A scalable matrix-free Stokes-Brinkman solver in Julia

Brandon Williams

New York University, Courant Institute of Mathematical Sciences

251 Mercer Street New York, NY 10012 September 2015

A thesis submitted in partial fullfillment
of the requirements for the degree of
master's of science
Department of Mathematics
New York University
September 2015

Georg Stadler, Faculty advisor

Contents

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

Julia is a programming language used for scientific computing and high-performance numerical linear algebra that was created in 2012. From the authors[1],

> 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.

The finite element method (FEM) has enjoyed great success as a 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[2], MILAMIN[3], FeniCS[4], and FreeFem++[5]) and proprietary (e.g., COMSOL[6]). Our aim is to develop a matrix-free Stokes solver in Julia for the following reasons:

- We want a library in a high-level language that is easy to read and modify.
- To compare performance with Matlab, the current dominant high-level language for numerical linear algebra.
- To experiment with Julia, as it is a new language and there are not many FEM libraries written in the language.

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[7] and HoMG,[8] 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 that enable the solution of very large problems. It includes matrix-free methods for all operators, implements high-order elements, and is able to run on distributed-memory parallel architectures. Multiple implementations of the solver are included in the software to compare the performance and efficiency of different methods. We are primarily interested in viscous fluid flows modeled by the Stokes and Brinkman equations. These equations have far-reaching applications in industry and science and are studied extensively [9, 10, 21, 22, 26].

3 Problems

3.1 Stokes Flow

Stokes flow is a highly viscous flow modeled by the classical Stokes equations, which are the limit of the Navier-Stokes equations when the Reynolds number

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

is small ($Re \ll 1$). They can be discretized by a wide array of PDE methods, including the FEM. The Stokes model is studied frequently in high-performance computing for its simplicity and wide range of physical applications. We mostly follow the formulation from [13], where further details can be found. The problem is to find velocity $\boldsymbol{u} \in H^1(\Omega)^d$ and pressure $p \in L^2(\Omega)$ 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 $d = 2, 3$, $L^2(\Omega)$ is the space of square-integrable functions, and $H^1(\Omega)^d$ is the Sobolev space with elements in $L^2(\Omega)$. The equations additionally require appropriate boundary conditions to be solved. In equation (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 for two-dimensional flow (the three-dimensional case is a straightforward extension), the problem is to find

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

where $(\cdot, \cdot)$ is the L^2-inner product, $\boldsymbol{v}_h = [v_x, v_y]^T$ are the vector-valued velocity functions and q_h are the scalar pressure functions. Suppose the finite element grid has n_u interior nodes and n_b Dirichlet boundary nodes. Then the approximate solution for velocity will be of the form

$$u_h(x, y) = \sum_{i=1}^{n_u} u_i\phi_i(x, y) + \sum_{i=n_u+1}^{n_u+n_b} u_i\phi_i(x, y), \tag{3}$$

where $\{\phi_i\}_{i=1}^{n_u}$ are a set of basis functions for X_h. The coefficients in the second term are chosen so that the equation correctly interpolates the solution on the boundary nodes. Similarly, the approximate solution for pressure will be of the form

$$p_h = \sum_{j=1}^{n_p} p_j\psi_j, \tag{4}$$

where $\{\psi_j\}_{j=1}^{n_p}$ are a set of basis functions for M_h. 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

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

where A is the $n_u \times n_u$ vector-Laplacian matrix and B is the $n_p \times n_u$ divergence matrix.

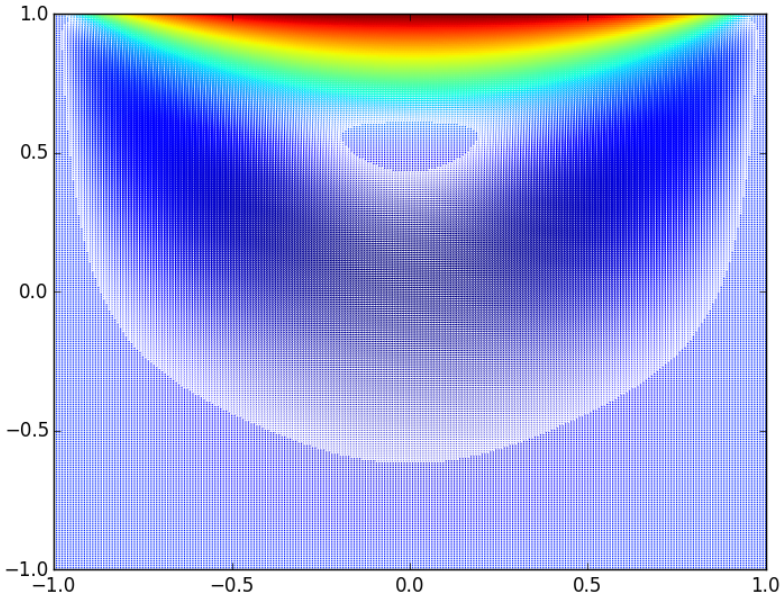


Fig. 1: Solution of lid-driven cavity flow on a 128x128 macroelement grid.

