definition in HYPRE_IJMatrix.c
    size_t length = strnlen(filename, MAX_STRING_LENGTH);
    strncpy(truncated, filename, length);
    truncated[length - 6] = '\0';
}

void read_matrix(const char *filename, HYPRE_IJMatrix *ij_matrix,
                 HYPRE_ParCSRMatrix *parcsr_matrix) {
    char truncated[MAX_STRING_LENGTH];
    strip_suffix(filename, truncated);
    hec(HYPRE_IJMatrixRead(truncated, MPI_COMM_WORLD, HYPRE_PARCSR, ij_matrix));
    hec(HYPRE_IJMatrixGetObject(*ij_matrix, (void **)parcsr_matrix));
}

void read_vector(const char *filename, HYPRE_IJVector *ij_vector,
                 HYPRE_ParVector *par_vector) {
    // REMOVE the 0000 suffix to read it, so IJ.out.b.0000 becomes IJ.out.b
    // For more details see the definition in HYPRE_IJMatrix.c
    char truncated[MAX_STRING_LENGTH];
    strip_suffix(filename, truncated);
    hec(HYPRE_IJVectorRead(truncated, MPI_COMM_WORLD, HYPRE_PARCSR, ij_vector));
    hec(HYPRE_IJVectorGetObject(*ij_vector, (void **)par_vector));
}

void create_x_vector(HYPRE_IJVector *ij_x, HYPRE_ParVector *par_x, int jlower,
                     int jupper) {
    hec(HYPRE_IJVectorCreate(MPI_COMM_WORLD, jlower, jupper, ij_x));
    hec(HYPRE_IJVectorSetObjectType(*ij_x, HYPRE_PARCSR));
    hec(HYPRE_IJVectorInitialize(*ij_x));
    hec(HYPRE_IJVectorAssemble(*ij_x));
    hec(HYPRE_IJVectorGetObject(*ij_x, (void **)par_x));
}

void print_run_info(int num_iterations, double final_res_norm) {
    printf("\n");
    printf("Iterations = %d\n", num_iterations);
    printf("Final Relative Residual Norm = %e\n", final_res_norm);
    printf("\n");
}

void solve_cg(const HYPRE_ParCSRMatrix *parcsr_A, const HYPRE_ParVector *par_b,
              HYPRE_ParVector *par_x, double convergence_tolerance,
              uint8_t verbosity, double *setup_time, double *solve_time) {
```

```
    // Solve using CG

    HYPRE_Solver solver;
    hec(HYPRE_ParCSRPCGCreate(MPI_COMM_WORLD, &solver));
    hec(HYPRE_PCGSetMaxIter(solver, MAX_NUM_ITER));       // max iterations
    hec(HYPRE_PCGSetTol(solver, convergence_tolerance));  // conv. tolerance
    hec(HYPRE_PCGSetTwoNorm(solver,
                            1));  // use the two norm as the stopping criteria
    if (verbosity) {
        hec(HYPRE_PCGSetPrintLevel(solver, 2));  // print solve info
        hec(HYPRE_PCGSetLogging(solver, 1));     // needed to get run info later
    }

    double start = MPI_Wtime();
    hec(HYPRE_ParCSRPCGSetup(solver, *parcsr_A, *par_b, *par_x));
    *setup_time = MPI_Wtime() - start;
    start = MPI_Wtime();
    hec(HYPRE_ParCSRPCGSolve(solver, *parcsr_A, *par_b, *par_x));
    *solve_time = MPI_Wtime() - start;

    int num_iterations;
    double final_res_norm;
    hec(HYPRE_PCGGetNumIterations(solver, &num_iterations));
    hec(HYPRE_PCGGetFinalRelativeResidualNorm(solver, &final_res_norm));
    print_run_info(num_iterations, final_res_norm);

    hec(HYPRE_ParCSRPCGDestroy(solver));
}

void solve_bicgstab(const HYPRE_ParCSRMatrix *parcsr_A,
                    const HYPRE_ParVector *par_b, HYPRE_ParVector *par_x,
                    double convergence_tolerance, uint8_t verbosity,
                    double *setup_time, double *solve_time) {
    // Solve using BiCGStab

