Cached Gaussian elimination for simulating Stokes flow on domains with repetitive geometry

Ounan Ding, Craig Schroeder[1]

Department of Computer Science and Engineering
University of California Riverside

Abstract

Microfluidic "lab on a chip" devices are small devices that operate on small length scales on small volumes of fluid; these devices find uses in a variety of applications. Designs for microfluidic chips are generally composed of standardized and often repeated components connected by long, thin, straight fluid channels. We propose a novel meshing algorithm for use in simulating the linear incompressible stationary Stokes equations on geometry with these features, which produces sparse symmetric positive indefinite systems with many repeated matrix blocks. We use a discretization that is formally third order accurate for velocity and second order accurate for pressure in the L^∞ norm. We also propose a novel linear system solver based on cyclic reduction, reordered sparse Gaussian elimination, and operation caching that is designed to efficiently solve systems with repeated matrix blocks. We demonstrate that the resulting fluid solver is significantly faster than existing methods up to resolutions of a few million degrees of freedom for microfluidic problems.

Keywords: Microfluidics, Stokes flow, Cyclic reduction, Sparse direct solver

1. Introduction

Fluid simulation plays an important role in engineering. These applications vary greatly in the type of fluid being considered, the shape and size of the fluid domain, the number of phases, and in many other ways. This has led to the development of a wide variety of methods that try to be as flexible and general as possible, maximizing their applicability to a wide range of problems. This flexibility comes at a cost, as such methods are unable to take advantage of application-specific properties. In this work, we develop a method for simulating the stationary flow of fluids through channel-based microfluidic devices. The fluid domains for these devices generally consist of simple components connected by long, thin pipes. In this paper, we specifically develop a meshing algorithm that, when combined with a standard finite element discretization, converts the stationary Stokes flow problem into a linear algebra problem that is of a form that can be efficiently solved, and we propose an algorithm to solve this linear algebra problem very efficiently. We use a discretization that is formally third order accurate in L^∞ for velocity and second order accurate

[1]craigs@cs.ucr.edu

in L^∞ for pressure. We demonstrate that the proposed algorithm achieves significant speedups on such problems at practical resolutions.

Existing methods for microfluidic simulation. Numerical simulation of microfluidic devices presents few fundamental problems for existing methods, and software packages suitable for microfluidics applications are readily available. Indeed, numerical studies are typically carried out using an off-the-shelf software package such as OpenFOAM [1], COMSOL [2, 3], CFD-ACE+ [4], or Fluent [5] (See [6, 7] for an overview of existing tools). Although some of these software packages often have support specifically for microfluidic applications, they operate using general-purpose numerical methods and do not take advantage of the special properties of these devices. The high computational cost of these methods has led to significant interest in application-specific numerical methods. A particularly popular approach is the one-dimensional analysis model, which approximates the full fluid equations based on an analogy between flow of fluid through tubes and the flow of current through wires [8, 9, 10, 11]. This is a modeling approximation and will introduce systemic errors. See also [12] for a thorough introduction to these techniques. We take a different approach to obtaining faster simulation results; in contrast to the one-dimensional analysis model our method avoids systematic errors. Rather than relying on properties of these devices to approximate the physics, we instead use these properties to accelerate the solution of the full fluid equations.

Properties of microfluidic devices. Microfluidic devices are devices that operate on fluids on small (microliter or nanoliter) scales to perform a variety of tasks, such as common laboratory tests. Microfluidic chips are generally constructed by laying out components that perform specific operations on fluid volumes. Examples of common microfluidic operations are merging (combining different reagents together), mixing (forcing fluids through a serpentine flow to encourage the fluid to mix through molecular diffusion), delaying (holding fluid for a designated period of time to allow chemical reactions to complete), or forking (dividing a fluid flow among multiple directions for separate uses). These components are then connected with thin fluid channels to route fluid from one component to the next. A natural result of the way these devices are designed and constructed is that the geometry contains many duplicated copies of a relatively small number of distinct components. A relatively large fraction of the fluid domain consists of thin, straight (or occasionally circular) fluid channels. The global topology of the device is typically quite simple, usually planar and sometimes even lacking loops. Although the proposed algorithm is a general-purpose algorithm for single-phase Stokes flow, it is specifically designed and optimized around the particular features of the geometry of the fluid domain. It can be readily adapted to a variety of PDEs, including the Navier-Stokes equations, the Poisson equation, and the heat equation.