3.1.1 Numerical Results

As a test case, we solve the lid-driven cavity problem, which is an enclosed flow in the two-dimensional square domain $\Omega = (-1, 1)^2$ with Dirichlet boundary conditions. The flow is generated by horizontal velocity on the top side (that is, tangent to the side) of the cavity. We model a so-called "regularized" cavity, with velocity given by

$$y = 1; -1 \leq x \leq 1 \text{ such that } u_x = 1 - x^4. \tag{5}$$

The viscosity is constant, with $\mu \equiv 1$. We find the solution using the geometric multigrid preconditioned (see section (4.2)) iterative solver GMRES [28]. The streamlines are shown in figure (1) for a 128x128 macroelement grid.

In figure (2), we show the normalized logarithm of the relative residual at each GMRES iteration until convergence is achieved with a stopping relative error tolerance of 10^{-6}. Note that the number of discretization points 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.

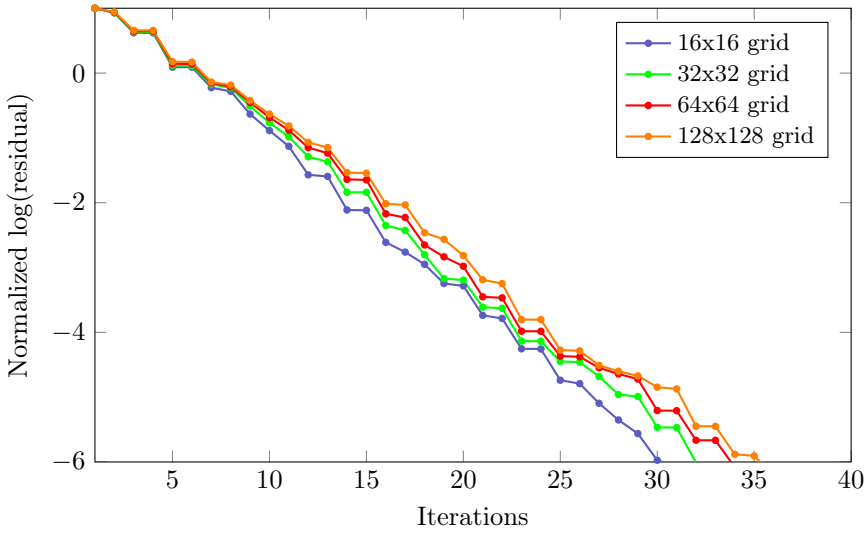


Fig. 2: The number of preconditioned GMRES iterations to reach the solution with an error tolerance of 10^{-6} for lid-driven cavity flow for various mesh sizes.

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 [8] and are a popular area of research today [18, 20, 23, 30]. They have a wide range of applications across biology, 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 $\boldsymbol{u} \in H^1(\Omega)^d$ and pressure $p \in L^2(\Omega)$ such that

$$\begin{aligned} -\mu\Delta\boldsymbol{u} + \nabla p + \mu\kappa^{-1}\boldsymbol{u} &= \boldsymbol{f} \text{ in } \Omega, \\ \nabla \cdot \boldsymbol{u} &= 0 \text{ in } \Omega. \end{aligned} \tag{6}$$

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 field, a measure of the ability of fluid to flow through the media. 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 resolving the obstacles by a mesh becomes difficult. E.g., in [27], the authors model a pebble bed reactor with 440,000 solid spheres. On the other hand, the Darcy equations,

$$\begin{aligned} \nabla p + \mu\kappa^{-1}\boldsymbol{u} &= \boldsymbol{f} \text{ in } \Omega, \\ \nabla \cdot \boldsymbol{u} &= 0 \text{ in } \Omega, \end{aligned} \tag{7}$$

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, 14], and adding an interface condition between the two regions, e.g., the Beavers-Joseph-Saffman interface condition [4, 11]. However, the Stokes-Darcy equations might not be accurate for many reasons (see [25] 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.

