A Stokes-Brinkman solver in Julia

Brandon Williams

A thesis presented in partial fullfillment
of the requirements for the degree of
Master of Mathematics

Department of Mathematics
Courant Institute of Mathematical Sciences
New York University
2015

Contents

1 Abstract 2
2 Introduction 3
3 Problems 4
3.1 Stokes Flow 4
3.2 Brinkman Flow 4
3.3 Numerical Results 6
4 Methods 10
4.1 Matrix-free Methods 10
4.1.1 Numerical Results 12
4.2 Multigrid as a Preconditioner 14
4.2.1 Numerical Results 14
4.3 High-performance Computing 15
4.3.1 Numerical Results 15
5 Conclusion and Future Work 16
A
Parallel implementation for matrix-free Laplacian-vector operator 17

1 Abstract

We present a finite element Stokes-Brinkman solver developed in Julia that is matrix-free, parallelized, and easy to read and modify. We compare performance for various methods and parameters, using lid-driven cavity flow (Stokes) and channel flow through obstacles (Brinkman) as test problems.

Keywords: Julia, Stokes, Brinkman, geometric multigrid preconditioner, finite element, matrix-free, parallel

2 Introduction

The finite element method (FEM) has enjoyed great success as arguably the most popular and effective tool for solving boundary value partial differential equations. There are many widely-used FEM libraries, both open-source (e.g., deal.II [1], MILAMIN [2], and FreeFem++ [3]) and proprietary (e.g., COMSOL [4]). Although there are many FEM solvers already in existence, we developed our own for a few reasons:

- Most FEM software is either proprietary or the source good is difficult to understand. Our goal was to develop a library in a high-level language that is easy to read and modify.
- We want to outperform Matlab, the current dominant high-level language for FEA.
- To experiment with Julia, as it is a new language and there are not many FEM libraries written in the language.

Julia is a programming language primarily used for scientific computing that was created in 2012. From the authors[5],

> Julia is a high-level, high-performance dynamic programming language for technical computing, with syntax that is familiar to users of other technical computing environments. It provides a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library.

We chose to develop our software in Julia since it is easily parallelized, syntactically similar to Matlab, and claims to be as fast as C.

Our code is heavily influenced by IFISS [6] and HoMG [7], and many ideas and functions were taken directly from these open-source projects. In the same vein, all of our code is open-source and can be found at `http://bmwilly.github.io/brinkman-stokes/`.

The software employs many modern FEM techniques. It includes matrix-free methods for all operators, implements high-order elements, and is parallelized to run on any architechture with an arbitrary number of processors. Multiple implementations of the solver are included in the software to compare performance modify at will, including versions that use assembled matrices instead of matrix-free methods, that are optimized for serial execution, etc.

We are primarily interested in viscous fluid flows, namely those modeled by the Stokes and Brinkman equations. These equations have far-reaching applications in industry and science.

[1] `https://www.dealii.org/`
[2] `http://www.milamin.org/`
[3] `http://www.freefem.org/ff++/`
[4] `http://www.comsol.com/`
[5] `www.julialang.org/`
[6] `http://www.maths.manchester.ac.uk/~djs/ifiss/`
[7] `http://hsundar.github.io/homg/`

3 Problems

3.1 Stokes Flow

Stokes flow is a highly viscous flow modeled by the classical Stokes equations, a linearization of the famous Navier-Stokes equations. They can be solved by a wide array of linear PDE methods, the most popular method being the FEM. The Stokes model is studied frequently in high-performance computing for its simplicity and wide range of physical applications. The problem is to find velocity $\boldsymbol{u} \in H_0^1(\Omega)^d$ and pressure $p \in L_0^2(\Omega)$ such that

$$\begin{aligned} -\mu\Delta\boldsymbol{u} + \nabla p &= \boldsymbol{f} \text{ in } \Omega, \\ \nabla \cdot \boldsymbol{u} &= 0 \text{ in } \Omega, \end{aligned} \tag{1}$$

where $\Omega \in \mathbb{R}^d$ for $d = 2, 3$, $L_0^2(\Omega)$ is the space of square-integrable functions with homogeneous boundary conditions, and $H_0^1(\Omega)^d$ is the Sobolev space with elements in $L_0^2(\Omega)$. In (1), the first equation corresponds to the conservation of momentum and the second equation corresponds to the incompressibility condition, or conservation of mass. Converting the problem to the discrete case, the problem is to find

$$\begin{aligned} (\nabla\boldsymbol{u}_h, \nabla\boldsymbol{v}_h) - (p_h, \nabla\cdot\boldsymbol{v}_h) &= \boldsymbol{f} \text{ for all } \boldsymbol{v}_h \in X_h \subset H_0^1(\Omega)^d, \\ (q_h, \nabla\cdot\boldsymbol{u}_h) &= 0 \text{ for all } q_h \in M_h \subset L_0^2(\Omega), \end{aligned} \tag{2}$$