1.1. Meshing strategies

In this paper, we discretize the Stoke equations using the finite element method with tetrahedral (triangular) elements. The effectiveness of the proposed algorithm relies on our meshes having special properties. The meshing of repeated components needs to be identical, and the mesh within pipes needs to be highly repetitive. We also require

a few simple additional properties of the mesh. These are fairly unusual properties to request from a general-purpose meshing algorithm, and we are aware of no algorithms that satisfy them. For these reasons, we construct our own simple application-specific meshing algorithm. Our input geometry is assumed to be broken into components of known types. This allows us to naturally follow a decomposition/template-matching approach [13, 14].

1.2. Existing sparse linear system solvers

The most significant contribution of the proposed method is the special structure of the linear algebra problem and our algorithm for solving it. The general linear algebra problem we consider here is symmetric, indefinite, and sparse. Methods for solving these problems fall generally into direct and indirect methods.

1.2.1. Direct solvers

Direct methods for solving sparse linear systems of equations have been extensively studied [15]. These methods are mostly variations of Gaussian elimination and the related LU, Cholesky, and LDL^T factorizations. Simply applying direct dense methods to sparse systems tends to quickly result in large amounts of fill-in. Effective direct solvers for sparse systems seek to strike a balance between reducing fill-in, utilizing available computational resources (SIMD, threading), and controlling memory usage. Our algorithm is a block-elimination algorithm that is designed around the specific properties of our fluid domains. It is designed to exploit commodity manycore hardware with significant SIMD processing resources. The bulk of the runtime is spent performing large numbers of simple block matrix operations, which make very efficient use of SIMD resources and can be scheduled in parallel across many threads. Our elimination algorithm is divided into four distinct stages and draws on ideas borrowed from a variety of other direct methods.

Elimination ordering. Fill-in can be reduced by choosing a suitable elimination ordering [16], and many ordering strategies have been evaluated. Of these, two strategies are most relevant to our method. The first of these is the COLAMD algorithm [17], which is an approximation of the minimum degree ordering [18, 19]. Variations on the minimum degree ordering have been popular throughout the history of the development of sparse direct solvers, and we use the COLAMD ordering in the final stage of our elimination algorithm. Nested dissection [20, 21] has also received significant attention. Nested dissection is a recursive divide-and-conquer strategy where the domain is first divided in half by inserting a *separator*; this divides the domain into two independent problems, which may be solved in parallel. In a final step, the separators are eliminated, which requires a global solve. Separators play a similar role in our algorithm, where we use them for isolation, to expose parallelism, and to expose redundancy. Unlike with more general problems, where eliminating the final separator is often the most expensive step in the entire algorithm, our special domain-specific geometry means that separators are generally very small and can be eliminated relatively efficiently.

Multifrontal methods. Permuting the rows and columns of the matrix before performing factorization suffices to reduce fill-in, but the straightforward algorithm is not able to effectively utilize SIMD performance. This led to the

development of frontal methods [22, 23], which perform the elimination steps on a dense *frontal matrix*, which allows dense linear algebra (and efficient BLAS routines) to be used. These methods were replaced with multifrontal methods, which are based on the observation that elimination dependencies take the form of an *elimination tree*, and a new independent elimination front can be started from each leaf of the tree [24]. Since these fronts are independent, they may be eliminated in parallel [25, 26, 27, 28]. A process of amalgamation (also called supernodes) is used to eliminate multiple rows with similar sparsity patterns at the same time to exploit more efficient level-3 BLAS operations [24, 29, 30]. Publicly available libraries implementing the multifrontal method are readily available, including MUMPS [28, 31, 32] and UMFPACK [33, 34, 35]. We compare the performance of the proposed method against both libraries in Section 6.6.