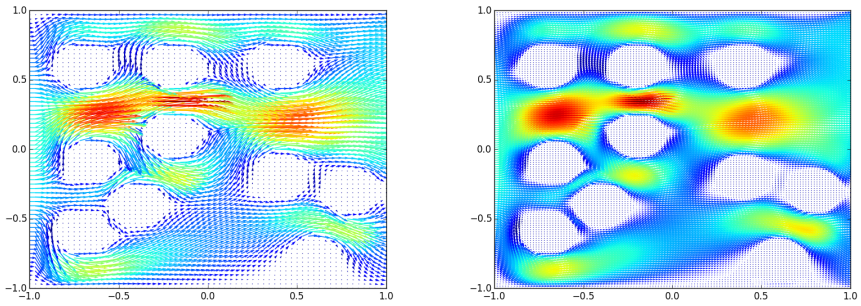
The Brinkman model does not have any of these problems. There is no need for the generation of complicated meshes, as the no-slip boundary conditions are replaced by drag terms in the solid obstacles or porous areas. Furthermore, these features make the Brinkman model easier to implement in a finite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

3.2.1 Numerical Results

As a test problem, we will show Brinkman flow in a channel domain with ten obstacles. Here, the permeability κ is large at nodes that overlap with the obstacles and zero elsewhere. The domain and viscosity are the same as in the Stokes test problem, with $\Omega = (-1, 1)^2$ and $\mu \equiv 1$. The prescribed flow has inflow and outflow boundary conditions (as opposed to the enclosed flow of the lid-driven cavity problem), given by the equations

$$u_x = 1 - y^2, \quad u_y = 0. \tag{8}$$

The streamlines become more accurate (that is, 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. Again, we use GMRES to solve the problem, with a stopping tolerance of 10^{-6}.



(a) 32x32 macroelement grid. (b) 64x64 macroelement grid.

Fig. 3: Brinkman flow through ten obstacles of radius 0.075. $\kappa = 10^6$ at all nodal points that overlap the obstacles.

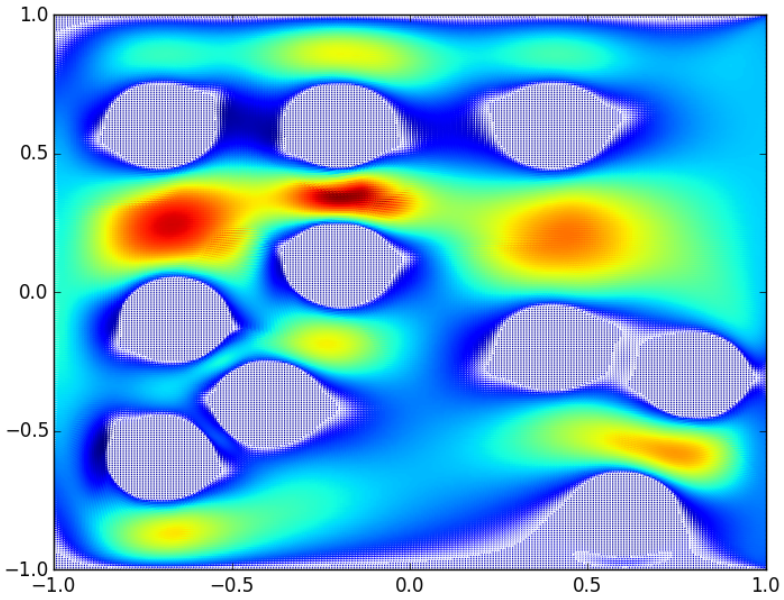


Fig. 4: Magnitude of Brinkman flow on a 128x128 macroelement grid.

In figure (5), we show the normalized logarithm of the relative residual at each GMRES iteration until convergence is achieved with a stopping relative error tolerance of 10^{-6}. As before, the number of discretization points does not affect the required number of iterations since we are using a GMG preconditioner. However, our preconditioner is not optimal for Brinkman flow; see section (4.2) for details.

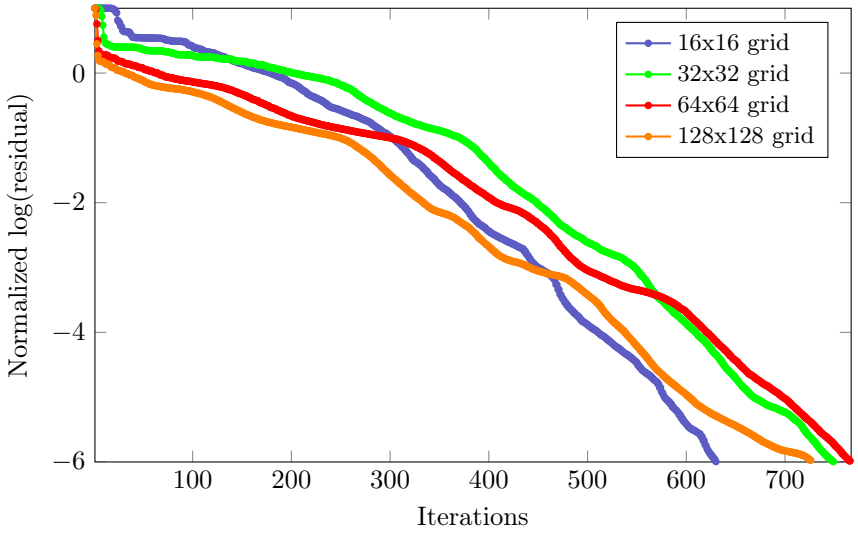


Fig. 5: The number of preconditioned GMRES iterations to reach the solution with an error tolerance of 10^{-6} for the above Brinkman flow for various mesh sizes.