where $\boldsymbol{v}_h = [v_x, v_y]^T$ are the vector-valued velocity functions and q_h are the scalar pressure functions. We primarily use the $\boldsymbol{Q_2} - \boldsymbol{P_{-1}}$ element for discretization. The biquadratic quadrilateral element $\boldsymbol{Q_2}$ is a square element with biquadratic basis functions of the form $(ax+by+c)(dx+ey+f)$. Thus, there are nine unknowns per element, one each for the terms $1, x, y, xy, x^2, y^2, x^2y, xy^2,$ and x^2y^2. The discontinuous linear pressure $\boldsymbol{P_{-1}}$ has three degrees of freedom, corresponding to the central node and its x and y derivatives there. The nodal values of the basis functions are evaluated at the Gauss quadrature points. This leads to the construction of a finite element coefficient matrix and the linear system takes the form

$$\begin{bmatrix} \boldsymbol{A} & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix},$$

where A is the vector-Laplacian matrix and B is the divergence matrix.

As a test case, we solve the lid-driven cavity problem. The domain is the two-dimensional unit square with Dirichlet boundary conditions on all sides, and stationary sides except for the top, which has velocity tangent to the cavity. The streamlines are shown in figure (1).

3.2 Brinkman Flow

The Brinkman equations are a modification of the Stokes equations, designed to model viscous flow through heterogeneous and porous media. They were originally proposed by Brinkman in 1947 [7] and are a popular area of research today [14, 16, 17, 21]. They have a wide range of applications across biology,

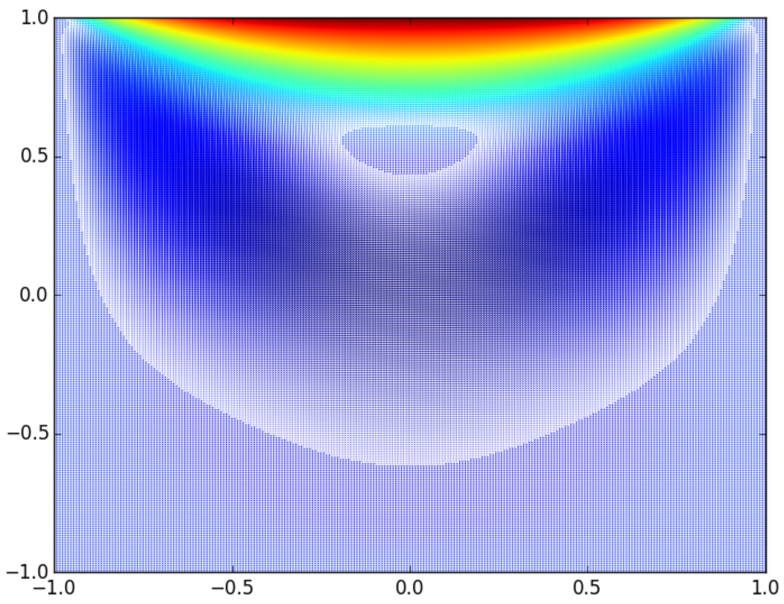


Fig. 1: Solution of lid-driven cavity flow on a 128x128 macroelement grid.

chemistry, and geology, including the modeling of groundwater flow, oil reservoirs, blood vessels, chemical reactions, etc.

The Brinkman (or Stokes-Brinkman) model is as follows. Let $\Omega \in \mathbb{R}^d$ for $d = 2, 3$. We want to find velocity $\boldsymbol{u} \in H_0^1(\Omega)^d$ and pressure $p \in L_0^2(\Omega)$ such that

$$\begin{aligned} -\mu\Delta\boldsymbol{u} + \nabla p + \mu\kappa^{-1}\boldsymbol{u} &= \boldsymbol{f} \text{ in } \Omega, \\ \nabla \cdot \boldsymbol{u} &= 0 \text{ in } \Omega, \\ \boldsymbol{u} &= \boldsymbol{0} \text{ on } \partial\Omega. \end{aligned} \tag{3}$$

As in the Stokes equations, μ is the viscosity and $\boldsymbol{f}$ is the source term. Here, $\kappa^{-1} \in L^\infty(\Omega), \kappa = \kappa(x) > 0$ is the permeability tensor, a measure of the ability of fluid to flow through the media. In general, since κ is a tensor, the permeability at any point affects the permeability at any other point. For many applications, κ can be diagonalized or reduced to a scalar function, significantly simplifying calculations.

The Stokes equations are sufficient for modeling viscous flow in a domain with a few solid obstacles by putting no-slip boundary conditions at the solid interfaces. However, this method becomes impractical if the number of solids in the domain is large, as generating a mesh becomes difficult. E.g., in [19], the authors model a pebble bed reactor with 440,000 solid spheres. On the other hand, the Darcy equations,

$$\begin{aligned} \nabla p + \mu \kappa^{-1} \boldsymbol{u} &= \boldsymbol{f} \text{ in } \Omega, \\ \nabla \cdot \boldsymbol{u} &= 0 \text{ in } \Omega, \end{aligned} \tag{4}$$

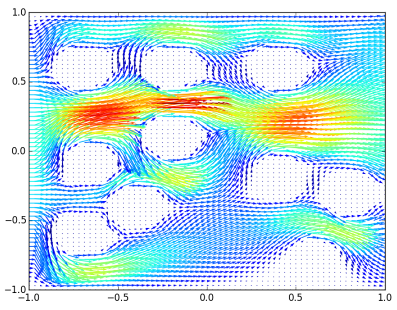
can model flow through porous media, but they are not sufficient to model flow through complicated domains with both porous and free-flow regions.