    HYPRE_Solver solver;
    hec(HYPRE_ParCSRBiCGSTABCreate(MPI_COMM_WORLD, &solver));
    hec(HYPRE_BiCGSTABSetMaxIter(solver, MAX_NUM_ITER));  // max iterations
    hec(HYPRE_BiCGSTABSetTol(solver,
                             convergence_tolerance));  // conv. tolerance
    if (verbosity) {
        hec(HYPRE_BiCGSTABSetPrintLevel(solver, 2));  // print solve info
        hec(HYPRE_BiCGSTABSetLogging(solver,
                                     1));  // needed to get run info later
    }

    double start = MPI_Wtime();
    hec(HYPRE_ParCSRBiCGSTABSetup(solver, *parcsr_A, *par_b, *par_x));
    *setup_time = MPI_Wtime() - start;
    start = MPI_Wtime();
    hec(HYPRE_ParCSRBiCGSTABSolve(solver, *parcsr_A, *par_b, *par_x));
    *solve_time = MPI_Wtime() - start;

    int num_iterations;
    double final_res_norm;
    hec(HYPRE_BiCGSTABGetNumIterations(solver, &num_iterations));
    hec(HYPRE_BiCGSTABGetFinalRelativeResidualNorm(solver, &final_res_norm));
    print_run_info(num_iterations, final_res_norm);

    hec(HYPRE_ParCSRBiCGSTABDestroy(solver));
}

void solve_amg(const HYPRE_ParCSRMatrix *parcsr_A, const HYPRE_ParVector *par_b,
```

```
                HYPRE_ParVector *par_x, double convergence_tolerance,
                uint8_t verbosity, double *setup_time, double *solve_time) {
    // Solve using AMG

    HYPRE_Solver solver;
    hec(HYPRE_BoomerAMGCreate(&solver));

    if (verbosity) {
        hec(HYPRE_BoomerAMGSetPrintLevel(
            solver, 3)); /* print solve info + parameters */
    }
    hec(HYPRE_BoomerAMGSetOldDefault(
        solver)); /* Falgout coarsening with modified classical interpolaiton */
    hec(HYPRE_BoomerAMGSetRelaxType(solver,
                                    3)); /* G-S/Jacobi hybrid relaxation */
    hec(HYPRE_BoomerAMGSetRelaxOrder(solver, 1)); /* uses C/F relaxation */
    hec(HYPRE_BoomerAMGSetNumSweeps(solver, 1));  /* Sweeps on each level */
    hec(HYPRE_BoomerAMGSetMaxLevels(solver, 20)); /* maximum number of levels */
    hec(HYPRE_BoomerAMGSetTol(solver,
                              convergence_tolerance)); /* conv. tolerance */
    hec(HYPRE_BoomerAMGSetMaxIter(solver, MAX_NUM_ITER));

    double start = MPI_Wtime();
    hec(HYPRE_BoomerAMGSetup(solver, *parcsr_A, *par_b, *par_x));
    *setup_time = MPI_Wtime() - start;
    start = MPI_Wtime();
    hec(HYPRE_BoomerAMGSolve(solver, *parcsr_A, *par_b, *par_x));
    *solve_time = MPI_Wtime() - start;

    int num_iterations;
    double final_res_norm;
    hec(HYPRE_BoomerAMGGetNumIterations(solver, &num_iterations));
    hec(HYPRE_BoomerAMGGetFinalRelativeResidualNorm(solver, &final_res_norm));
    print_run_info(num_iterations, final_res_norm);

    hec(HYPRE_BoomerAMGDestroy(solver));
}

void solve_gmres(const HYPRE_ParCSRMatrix *parcsr_A,
                 const HYPRE_ParVector *par_b, HYPRE_ParVector *par_x,
                 double convergence_tolerance, uint8_t verbosity,
                 double *setup_time, double *solve_time) {
    // Solve using GMRES