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 (6). 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).

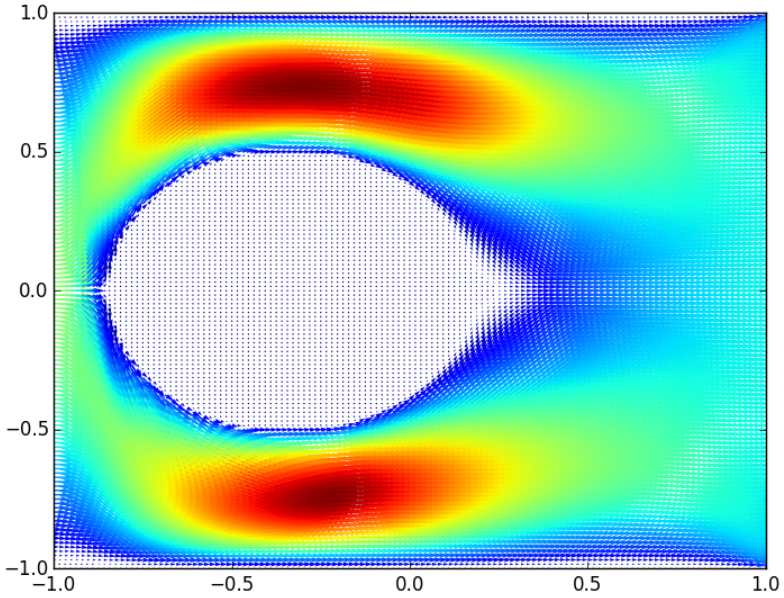


Fig. 6: Brinkman flow with one obstacle on a 64x64 macroelement grid.

4 Methods

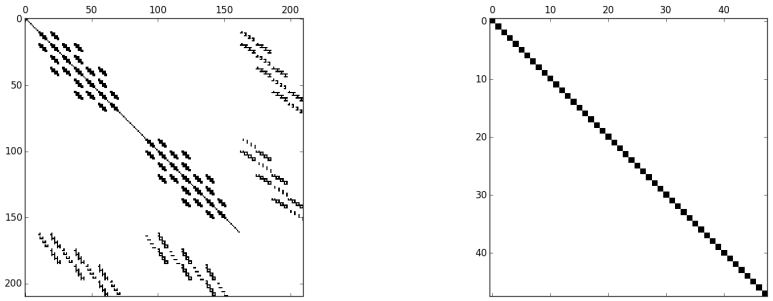
In modern FEM solvers, there are many methods to increase the speed and reduce the computing resources needed to find the solution. We implement several of these methods and compare the efficiency improvements that are achieved.

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 assemble it into a global matrix. The below code shows the general idea for Q_2 elements in two dimensions. The structures of the assembled global matrices are shown in figures (7) and (8).