The coupled Stokes-Darcy model attempts to solve these problems by using the Stokes equations in the fluid regions and the Darcy equations in the porous regions [2, 3, 4, 10], and adding an interface condition between the two regions, e.g., the Beavers-Joseph-Saffman interface condition [4, 8]. However, the Stokes-Darcy equations might not be accurate for many reasons (see [18] and the references therein). First, the permeability of the region needs to be known on a fine scale, which is not usually available and/or too computationally intensive to determine. Second, the interface condition can be difficult or impossible to compute, as the permeability may highly vary across the interface.

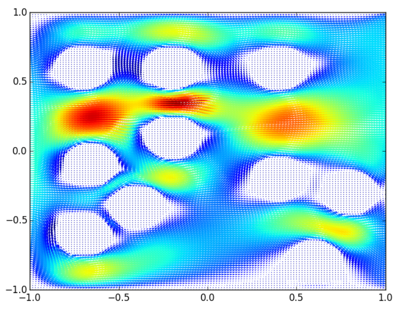
The Brinkman model does not have any of these problems. There is no need for the generation of complicated meshes, as the no-slip boundary conditions are replaced by drag terms in the solid obstacles or porous areas. Furthermore, these features make the Brinkman model easier to implement in a finite element solver. Note that for large κ (i.e., infinite permeability corresponding to the fluid regions), the Brinkman equations behave like the Stokes equations, and for small κ and small μ (i.e., small permeability and viscosity corresponding to pores, obstacles, or cracks), the Brinkman equations behave like the Darcy equations.

3.3 Numerical Results

As a test problem, we will show Brinkman flow in a channel domain with ten obstacles. Here, the permeability is large at nodes that overlap with the obstacles and zero elsewhere. The streamlines become more accurate (i.e., there is no flow through the obstacles) the finer the mesh and the larger the values of κ become. For each mesh size below, κ is defined to be 10^6 at the nodal points that overlap with the obstacles and zero elsewhere.



(a) 32x32 macroelement grid.



(b) 64x64 macroelement grid.

Fig. 2: Brinkman flow through ten obstacles of radius 0.075. $\kappa = 10^6$ at all nodal points that overlap the obstacles.

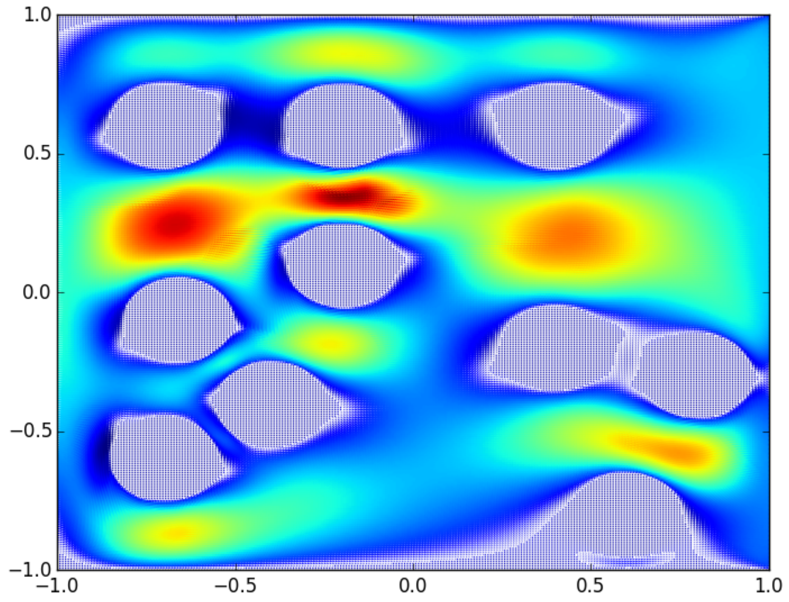


Fig. 3: Brinkman flow on a 128x128 macroelement grid.

Below, we show the normalized logarithm of the residual at each GMRES iteration until convergence is achieved with an error tolerance of 10^{-6}. Note that the size of the problem does not affect the required number of iterations. This occurs only when we use a geometric multigrid preconditioner, discussed in greater detail in section 4.2.