    HYPRE_Solver solver;
    hec(HYPRE_ParCSRGMRESCreate(MPI_COMM_WORLD, &solver));
    hec(HYPRE_GMRESSetMaxIter(solver, MAX_NUM_ITER));       // max iterations
    hec(HYPRE_GMRESSetTol(solver, convergence_tolerance));  // conv. tolerance
    if (verbosity) {
        hec(HYPRE_GMRESSetPrintLevel(solver, 2));  // print solve info
        hec(HYPRE_GMRESSetLogging(solver, 1));  // needed to get run info later
    }

    double start = MPI_Wtime();
    hec(HYPRE_ParCSRGMRESSetup(solver, *parcsr_A, *par_b, *par_x));
    *setup_time = MPI_Wtime() - start;
    start = MPI_Wtime();
    hec(HYPRE_ParCSRGMRESSolve(solver, *parcsr_A, *par_b, *par_x));
    *solve_time = MPI_Wtime() - start;

    int num_iterations;
    double final_res_norm;
    hec(HYPRE_GMRESGetNumIterations(solver, &num_iterations));
```

```
    hec(HYPRE_GMRESGetFinalRelativeResidualNorm(solver, &final_res_norm));
    print_run_info(num_iterations, final_res_norm);

    hec(HYPRE_ParCSRGMRESDestroy(solver));
}

void solve_lgmres(const HYPRE_ParCSRMatrix *parcsr_A,
                  const HYPRE_ParVector *par_b, HYPRE_ParVector *par_x,
                  double convergence_tolerance, uint8_t verbosity,
                  double *setup_time, double *solve_time) {
    // Solve using LGMRES

    HYPRE_Solver solver;
    hec(HYPRE_ParCSRLGMRESCreate(MPI_COMM_WORLD, &solver));
    hec(HYPRE_LGMRESSetMaxIter(solver, MAX_NUM_ITER));       // max iterations
    hec(HYPRE_LGMRESSetTol(solver, convergence_tolerance));  // conv. tolerance
    if (verbosity) {
        hec(HYPRE_LGMRESSetPrintLevel(solver, 2));  // print solve info
        hec(HYPRE_LGMRESSetLogging(solver, 1));  // needed to get run info later
    }

    double start = MPI_Wtime();
    hec(HYPRE_ParCSRLGMRESSetup(solver, *parcsr_A, *par_b, *par_x));
    *setup_time = MPI_Wtime() - start;
    start = MPI_Wtime();
    hec(HYPRE_ParCSRLGMRESSolve(solver, *parcsr_A, *par_b, *par_x));
    *solve_time = MPI_Wtime() - start;

    int num_iterations;
    double final_res_norm;
    hec(HYPRE_LGMRESGetNumIterations(solver, &num_iterations));
    hec(HYPRE_LGMRESGetFinalRelativeResidualNorm(solver, &final_res_norm));
    print_run_info(num_iterations, final_res_norm);

    hec(HYPRE_ParCSRLGMRESDestroy(solver));
}

void solve_flexgmres(const HYPRE_ParCSRMatrix *parcsr_A,
                     const HYPRE_ParVector *par_b, HYPRE_ParVector *par_x,
                     double convergence_tolerance, uint8_t verbosity,
                     double *setup_time, double *solve_time) {
    // Solve using FlexGMRES