Cyclic reduction. The proposed algorithm utilizes the idea of cyclic reduction, a variation on Gaussian elimination for tridiagonal systems where all odd rows are eliminated in parallel to expose opportunities for parallelism [36, 37]. This reduces the problem size by approximately half, and it produces another tridiagonal system so the process can be repeated. As a serial algorithm, cyclic reduction requires about 2.7 times as many operations as the usual Gaussian elimination [38, 39]. The benefit of this method is that it exposes large numbers of operations that can be performed efficiently in parallel on a variety of architectures [40], especially on GPUs [41]. Cyclic reduction may also be formulated as a divide-and-conquer algorithm, with separate subproblems for even and odd variables [42]. Although our domains may have complex topology and do not lead to tridiagonal systems, many of the decisions that we make during meshing are designed to produce a tridiagonal block structure over significant portions of the matrix. This allows us to take advantage of cyclic reduction during a portion of our elimination phase.

Relation to the proposed method. Although our method is neither a multifrontal method nor cyclic reduction, it has many similarities to these methods. Our block-elimination may be considered as an amalgamation strategy to increase opportunities for level 3 BLAS use. We plan out our computations during a planning stage, and we also make critical use of the ability to begin elimination from many blocks in parallel. Since our blocks are large enough to make effective use of vector resources, we do not assemble fronts in the proposed method. This effectively breaks up large frontal calculations into similarly-sized pieces as was done in [31]. As in cyclic reduction, we have a tridiagonal block structure for significant portions of our matrix, and we eliminate them using a recursive even-odd strategy to maximize parallelism and (when possible) caching opportunities.

1.3. Iterative methods

Some of the most efficient algorithms known for solving large sparse linear systems are iterative. Of these, the Krylov methods are perhaps the most popular, with the conjugate gradient (CG) algorithm being the earliest, best known, and most understood [43]. CG assumes that the matrix Qis symmetric positive definite, but other Krylov schemes such as MINRES [44] or GMRES [45] may be used instead when the system is symmetric but indefinite. The convergence of Krylov methods depends on the conditioning of the system [46, 47], and a preconditioner is often

required for rapid convergence [48]. One important class of efficient preconditioners is based on domain decomposition [49, 50], which splits the domain into subdomains. The smaller (and cheaper) sub-problems provide rapid local convergence, and a coarsened problem is solved to improve global convergence. The most efficient preconditioners, however, are multigrid methods, which also use a coarsened problem to improve low-frequency convergence but use a smoother instead for high-frequency convergence; this coarsening process is repeated in a hierarchy for optimal $O(n)$ convergence.

Multigrid methods have become the standard for efficient large-scale preconditioners, especially for the elliptic problems [51, 52], though they can also be applied to the Stokes equations [53], the Navier-Stokes equations [54, 55, 56], the Euler equations [57], and fluid-structure interaction [58]. Multigrid parallelizes well and is well-suited to GPU implementation [59, 60] and heterogeneous environments [61]. Although multigrid is asymptotically optimal for large problems (even scaling to billions of degrees of freedom [62]), it is generally not the most efficient choice at medium or low resolutions, especially in domains with the small features typical of microfluidic designs. This work thus fills two important roles. (1) *The proposed algorithm allows the incompressible stationary Stokes equations to be solved more efficiently on microfluidic problems at small and medium resolutions.* (2) At high resolution, multigrid methods require a separate solver to solve the system at the coarsest resolution; *the proposed algorithm may be used for this purpose.* We have not pursued this strategy, but it may be a promising avenue for future work.

Contributions and novelty. In this paper, we make the following novel contributions.