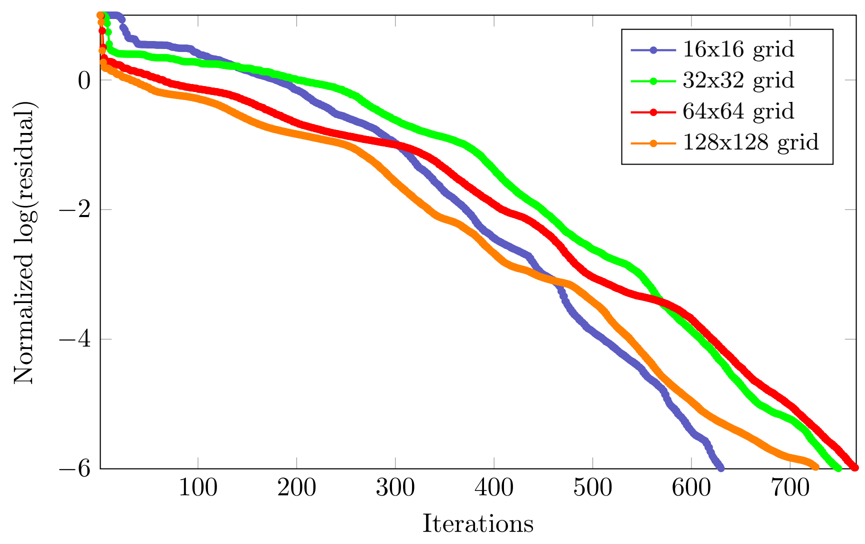


Fig. 4: The number of GMRES iterations to reach the solution with an error tolerance of 10^{-6} for the above Brinkman flow for various mesh sizes.

To demonstrate the ease with which we can solve a large variety of problems using the Brinkman equations, we include the solution for another domain in figure (5). We are able to solve for flow through domains with arbitrary obstacles and levels of porousness by changing one line of code (where we define the centers of the pores/obstacles and the magnitude of κ).

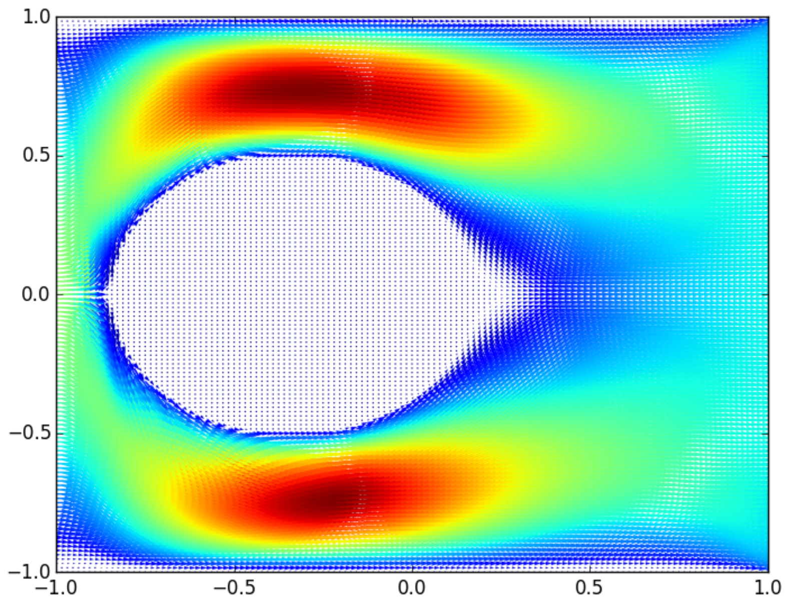


Fig. 5: Brinkman flow with one obstacle on a 64x64 macroelement grid.

4 Methods

In modern FEM solvers, there are many methods to increase the speed and reduce the computing resources needed to find the solution. We implement many of these methods and compare the efficiency improvements that are achieved.

4.1 Matrix-free Methods

The classical FEM setup involves storing the assembled stiffness and mass matrices in memory and then passing these large sparse matrices to a direct or iterative solver. To assemble the global stiffness matrix, one would loop over each element, calculate the element stiffness matrix, and place into a global matrix. The below code shows the general idea for Q_2 elements in two dimensions.

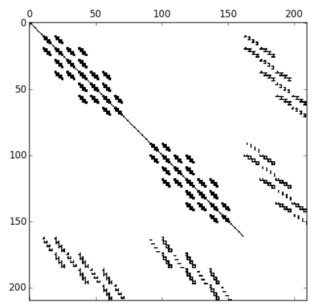
```
# loop over Gauss points
for igpt = 1:nngpt
  # evaluate derivatives, etc.
  (jac, invjac, phi, dphidx, dphidy) = deriv(sigpt, tigpt
      , xlv, ylv)
  (psi, dpsidx, dpsidy) = qderiv(sigpt, tigpt, xlv, ylv)
  (chi, dchidx, dchidy) = lderiv(sigpt, tigpt, xlv, ylv)
  for j = 1:9
    for i = 1:9
      ae[:, i, j] += wght * dpsidx[:, i] .* dpsidx[:, j]
          .* invjac[:]
      ae[:, i, j] += wght * dpsidy[:, i] .* dpsidy[:, j]
          .* invjac[:]
    end
  end
end # end of Gauss point loop

## element assembly into global matrices
# component velocity matrices
for krow = 1:9
  nrow = mv[:, krow]
  for kcol = 1:9
    ncol = mv[:, kcol]
    A += sparse(nrow, ncol, ae[:, krow, kcol], nu, nu)
    A += sparse(nrow + nvtx, ncol + nvtx, ae[:, krow, kcol
        ], nu, nu)
  end
end
```

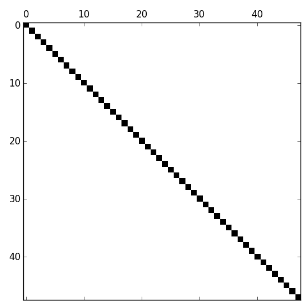
The structures of the assembled global matrices are shown in figure 6.