    HYPRE_Solver solver;
    hec(HYPRE_ParCSRFlexGMRESCreate(MPI_COMM_WORLD, &solver));
    hec(HYPRE_FlexGMRESSetMaxIter(solver, MAX_NUM_ITER));  // max iterations
    hec(HYPRE_FlexGMRESSetTol(solver,
                              convergence_tolerance));  // conv. tolerance
    if (verbosity) {
        hec(HYPRE_FlexGMRESSetPrintLevel(solver, 2));  // print solve info
        hec(HYPRE_FlexGMRESSetLogging(solver,
                                      1));  // needed to get run info later
    }

    double start = MPI_Wtime();
    hec(HYPRE_ParCSRFlexGMRESSetup(solver, *parcsr_A, *par_b, *par_x));
    *setup_time = MPI_Wtime() - start;
    start = MPI_Wtime();
    hec(HYPRE_ParCSRFlexGMRESSolve(solver, *parcsr_A, *par_b, *par_x));
    *solve_time = MPI_Wtime() - start;

    int num_iterations;
    double final_res_norm;
```

```
    hec(HYPRE_FlexGMRESGetNumIterations(solver, &num_iterations));
    hec(HYPRE_FlexGMRESGetFinalRelativeResidualNorm(solver, &final_res_norm));
    print_run_info(num_iterations, final_res_norm);

    hec(HYPRE_ParCSRFlexGMRESDestroy(solver));
}

void solve_hybrid(const HYPRE_ParCSRMatrix *parcsr_A,
                  const HYPRE_ParVector *par_b, HYPRE_ParVector *par_x,
                  double convergence_tolerance, uint8_t verbosity,
                  double *setup_time, double *solve_time) {
    // Solve using Hybrid

    HYPRE_Solver solver;
    hec(HYPRE_ParCSRHybridCreate(&solver));
    hec(HYPRE_ParCSRHybridSetDSCGMaxIter(solver,
                                         MAX_NUM_ITER));  // max iterations
    hec(HYPRE_ParCSRHybridSetPCGMaxIter(solver,
                                        MAX_NUM_ITER));  // max iterations
    hec(HYPRE_ParCSRHybridSetTol(solver,
                                 convergence_tolerance));  // conv. tolerance
    if (verbosity) {
        hec(HYPRE_ParCSRHybridSetPrintLevel(solver, 2));  // print solve info
        hec(HYPRE_ParCSRHybridSetLogging(solver,
                                         1));  // needed to get run info later
    }

    double start = MPI_Wtime();
    hec(HYPRE_ParCSRHybridSetup(solver, *parcsr_A, *par_b, *par_x));
    *setup_time = MPI_Wtime() - start;
    start = MPI_Wtime();
    hec(HYPRE_ParCSRHybridSolve(solver, *parcsr_A, *par_b, *par_x));
    *solve_time = MPI_Wtime() - start;

    int num_iterations;
    double final_res_norm;
    hec(HYPRE_ParCSRHybridGetNumIterations(solver, &num_iterations));
    hec(HYPRE_ParCSRHybridGetFinalRelativeResidualNorm(solver,
                                                       &final_res_norm));
    print_run_info(num_iterations, final_res_norm);

    hec(HYPRE_ParCSRHybridDestroy(solver));
}

void solve_amg_cg(const HYPRE_ParCSRMatrix *parcsr_A,
                  const HYPRE_ParVector *par_b, HYPRE_ParVector *par_x,
                  double convergence_tolerance, uint8_t verbosity,
                  double *setup_time, double *solve_time) {
    // Solve using AMG-preconditioned CG

    // CG solver
    HYPRE_Solver solver;
    hec(HYPRE_ParCSRPCGCreate(MPI_COMM_WORLD, &solver));
    hec(HYPRE_PCGSetMaxIter(solver, 1000));               // max iterations
    hec(HYPRE_PCGSetTol(solver, convergence_tolerance));  // conv. tolerance
    hec(HYPRE_PCGSetTwoNorm(solver,
                            1));  // use the two norm as the stopping criteria
    if (verbosity) {
        hec(HYPRE_PCGSetPrintLevel(solver, 2));  // print solve info
        hec(HYPRE_PCGSetLogging(solver, 1));     // needed to get run info later
    }

    // AMG preconditioner
```

```
    HYPRE_Solver precond;
    hec(HYPRE_BoomerAMGCreate(&precond));
    if (verbosity) {
        hec(HYPRE_BoomerAMGSetPrintLevel(precond,
                                         1));  // print amg solution info
    }
    hec(HYPRE_BoomerAMGSetCoarsenType(precond, 6));
    hec(HYPRE_BoomerAMGSetOldDefault(precond));
    hec(HYPRE_BoomerAMGSetRelaxType(precond, 6));  // Sym G.S./Jacobi hybrid
    hec(HYPRE_BoomerAMGSetNumSweeps(precond, 1));
    hec(HYPRE_BoomerAMGSetTol(precond, 0.0));    // conv. tolerance zero
    hec(HYPRE_BoomerAMGSetMaxIter(precond, 1));  // do only one iteration!