- We propose a special solver for sparse symmetric indefinite systems of linear equations that have many repeated matrix blocks. This solver uses a combination of caching, cyclic reduction, and general sparse solver techniques to solve these linear systems very rapidly. The algorithm is designed to maximize the occurrence of repeated matrix blocks and duplicated linear algebra computations during Gaussian elimination. Duplicate block matrices and vectors are detected during a planning stage, avoiding the need to compute or store matrices that are equivalent.

- We propose a meshing algorithm that can be combined with a standard finite element discretization for the incompressible stationary Stokes equations to produce linear systems in a form suitable for our new rapid solver.

- We evaluate our solution method across different resolutions and core counts; we also compare its performance to existing solvers. The full Stokes algorithm is significantly faster than existing solvers at medium resolutions (around 1M degrees of freedom) on the types of geometry that typically occur in designs for microfluidic devices. The algorithm scales well to many cores.

2. Overview of algorithm

2.1. Stokes equations

Microfluidic devices operate at small length scales (feature width $< 0.1\,mm$) on small volumes of fluid ($< 1\,\mu L$) traveling at slow speeds ($< 1\,cm\,s^{-1}$). At these scales, the Reynolds number is low ($\ll 1$), and Stokes flow becomes a good approximation for the fluid flow (though not always [63]). Within the Stokes regime, the dynamics are dominated by incompressibility and a balance of viscous and pressure forces. The momentum and continuity equations reduce to

$$\nabla \cdot \boldsymbol{\sigma} = 0, \qquad \nabla \cdot \mathbf{u} = 0, \qquad \boldsymbol{\sigma} = \mu(\nabla \mathbf{u} + \nabla \mathbf{u}^T) - p\mathbf{I},$$

where $\boldsymbol{\sigma}$ is the fluid stress, $\mathbf{u}$ is the fluid velocity, μ is the dynamic viscosity, and p is the pressure. We consider a mixture of velocity ($\mathbf{u} = \mathbf{a}$) and traction ($\boldsymbol{\sigma}\mathbf{n} = \mathbf{b}$) boundary conditions, where $\mathbf{n}$ is the normal direction. The resulting meshing and discretization leads to a symmetric indefinite sparse linear system of equations for the unknowns $\mathbf{u}$ and p.

Although we will limit our discussion to the incompressible stationary Stokes equations, most of the ideas are not specific to the Stokes equations. The proposed algorithm can be readily adapted to solve the linear systems that arise from the discretization of the Poisson equation, the heat equation, the incompressible unsteady Stokes equations, and the Navier-Stokes equations. Each of these problems results in one or more symmetric linear systems of equations whose sparsity and block structure are the same as the Stokes equations.

2.2. Properties of microfluidic devices

Although fluid domains may be very irregular and complex, this is not the case in some important applications. For example the plumbing in a typical building is composed almost entirely from pipes with standardized diameters, which meet at a relatively small number of standardized junctions (tee, elbow, cross, reducer, plug, valve, etc.). The advantages of designing the plumbing in buildings in this way are obvious; the standardized parts can be cheaply mass produced. As long as these pipes and standardized junctions are meshed and discretized in exactly the same way each time they occur, they will produce identical matrix blocks in the final system.

Although microfluidic devices are fabricated entirely differently (typically by a process like CNC milling), in practice the designs of these devices tend to closely resemble plumbing. These designs are dominated by standardized components (joints, mixers, delays) connected by straight (or less commonly circular) fixed-width channels. A typical chip is designed by first determining which components are required to perform the desired fluidic operations (combine two input fluids, mix them together thoroughly, let them react for a specified amount of time, etc.). Then, the components are connected by channels to route fluids from component to component in the proper sequence. The result is that, as with the plumbing example, one may mesh and discretize the fluid domain so that the final matrix contains many identical matrix blocks.