Any matrix-vector multiplication Cu takes $nm(2nm-1)$ floating point operations (flops), where C is a nxm matrix. Thus, the matrix-vector multiplication $w = Kv$ is $O(n^4)$ since K is square.

Assembling a global stiffness matrix is optimal for small problems; however, for large problems, as the size of the mesh or the order of the basis elements



(a) Global stiffness matrix.



(b) Global mass matrix

Fig. 6: Structures of finite element matrices for an 8x8 grid.

grow, these matrices become too large to store in memory for efficient computations. In this case, we employ matrix-free methods. The basic idea is to construct an operator that behaves like the assembled matrix without ever assembling the matrix. I.e., if $K : \mathbb{R}^n \to \mathbb{R}^n$ is our stiffness matrix, we want to construct a function $k : \mathbb{R}^n \to \mathbb{R}^n$ such that $k(v) = Kv$ for all $v \in \mathbb{R}^n$.
Properties of tensor-product forms allow us to create matrix-free operators that are more efficient than assembling the global matrix. In 2D, if $C = A \otimes B$, then $Cu = (A \otimes I)(I \otimes B)u$, and the number of flops needed is only $2nm(n+m-1)$; that is, $O(n^3)$ if $n = m$. Similarly, in 3D, the assembled matrix is $O(n^6)$ and the tensor-product form is only $O(n^4)$. In general, the tensor-product operator is $O(n^{d+1})$ in dimension d; see [9] for details. Thus, tensor-product operators are more efficient the higher the order and the higher the dimension compared to the assembled matrix.

We implement two different matrix-free methods for each operator (Laplacian, divergence, and mass matrices). For illustrative purposes, we will show the methods for the stiffness matrix only and ignore boundary conditions. The first method takes the local stiffness matrix, applies it directly to the correct indices of the input vector u, and places the result into the correct indices of the output vector w.

```
# loop through elements
for e = 1:ne
  # get nodal points, Jacobian, etc.
                              ⋮
  # calculate stiffness matrix for element
  eMat = element_stiffness(mesh, e, refel, detJac, Jac)
  # get indices corresponding to element
  idx = vec(mv[e, :]')
  # apply local stiffness matrix to input vector and
    store result
  w[idx] += eMat * u[idx]
  w[idx+dof] += eMat * u[idx+dof]
```

```
end
```

The second method is similar, but we reshape u to be a matrix, allowing us to transform the tensor-product matrix-vector multiplications into matrix-matrix products. The advantage here is that the matrix-matrix product can be evaluated using a single, highly optimized routine.

```
# loop through elements
for e = 1:ne
  # get nodal points, Jacobian, etc.
                                      ⋮
  # calculate stiffness matrix for element and store
     result
  eMat = element_stiffness(mesh, e, refel, detJac, Jac)
  eMats[(e-1)*NP+1:e*NP,:] = eMat

  # get indices corresponding to element
  idx = vec(mv[e, :]')
  # reshape input vector into matrix
  Ux[:,e] = u[idx]
  Uy[:,e] = u[idx+dof]
end
# apply the tensor-product form as a matrix-matrix
    product
Wx = eMats * Ux; Wy = eMats * Uy;
# loop through elements and reshape output into a vector
for e = 1:ne
  idx = vec(mv[e, :]')
  w[idx] += Wx[(e-1)*NP+1:e*NP,e]
  w[idx+dof] += Wy[(e-1)*NP+1:e*NP,e]
end
```

4.1.1 Numerical Results

The most time-consuming aspect of setting up a Stokes-Brinkman flow problem and solving it with an iterative method boils down to how fast our program can carry out matrix-vector and matrix-matrix products. Thus, it is reasonable to examine the behavior of these operations for various parameters of our problem. We compare the results of the three methods for different order elements for both 2D and 3D domains in the figures (7) and (8). For each of these tests, we construct the vector-Laplacian operator and record the time it takes to apply the operator to 100 vectors. Tests were carried out on a 2013 Macbook Pro with a 2.3 GHz Intel Core i7 and 16GB memory.