    // Set the PCG preconditioner
    hec(HYPRE_PCGSetPrecond(solver, (HYPRE_PtrToSolverFcn)HYPRE_BoomerAMGSolve,
                            (HYPRE_PtrToSolverFcn)HYPRE_BoomerAMGSetup,
                            precond));

    // Now setup and solve!
    double start = MPI_Wtime();
    hec(HYPRE_ParCSRPCGSetup(solver, *parcsr_A, *par_b, *par_x));
    *setup_time = MPI_Wtime() - start;
    start = MPI_Wtime();
    hec(HYPRE_ParCSRPCGSolve(solver, *parcsr_A, *par_b, *par_x));
    *solve_time = MPI_Wtime() - start;

    // Get run information
    int num_iterations;
    double final_res_norm;
    hec(HYPRE_PCGGetNumIterations(solver, &num_iterations));
    hec(HYPRE_PCGGetFinalRelativeResidualNorm(solver, &final_res_norm));
    print_run_info(num_iterations, final_res_norm);

    // Destroy solver and preconditioner
    hec(HYPRE_ParCSRPCGDestroy(solver));
    hec(HYPRE_BoomerAMGDestroy(precond));
}

int file_is_missing(const char *filename, struct stat *stat_buffer) {
    int status = stat(filename, stat_buffer);
    if (status != 0) {
        fprintf(stderr, "The file %s does not exist\n", filename);
    }
    return status * -1;
}

void print_usage(int argc, char ***argv) {
    const char *myname = *argv[0];
    fprintf(stderr, "usage: %s A b solver verbosity\n\n", myname);
    fprintf(stderr, "A is the coefficient matrix\n");
    fprintf(stderr, "b is the right-hand side\n");
    fprintf(stderr, "solver is one of the following\n");
    fprintf(stderr, "verbosity=0 quiet, verbosity=1 print residuals\n");
    fprintf(stderr, "\tamg_cg\n");
    fprintf(stderr, "\thybrid\n");
    fprintf(stderr, "\tflexgmres\n");
    fprintf(stderr, "\tlgmres\n");
    fprintf(stderr, "\tgmres\n");
    fprintf(stderr, "\tamg\n");
    fprintf(stderr, "\tbicgstab\n");
    fprintf(stderr, "\tcg\n");
    fprintf(stderr, "examples: \n");
    fprintf(stderr, "%s A.hij.00000 b.hij.00000 amg_cg 1\n", myname);
```

```
    fprintf(stderr, "%s A.hij.00000 b.hij.00000 gmres 0\n", myname);
}

SolveFunction parse_args(int argc, char ***argv, char **A_filename,
                         char **b_filename, char **solvername,
                         uint8_t *verbosity) {
    if (argc < 5) {
        print_usage(argc, argv);
        exit(-1);
    }
    *A_filename = (*argv)[1];
    *b_filename = (*argv)[2];
    *solvername = (*argv)[3];
    *verbosity = (uint8_t)atoi((*argv)[4]);

    struct stat stat_buffer;
    int missing_files = 0;
    missing_files += file_is_missing(*A_filename, &stat_buffer);
    missing_files += file_is_missing(*b_filename, &stat_buffer);
    if (missing_files != 0) {
        exit(-1);
    }