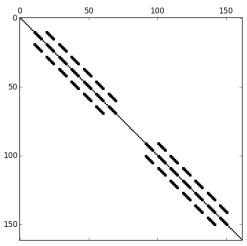
```
# loop over Gauss points
for igpt = 1:nngpt
 # evaluate derivatives, etc.
 (jac, invjac, phi, dphidx, dphidy) = deriv(sigpt, tigpt
    , xlv, ylv)
 (psi, dpsidx, dpsidy) = qderiv(sigpt, tigpt, xlv, ylv)
 (chi, dchidx, dchidy) = lderiv(sigpt, tigpt, xlv, ylv)
 for j = 1:9
  for i = 1:9
   ae[:, i,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

Assembling a global stiffness matrix is optimal for low-order elements. For high-order discretizations, the 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 $K : \mathbb{R}^d \to \mathbb{R}^d$ is our stiffness matrix, we want

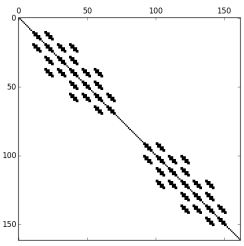


(a) Global stiffness matrix. (b) Global mass matrix

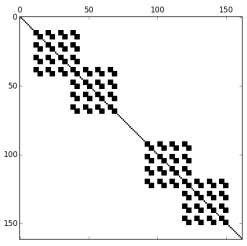
Fig. 7: Structures of finite element matrices for Q_2-P_{-1} elements on an 8x8 grid.



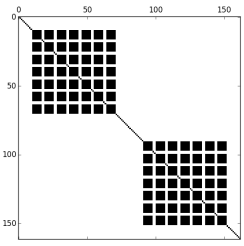
(a) Q_1 elements on an 8x8 grid. The average number of nonzeros per row is 4.85 (2.99%).



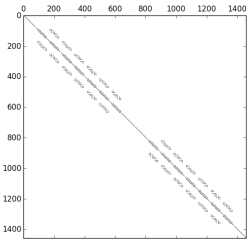
(b) Q_2 elements on a 4x4 grid. The average number of nonzeros per row is 6.96 (4.28%).



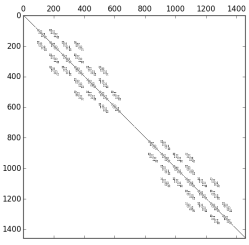
(c) Q_4 elements on a 2x2 grid. The average number of nonzeros per row is 12.26 (7.57%).



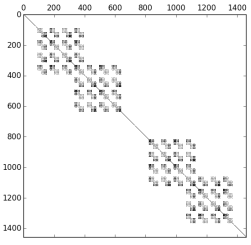
(d) Q_8 elements on a 1x1 grid. The average number of nonzeros per row is 30.04 (18.54%).



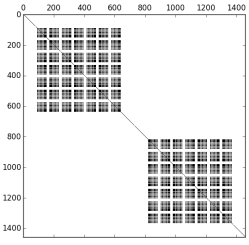
(e) Q_1 elements on an 8x8x8 grid. The average number of nonzeros per row is 9.94 (0.68%).



(f) Q_2 elements on a 4x4x4 grid. The average number of nonzeros per row is 17.22 (1.18%).



(g) Q_4 elements on a 2x2x2 grid. The average number of nonzeros per row is 41.40 (2.84%).



(h) Q_8 elements on a 1x1 grid. The average number of nonzeros per row is (11.11%).

Fig. 8: Structures of the FEM matrices for the Laplacian operator for various elements and grids. For each dimension, parameters are changed so that the number of degrees of freedom stays the same (162 for 2D and 1458 for 3D). We include the mean number of nonzero entries for each row to demonstrate the decreasing level of sparsity that emerges for higher orders and larger grid sizes, especially in three dimensions (the number in parenthesis is the mean row density).

to construct a function $k : \mathbb{R}^d \to \mathbb{R}^d$ such that $k(v) = Kv$ for all $v \in \mathbb{R}^d$. Properties of tensor-product forms allow us to create matrix-free operators that are more efficient than assembling the global matrix. Consider the action of the stiffness matrix K on an input vector u of length n^d. Let D_i be the derivatives of the Lagrange interpolants at the Gauss quadrature points. For each element $e = 1, \ldots, N_e$ in $\mathbb{R}^d$,

$$v^e = \sum_{j=1}^{d} \sum_{i=1}^{d} D_i^T W_{ij}^e D_j u^e. \tag{9}$$

where W_{ij} are diagonal matrices (see appendix (A)). Since the W_{ij} are diagonal, the bulk of the work in (9) comes from the application of the derivative. In 2D, the action of the first derivative can be written in tensor form as $D_1 u^e = (I \otimes \hat{D})u^e$, where $\hat{D}$ is the 1D derivative matrix. On a rectangular domain, this leads to factoring the stiffness matrix into

$$K = (I \otimes \hat{D}^T)W(I \otimes \hat{D}) + (\hat{D}^T \otimes I)W(\hat{D} \otimes I), \tag{10}$$

where W is diagonal. The total cost of this operation is only $O(n^3)$ since the number of flops needed for each application of the derivative is $2n^2(2n-1)$ and the cost of the application of W is only $O(n^2)$. If the matrix is explicitly formed, the cost is $O(n^4)$ since the number of flops needed for a matrix-vector operation with a matrix of size $n^2 \times n^2$ is $n^2(2n^2 - 1)$. 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 [12] for further details. Thus, tensor-product operators are more efficient the higher the order and the higher the dimension compared to the assembled matrix.

We use two different matrix-free methods for each operator (Laplacian, divergence, and mass matrices). Boundary conditions are dealt with easily in matrix-free methods. Instead of modifying the local stiffness matrices such that rows and columns corresponding to Dirichlet nodes (we want to zero out the row and column and place a one on the diagonal), simply place zeros in the entries of the input vector that correspond to the boundary node indices before looping over the elements. After applying the action of the stiffness matrices on the modified u, replace the entries corresponding to the boundary nodes with the original data. This way, boundary indices are only accessed once instead of once for every element. For illustrative purposes, we will show the methods for the stiffness matrix only. 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.

```
# zero dirichlet bdy conditions
uu = copy(u)
u[bdy] = zeros(length(bdy))
u[bdy+dof] = zeros(length(bdy))

# 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
w[bdy] = uu[bdy]
w[bdy+dof] = uu[bdy+dof]
```

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 element stiffness matrices are applied once *outside* of the element loop. The advantage here is that the matrix-matrix product can be evaluated using a single, highly optimized routine. Here, the code for boundary conditions is omitted as they are dealt with in the same manner as above.

```
# 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 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 (9) and (10). 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.