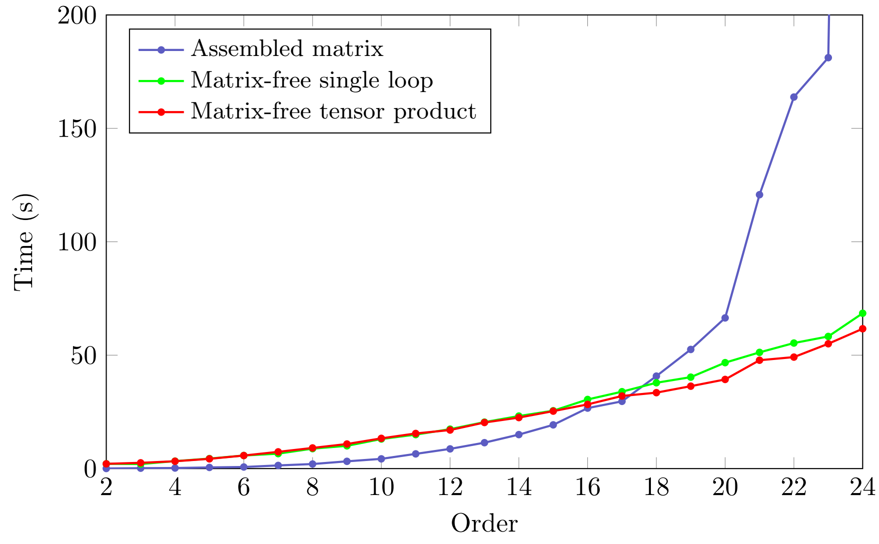


Fig. 7: Time to compute 100 2D vector-Laplacian operations for different orders on a 32x32 macroelement mesh.

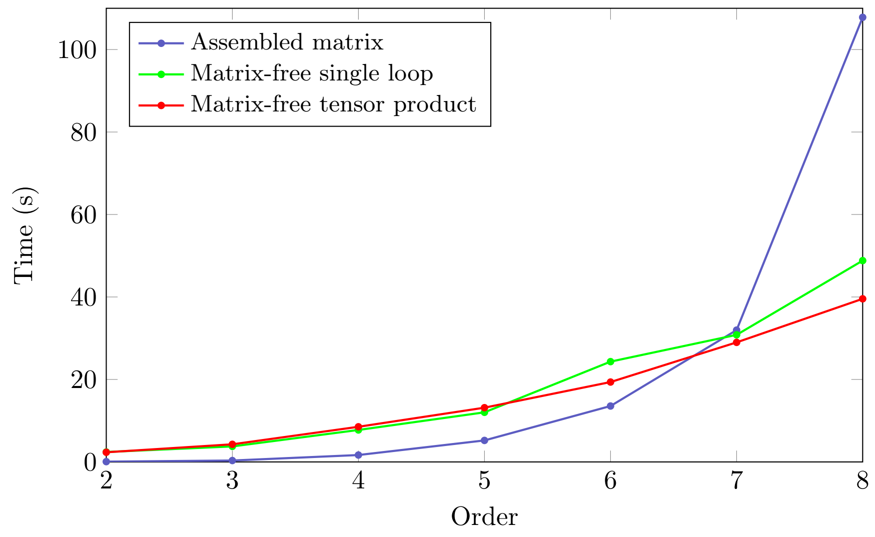


Fig. 8: Time to compute 100 3D vector-Laplacian operations for different orders on an 8x8x8 macroelement mesh.

In other high-level languages like Matlab, it is often recommended to vectorize one's code to increase performance. The opposite is true for Julia, in that loops are often faster than vectorized expressions. This feature suggests that Julia is perfectly suited for matrix-free methods, since the bulk of the computing time is dedicated to looping through the elements, as we see in figure (9).

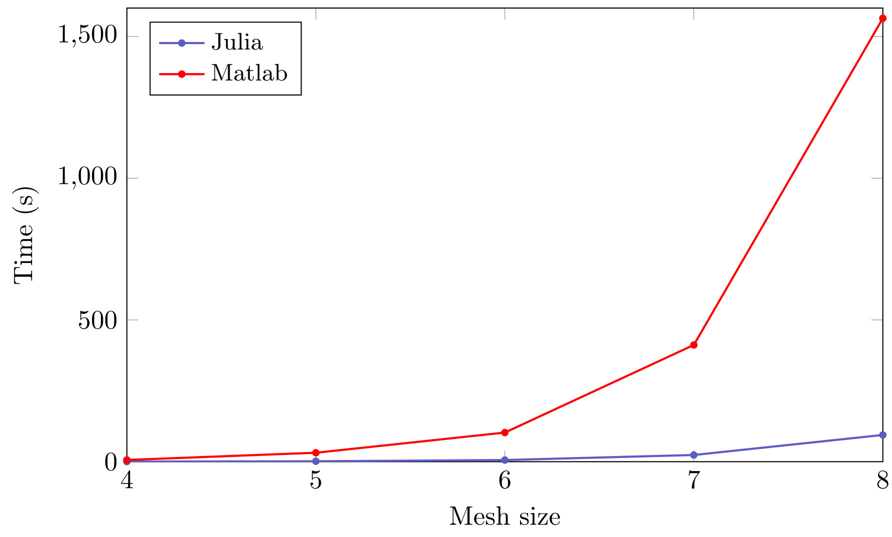


Fig. 9: Time to compute 100 2D vector-Laplacian operations for Q2 elements on different mesh sizes. The numbers n on the x-axis correspond to a mesh of size 2^n x 2^n.