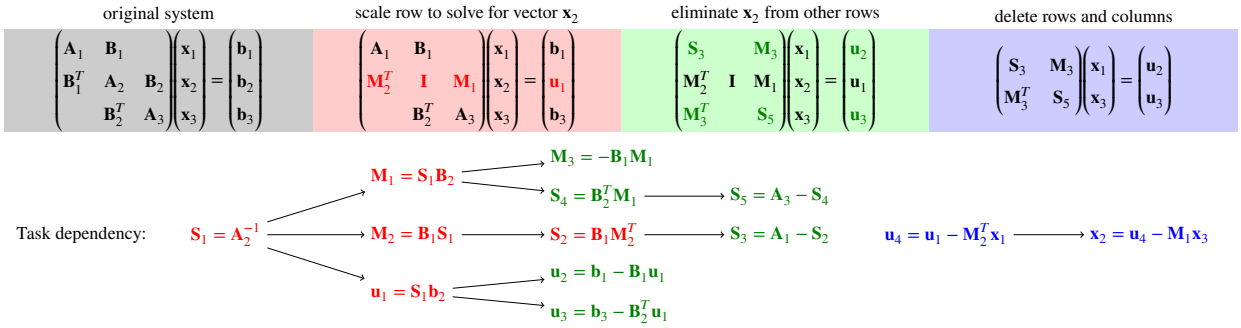


Figure 1: Any row of a system can be eliminated, provided the diagonal block can be inverted. Elimination preserves system symmetry; the matrices $\mathbf{A}_i$ and $\mathbf{S}_i$ are symmetric. Each step of the elimination process requires a primitive linear algebra operation, which may be considered as a task. Even on this very small example, opportunities for computing tasks in parallel emerge rapidly.

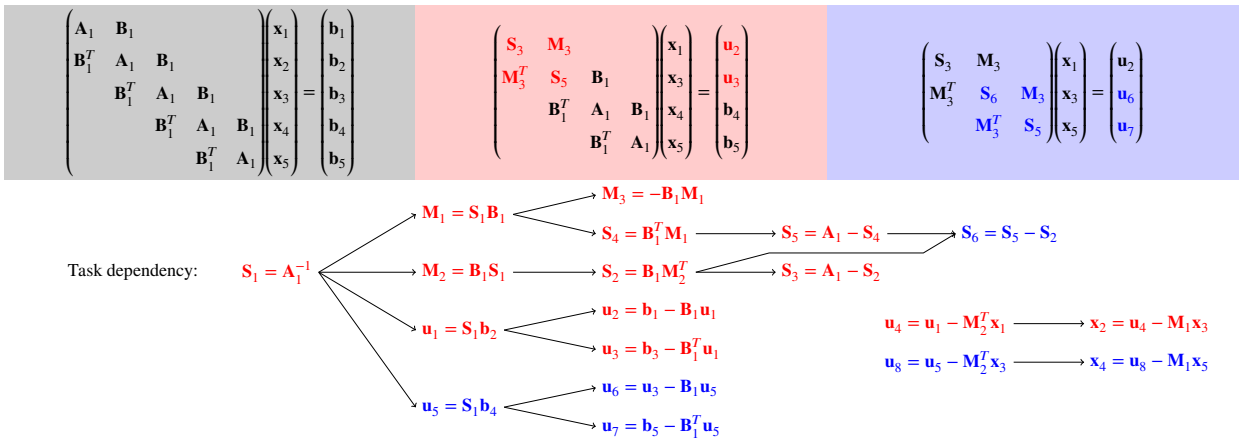


Figure 2: Eliminating similar but independent rows benefits greatly from caching. In this case, the first elimination (red) generates 13 tasks, of which 6 are $O(n^3)$ matrix operations, where the blocks are $n \times n$ matrices. The second elimination (blue) generates only 6 tasks, all of which are much cheaper $O(n^2)$ operations. Additional parallelism is introduced, including in the backsolve phase.