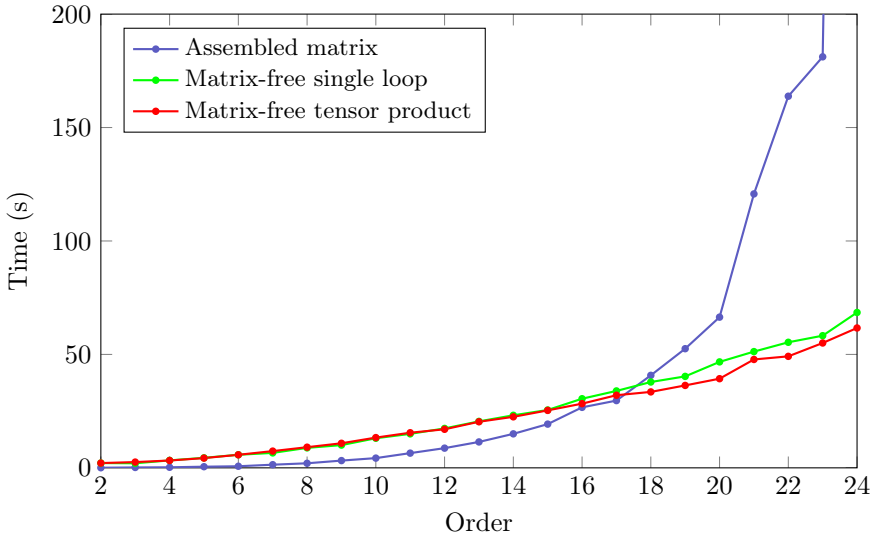


Fig. 9: Time to compute 100 2D vector-Laplacian operations for different orders on a 32x32 macroelement mesh.

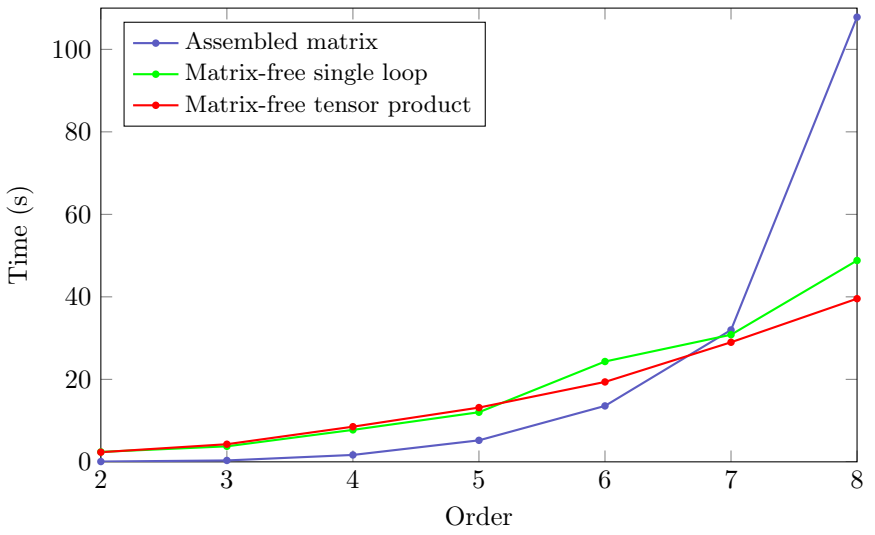


Fig. 10: Time to compute 100 3D vector-Laplacian operations for different orders on an 8x8x8 macroelement mesh.

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. For higher orders especially, we expect that our code performs as well as or better than similar code in Matlab. We include one speed comparison with similar Matlab code in figure (11).

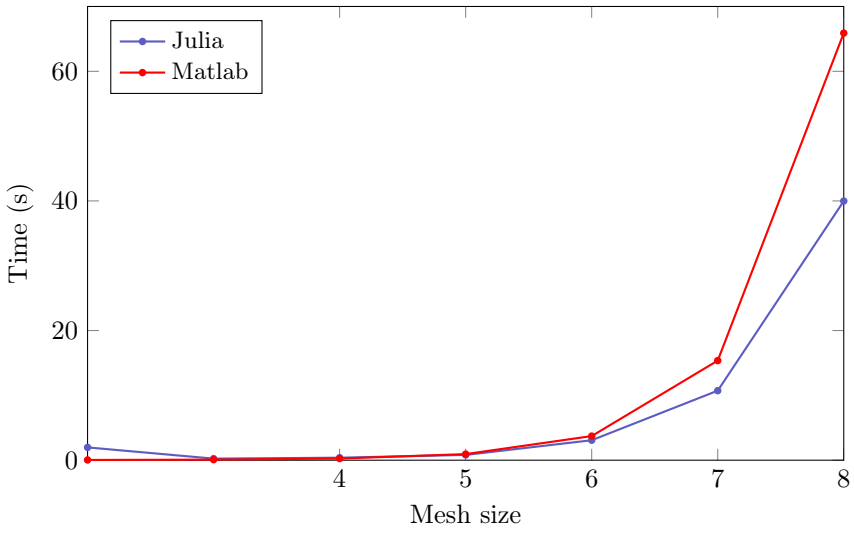


Fig. 11: Time to compute 100 2D vector-Laplacian operations for Q2 elements on different mesh sizes. The numbers n on the x-axis correspond to a mesh of size 2^n x 2^n.