4.2 Multigrid as a Preconditioner

We briefly mention the importance of preconditioning and the preconditioning method that we use for our solver. Multigrid methods are popular in modern computing because of their good performance and their adaptability to parallelization. The main method that we use is geometric multigrid (GMG) with Jacobi smoothing. The idea is to project the fine grid onto a coarser grid, use Jacobi smoothing to reduce the error, repeat for several levels of successively coarser grids, solve the discretized linear system on the coarse grid, and then interpolate the solution back up to the fine grid. Details can be found in a number of sources, e.g. [12]. The most important benefit of using a GMG preconditioner is that the number of iterations needed for convergence does not grow linearly with the problem size, as it does when no preconditioner is used. Although GMG can be used as a solver, for our purposes we employ it strictly as a preconditioner and always use GMRES to solve the system.

4.2.1 Numerical Results

The importance of preconditioning for complicated domains on fine meshes cannot be overstated. For the below plot, every time we use GMG as a preconditioner, we apply three pre-smoothing and three post-smoothing steps.

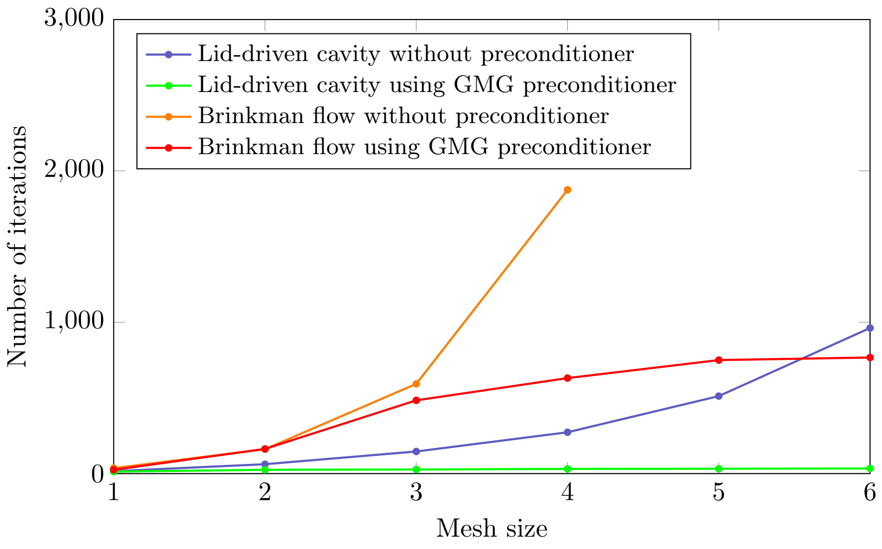


Fig. 10: We solve two of our test problems, lid-driven cavity flow and brinkman flow through ten obstacles, with and without using a GMG preconditioner. The y-axis shows how many iterations of GMRES were needed to reach an error tolerance of 10^{-6}. Note that the number of iterations needed quickly diverges for meshes larger than 8x8 (our solver would not work for Brinkman flow without using a GMG preconditioner for meshes larger than 16x16)

4.3 High-performance Computing

Julia supports distributed parallel execution in the standard library via a message-based system of remote references and remote calls. Parallelization in Julia is more natural and straight forward compared to parallelization in C++ or Matlab, as one is able to call functions that look like "high-level" functions to accomplish many complicated tasks such as synchronization.

Our parallel implementation uses only base Julia. The number of elements is equally distributed across all workers, and each worker computes the tensor-product operations for the subset of elements it is assigned. The parallel implementation of the matrix-free tensor-product matrix-vector Laplacian operator is shown in appendix A.

4.3.1 Numerical Results

We present matvec timings for large grids on 1, 2, 4, and 8 cores. Unsurprisingly, parallelization makes a larger difference the greater the degrees of freedom of the problem. We were not able to achieve perfect parallelization. Timings improve most when adding one extra processor, and we see decreasing marginal utility for each additional processor that we employ.

Mesh size	1 processor	2 processors	4 processors	8 processors
128x128	2.31	2	2.45	2.2
256x256	9.05	6.09	5.69	5.54
512x512	29.61	18.09	14	13.31
1024x1024	116.69	70.05	53.58	47.56

5 Conclusion and Future Work

We have shown that Julia has a lot of potential for high-performance computing FEM applications. We were able to implement modern methods like parallelization and multigrid preconditioners for high-order elements with relative ease compared to low-level languages such as C++ and Fortran, while obtaining faster and more efficient performance compared to Matlab.

Our package is a great starting point for future work which could include solving different problems such as Navier-Stokes flow, extending the code to solve 3D Stokes and Brinkman flow, and developing modules for better visualization and error testing. Additionally, there are opportunities to further speed up the software, including writing a faster GMRES solver, employing further parallelization, etc.