2.3. Elimination

Gaussian elimination classically precedes by eliminating rows from a matrix one by one in a serial algorithm. One may begin by eliminating any row or block of rows (ignoring stability concerns) as shown in Figure 1. This effectively modifies neighboring rows of the matrix based on the sparsity pattern. The eliminated row may be removed from the system, though its entries will be required during the backsolve phase. This is equivalent to forming the Schur complement. If the original matrix is symmetric and the diagonal block is chosen as the pivot, the new matrix will also be symmetric, as can be seen in Figures 1 and 2.

Observe that only the row being eliminated and its neighboring rows (based on the matrix sparsity pattern) are modified; rows that are not neighbors can thus be eliminated independently and in parallel. These are key observation that underlie the success of multifrontal methods [24]. If the two independent rows being eliminated contain identical matrix blocks, many of the calculations required to eliminate one of the rows can be reused when eliminating the other

row (See Figure 2).

These observations suggest that significant performance improvements may be possible if one is able to create duplicated matrix blocks in the system matrix. Under general circumstances of irregular problem domains and irregular meshing, one would not expect duplicated matrix blocks to occur. The ability to benefit from caching relies on repeated geometry and meshing that takes advantage of it. As noted earlier, the geometry of microfluidic devices tends to be redundant; we just need to be careful to mesh and discretize these redundancies consistently.

2.4. Pipes

Long and thin channels (which we will generally refer to as pipes) are common in microfluidic devices; indeed, a significant fraction of the fluid domain may consist of pipes. Pipes are special for our purposes because they are very efficient to eliminate. Consider a long thin pipe, which is broken up into fixed-width slices. Each slice has identical geometry and is meshed and discretized identically. The resulting system matrix will be block tridiagonal. All of the blocks along the diagonal are identical, and all of the off-diagonal blocks are identical (up to transpose). The matrix follows the same pattern as in Figure 2.

Observe that all odd rows may be eliminated independently for nearly the same cost as eliminating just one of the rows. The only calculations that cannot be reused are the much less expensive Qmatrix-vector multiplies and axpy operations that occur as part of the forward and backward triangular solves. Further, most of the matrices that are left behind after eliminating all of the odd rows are again identical (they all follow the $(\mathbf{M}_3^T, \mathbf{S}_6, \mathbf{M}_3)$ pattern observed in the middle row at end end of Figure 2). Thus, the process can be repeated. This recursive even-odd elimination pattern is just cyclic reduction [42]. Ignoring vector operations, each recursive step requires a constant number of matrix operations. Since the number of recursive steps is logarithmic in the number of slices in the pipe, long pipes can be eliminated very efficiently. Moreover, caching is possible between pipes even when they have different lengths, as long as the pipe diameters and slice widths are the same. In practice, there are additional complications relating to scaling and orientation; these will be addressed in Section 3.9.

2.5. Cross sections as blocks

The process of meshing and discretizing our geometry into a linear system begins with a geometric definition of a block. These blocks divide the fluid domain into small regions whose discretizations will eventually become matrix blocks. Blocks should be redundant where possible to facilitate the formation of repeated matrix blocks; the choice of blocks will have significant performance implications.

We have seen that a tridiagonal block matrix structure can be eliminated very efficiently and without fill-in using cyclic reduction. This suggests that the geometry should be sliced into cross sections that have only two neighboring cross sections as we do for pipes. This definition works for geometry that is topologically a pipe. For more irregular geometry like a tee junction, some blocks must have more than two neighbors, and some degree of fill-in is unavoidable. Instead, we seek to limit the propagation of fill-in through the matrix. We do this by inserting *separators* around