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 optimal complexity and resulting efficiency. 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. [16]. 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. For both Stokes and Brinkman flow, our preconditioner is of the form

$$M = \begin{bmatrix} P & 0 \\ 0 & \text{diag}(Q) \end{bmatrix},$$

where P^{-1} is calculated by applying a GMG V-cycle to the Laplacian and Q is the pressure mass matrix. For Stokes flow, this preconditioner performs well. Unfortunately, the same is not true for Brinkman flow. The condition number of M is affected by the mesh size and becomes worse if there are large jumps in

the permeability. A more complicated operator is needed for effective computation that usually involves approximating the action of the inverse of the Schur complement, $S = -BA^{-1}B^T$. See, for example, [24] for other preconditioning methods for Brinkman flow. This is one area of our code that could be improved in the future.

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.

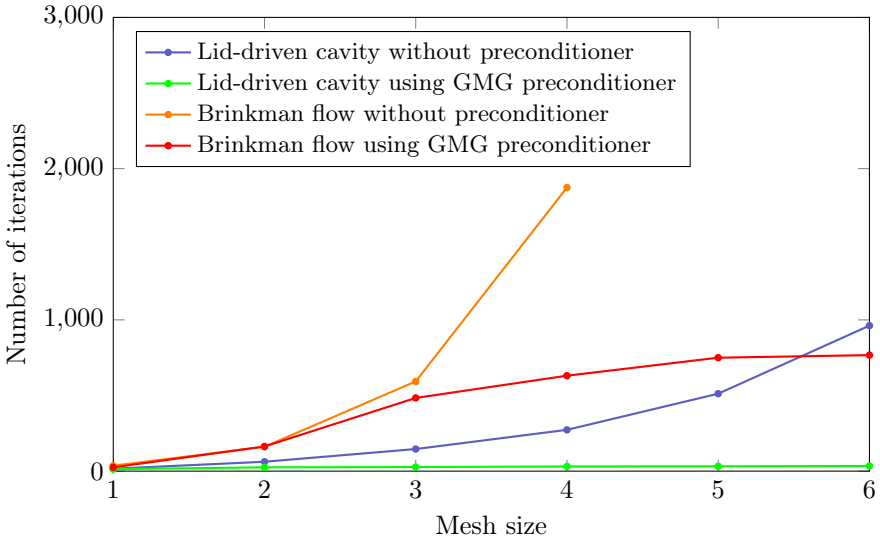


Fig. 12: We solve two of our test problems, lid-driven cavity flow and Brinkman flow through ten obstacles, with and without using a GMG preconditioner. The y-axis shows how many iterations of GMRES were needed to reach an error tolerance of 10^{-6}. Note that the number of iterations needed quickly diverges for meshes larger than 8x8 (our solver does not converge 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 B.

4.3.1 Numerical Results

We compare timings for 100 vector-Laplacian operations for Q_2 elements on large three-dimensional grids for a various number of processors, using the tensor-product matrix-free method. Unsurprisingly, parallelization makes a larger difference the greater the degrees of freedom of the problem. We were not able to achieve perfect parallelization because of communication overhead when waiting to sync results among all processors. The results suggest that a smaller number of workers is actually more efficient for smaller problems. For very large problems, speedup is achieved with each additional worker. Tests were carried out on a compute server with 6 Intel Xeon 2.3 GHz processors and 768 GB memory.

Mesh size	1 processor	2 processors	4 processors	8 processors
16x16x16	5.95	6.42	5.51	9.32
32x32x32	45.16	30.09	19.96	24.54
64x64x64	346.43	220.17	149.46	107.59

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 performance similar to Matlab.

Our package is a 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
Matrix-free tensor-product forms of element stiffness matrices