    SolveFunction solve_function;
    if (0 == strncmp(*solvername, "amg_cg", MAX_STRING_LENGTH)) {
        solve_function = solve_amg_cg;
    } else if (0 == strncmp(*solvername, "hybrid", MAX_STRING_LENGTH)) {
        solve_function = solve_hybrid;
    } else if (0 == strncmp(*solvername, "flexgmres", MAX_STRING_LENGTH)) {
        solve_function = solve_flexgmres;
    } else if (0 == strncmp(*solvername, "lgmres", MAX_STRING_LENGTH)) {
        solve_function = solve_lgmres;
    } else if (0 == strncmp(*solvername, "gmres", MAX_STRING_LENGTH)) {
        solve_function = solve_gmres;
    } else if (0 == strncmp(*solvername, "amg", MAX_STRING_LENGTH)) {
        solve_function = solve_amg;
    } else if (0 == strncmp(*solvername, "bicgstab", MAX_STRING_LENGTH)) {
        solve_function = solve_bicgstab;
    } else if (0 == strncmp(*solvername, "cg", MAX_STRING_LENGTH)) {
        solve_function = solve_cg;
    } else {
        fprintf(stderr, "solver %s not recognized\n", *solvername);
        print_usage(argc, argv);
        exit(-1);
    }
    return solve_function;
}

int main(int argc, char **argv) {
    char *A_filename;
    char *b_filename;
    char *solvername;
    uint8_t verbosity;
    SolveFunction solve_function = parse_args(
        argc, &argv, &A_filename, &b_filename, &solvername, &verbosity);
    initialize(&argc, &argv);

    HYPRE_IJMatrix ij_A;
    HYPRE_ParCSRMatrix parcsr_A;
    read_matrix(A_filename, &ij_A, &parcsr_A);
    hec(HYPRE_IJMatrixPrint(ij_A, "A.out"));

    HYPRE_IJVector ij_b;
```

```
    HYPRE_ParVector par_b;
    read_vector(b_filename, &ij_b, &par_b);
    hec(HYPRE_IJVectorPrint(ij_b, "b.out"));

    int jlower;
    int jupper;
    hec(HYPRE_IJVectorGetLocalRange(ij_b, &jlower, &jupper));

    HYPRE_IJVector ij_x;
    HYPRE_ParVector par_x;
    create_x_vector(&ij_x, &par_x, jlower, jupper);

    double convergence_tolerance = 1e-7;

    double setup_time = 0;
    double solve_time = 0;
    (*solve_function)(&parcsr_A, &par_b, &par_x, convergence_tolerance,
                      verbosity, &setup_time, &solve_time);
    printf("setup_time solve_time\n");
    printf("%.4e %.4e\n", setup_time, solve_time);

    char solutionname[MAX_STRING_LENGTH];
    snprintf(solutionname, MAX_STRING_LENGTH, "x_%s.out", solvername);
    hec(HYPRE_IJVectorPrint(ij_x, solutionname));
    finalize();
}
```

```
#include <stdio.h>

#include "HYPRE_utilities.h"
#include "mpi.h"

#define mec(status) mpi_error_check(status, __FILE__, __LINE__);
#define hec(status) hypre_error_check(status, __FILE__, __LINE__);

#define MAX_STRING_LENGTH 1024

void mpi_error_check(int status, const char *file, int line) {
    if (status == MPI_SUCCESS) {
        return;
    }
    char message[MAX_STRING_LENGTH];
    int length;
    MPI_Error_string(status, message, &length);
    fprintf(stderr, "MPI error %d: %s %s %d\n", status, message, file, line);
    exit(status);
}

void hypre_error_check(int status, const char *file, int line) {
    if (status == 0) {
        return;
    }
    char message[MAX_STRING_LENGTH];
    HYPRE_DescribeError(status, message);
    fprintf(stderr, "Hypre error %d: %s %s %d\n", status, message, file, line);
    exit(status);
}

void initialize(int *argc, char ***argv) {
    mec(MPI_Init(argc, argv));
    hec(HYPRE_Init());
}

void finalize() {
    mec(MPI_Finalize());
    hec(HYPRE_Finalize());
}
```