A
Parallel implementation for matrix-free Laplacian-vector operator

```
@everywhere function myrange(mv::SharedArray)
  ind = indexpids(mv)
  if ind == 0
    # this worker is not assigned a piece
    return 1:0
  end
  nchunks = length(procs(mv))
  splits = [iround(s) for s in linspace(0, size(mv, 1),
      nchunks + 1)]
  splits[ind]+1:splits[ind+1]
end

@everywhere function loop_elem!(w, u, mesh, order, dof,
    refel, centers, mv, erange)
  for e in erange
                                         ⋮
    idx = vec(mv[e, :]')
    w[idx] += eMat * u[idx]
    w[idx+nvtx] += eMat * u[idx+dof]
  end
  w
end

@everywhere loop_elem_chunk!(w, u, mesh, order, dof,
    refel, centers, mv) = loop_elem!(w, u, mesh, order,
    dof, refel, centers, mv, myrange(mv))

                                      ⋮

@sync begin
  for p in procs()
    @async remotecall_wait(p, loop_elem_chunk!, w, u, mesh
        , order, dof, refel, centers, mv)
  end
end
```

References

[1] Jorg E. Aarnes, Tore Cimse, and Knut-Andreas Lie, *An Introduction to the Numerics of Flow in Porous Media using Matlab*, Geometric Modelling, Numerical Simulation, and Optimization, pp 205-306, 2007.

[2] Todd Arbogast and Dana S. Brunson, *A Computational Method for Approximating a Darcy-Stokes System Governing a Vuggy Porous Medium*, Computational Geosciences, Volume 11, Issue 3, pp 2017-218, 2007.

[3] Todd Arbogast and Heather L. Lehr, *Homogenization of a Darcy-Stokes system modeling vuggy porous media*, Computational Geosciences, 10:291-302, 2006.

[4] G. Beavers and D. Joseph, *Boundary conditions at a naturally permeable wall*, J. Fluid Mech. 30, pp 197-207, 1967.

[5] D. Braess, *Finite Elements*, Cambridge University Press, London, 1997.

[6] F. Brezzi and M. Fortin, *Mixed and Hybrid Finite Element Methods*, Springer-Verlag, New York, 1991.

[7] H.C. Brinkman, *A Calculation of the Viscous Force Exerted by a Flowing Fluid on a Dense Swarm of Particles*, Appl. Sce. Res. A1, 27-34, 1947.

[8] Yanzhao Cao, Max Gunzburger, Fei Hua, Xiaoming Wang, *Coupled Stokes-Darcy Model with Beavers-Joseph Interface Boundary Condition*, Commun. Math. Sci. Volume 8, Number 1, pp 1-25, 2010.

[9] M.O. Deville, P.F. Fischer, E.H. Mund, *High-Order Methods for Incompressible Fluid Flow*, Cambridge University Press, 2004.

[10] V.J. Ervin, E.W. Jenkins, and Hyesuk Lee, *Approximation of the Stokes-Darcy system by optimization*, Journal of Scientific Computing, Volume 59, Issue 3, pp 775-794, 2014.

[11] A.F. Gulbransen, V.L. Hauge, and K.-A. Lie, *A multiscale mixed finite-element method for vuggy and naturally-fractured reservoirs*, 21 Nordic Seminar on Computational Mechanics, 2008.

[12] Wolfgang Hackbusch, *Multi-Grid Methods and Applications*, Springer-Verlag Berlin Heidelberg, 1985.

[13] Thamir Abdul Hafedh, Prayoto, Fransiskus Soesianto, and Sasongko Pramono Hadi, *Flud Flow In 2-D Petroleum Reservoir Using Darcy's Equation*, Integrasi Teknologi, No. 1, vol. 2, September 2004.

[14] Oleg P. Iliev, Raytcho D. Lazarov, and Joerg Willems, *Discontinuous Galerkin Subgrid Finite Element Method for Heterogeneous Brinkman's Equations*, Large-Scale Scientific Computing, pp14-25, 2010.

[15] Ross Ingram, *Approximating Fast, Viscous Fluid Flow in Complicated Domains*, PhD Thesis, University of Pittsburgh, 2011.

[16] Ingeborg S. Ligaarden, Marcin Krotkiewski, Knut-Andreas Lie, Mayur Pal, and Daniel W. Schmid, *On the Stokes-Brinkman Equations for Modeling Flow in Carbonate Reservoirs*, ECMOR XII - 12th European Conference on the Mathematics of Oil Recovery, 2010.

[17] G. Neale and W. Nader, *Practical significance of Brinkman's extension of Darcy's Law: Coupled parallel flows within a channel and a bounding porous medium*, Can. J. Chem. Eng., 52, pp 475-478, 1974.

[18] Peter Popov, Yalchin Efendiev, and Guan Qin, *Multiscale Modeling and Simulations of Flows in Naturally Fractured Karst Reservoirs*, Commun. Comput. Phys., 6:162-184, 2009.

[19] Chris H. Rycroft, Gary S. Grest, James W. Landry, and Martin Z. Bazant, *Analysis of granular flow in a pebble-bed nuclear reactor*, Physical Review E 74, 021306, 2006.

[20] Hari Sundar, Georg Stadler, George Biros, *Comparison of Multigrid Algorithms for High-order Continuous Finite Element Discretizations*, arXiv:1402.5938, 2014.

[21] Aziz Takhirov, *Stokes-Brinkman Lagrange Multiplier/Fictitious Domain Method for Flows in Pebble Bed Geometries*, PhD Thesis, University of Pittsburgh, 2014.