$$K_e = \begin{bmatrix} Q_x & Q_y & Q_z \end{bmatrix} \begin{bmatrix} r_x & r_y & r_z \\ s_x & s_y & s_z \\ t_x & t_y & t_z \end{bmatrix} JW \begin{bmatrix} r_x & s_x & t_x \\ r_y & s_y & t_y \\ r_z & s_z & t_z \end{bmatrix} \begin{bmatrix} Q_x \\ Q_y \\ Q_z \end{bmatrix}$$

```
function element_stiffness(self, eid, r, J, D)
 gpts = element_gauss(self, eid, r);
 nn = length(J);
 factor = zeros(nn, 6);

 if (self.dim == 2 )
      mu = map(self.coeff, gpts[:,1], gpts[:,2] );

      factor [:,1] = (D.rx.*D.rx + D.ry.*D.ry ) .* J .* r
          .W .* mu ; # d2u/dx^2
      factor [:,2] = (D.sx.*D.sx + D.sy.*D.sy ) .* J .* r
          .W .* mu ; # d2u/dy^2
      factor [:,3] = (D.rx.*D.sx + D.ry.*D.sy ) .* J .* r
          .W .* mu ; # d2u/dxdy

      Ke = r.Qx' * diagm(factor[:,1]) * r.Qx + r.Qy' *
          diagm(factor[:,2]) * r.Qy + r.Qx' * diagm(factor
          [:,3]) * r.Qy + r.Qy' * diagm(factor[:,3]) * r.
          Qx ;
 else
      mu = map(self.coeff, gpts[:,1], gpts[:,2], gpts
          [:,3] );

      # first compute dj.w.J.J'
      factor [:,1] = (D.rx.*D.rx + D.ry.*D.ry + D.rz.*D.
          rz ) .* J .* r.W .* mu ; # d2u/dx^2
      factor [:,2] = (D.sx.*D.sx + D.sy.*D.sy + D.sz.*D.
          sz ) .* J .* r.W .* mu ; # d2u/dy^2
      factor [:,3] = (D.tx.*D.tx + D.ty.*D.ty + D.tz.*D.
          tz ) .* J .* r.W .* mu ; # d2u/dz^2

      factor [:,4] = (D.rx.*D.sx + D.ry.*D.sy + D.rz.*D.
          sz ) .* J .* r.W .* mu ; # d2u/dxdy
      factor [:,5] = (D.rx.*D.tx + D.ry.*D.ty + D.rz.*D.
          tz ) .* J .* r.W .* mu ; # d2u/dxdz
      factor [:,6] = (D.sx.*D.tx + D.sy.*D.ty + D.sz.*D.
          tz ) .* J .* r.W .* mu ; # d2u/dydz

      Ke = r.Qx' * diagm(factor[:,1]) * r.Qx + r.Qy' *
          diagm(factor[:,2]) * r.Qy + r.Qz' * diagm(factor
          [:,3]) * r.Qz + r.Qx' * diagm(factor[:,4]) * r.
          Qy + r.Qy' * diagm(factor[:,4]) * r.Qx + r.Qx' *
```

```
          diagm(factor[:,5]) * r.Qz + r.Qz' * diagm(
         factor[:,5]) * r.Qx + r.Qz' * diagm(factor[:,6])
          * r.Qy + r.Qy' * diagm(factor[:,6]) * r.Qz ;
  end
  return Ke
end
```

B 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] Christine Bernardi and Frédéric Hecht, *More pressure in the finite element discretization of the Stokes problem*, ESAIM: Mathematical Modelling and Numerical Analysis, Volume: 34, Issue: 5, pp. 953-980, 2000.

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

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

[8] 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.

[9] Carsten Burstedde, Omar Ghattas, Georg Stadler, Tiankai Tu, Lucas C. Wilcox, *Parallel scalable adjoint-based adaptive solution for variable-viscosity Stokes flows*, Computer Methods in Applied Mechanics and Engineering Volume 198, pp 1691-1700, 2009.

[10] Mingchao Cai, Andy Nonaka, John B. Bell, Boyce E. Griffits, Aleksandar Donev, *Efficient Variable-Coefficient Finite-Volume Stokes Solvers*, Communications in Computational Physics Volume 16, pp 1263-1297, 2014.

[11] 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.

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

[13] H.C. Elman, D.J. Silverster, A.G. Wathen, *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics*, Oxford University Press, 2005.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

[21] Dave A. May, Jed Brown, and Le Pourhiet, *High-performance methods for long-term lithospheric dynamics*, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, IEEE Computer Society Press, 2014.

[22] Dave A. May and Louis Moresi, *Preconditioned iterative methods for Stokes flow problems arising in computational geodynamics*, Physics of the Earth and Planetary Interiors Volume 171, pp 33-47, 2008.

[23] 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.

[24] Peter Popov, *Preconditioning of Linear Systems Arising in Finite Element Discretizations of the Brinkman Equation*, Large-Scale Scientific Computing, Lecture Notes in Computer Science, pp 381-389, Springer Berlin Heidelberg, 2012.

[25] 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.

[26] Johann Rudi, A. Crisiano I. Malossi, Tobin Isaac, Georg Stadler, Michael Gurnis, Yves Ineichen, Costas Bekas, Alessandro Curioni, Omar Ghattas, *An Extreme-Scale Implicit Solver for Complex PDEs: Highly Heterogeneous Flow in Earth's Mantle*, SC15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (to appear), 2015.

[27] 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.

[28] Y. Saad and M.H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 7, pp. 856-869, 1986.

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

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