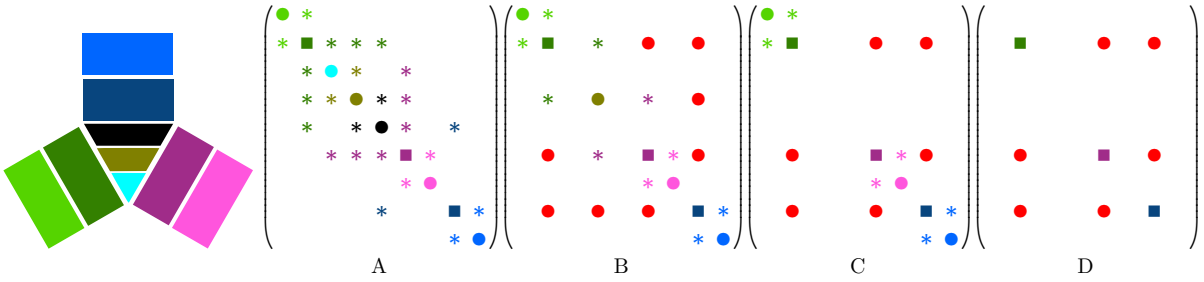


Figure 3: Eliminating components will not fill in past separators. On the left we show an example domain, which consists of four components (three arms and one joint) and is split into blocks marked by different colors. In (A) we show its corresponding system, where block matrices for separators are marked by squares (■ ■ ■), and other block matrices on the diagonal are marked by circles (● ● ● ● ● ●). Non-zero off-diagonal block matrices for connections are marked by *, colored by the connection's parent block. In (B) we eliminate the bottom and top block (● ●) inside the joint. This introduces fill-ins (red circles ●) but they are all confined in the separators. In (C), we eliminate the remaining middle block (●) inside the joint. Finally we eliminate all non-separators (● ● ●) to reach a system in (D). Note that before (D), eliminating a component does not cause fill-in in other components.

irregular components. We eliminate the separators after all other blocks, effectively dividing the system into isolated matrices. Fill-in from any component is localized to the component itself and the separator blocks that bound it (See Figure 3). We can then define a (non-separator) block to be a cross section of geometry that has at most two neighboring non-separator blocks. In this way, we can use cyclic reduction to efficiently eliminate the blocks within components, which comprise the significant majority of blocks. At this point, only a relatively small number of separator blocks remain. They are eliminated last; fill-in during this stage may be significant, but it is limited by both the small number of blocks involved and the planar connectivity typically found in microfluidic devices.

2.6. Reusable component

Separators isolate components from each other, allowing them to be meshed and discretized independently. Duplicated components need only be divided into blocks, meshed, and discretized once. In addition to saving time and space, this also ensures that duplicated components lead to duplicated blocks and duplicated block matrices. Reuse of computations occurs at the level of blocks, not components per se. For example, pipes of different lengths should be divided into blocks that are the same width so that calculations may be reused.

Transforms can change the block matrices of a component, preventing immediate reuse. We can nevertheless reuse components by meshing and discretizing them in a canonical coordinate system and then assembling matrix blocks in the local coordinate system. This can be accomplished through row and column scaling on the final system, as we show in Section 3.9.

2.7. Algorithm steps

We close this overview with the algorithmic tasks that must be completed for the proposed algorithm along with forward references to the discussion of each step.

1. Identify canonical components (Section 3.3)
2. Construct geometry blocks and identify canonical blocks (Section 3.4)
3. Construct canonical block meshes (Section 3.5)
4. Assign degrees of freedom to canonical blocks (Section 3.6)
5. Assemble canonical matrices (Section 3.6)
6. Assign global degrees of freedom (Section 3.7)
7. Assemble the system matrix blocks (Section 3.8)
8. Transform the right hand side (Section 3.9)
9. Plan block elimination (Section 4.1)
10. Execute jobs (Section 4.2)
11. Transform the solution (Section 3.9)

3. Discretization

We are interested in discretizing the Stokes equations on thin and repetitive geometry. We adopt a standard finite element treatment and finite element pair for the Stokes equations, which we summarize here for completeness.

3.1. Finite element formulation

Our finite element discretization follows [64, 65]. We start directly with $\nabla \cdot \boldsymbol{\sigma} + \mathbf{f} = \mathbf{0}$, where $\boldsymbol{\sigma} = \mu(\nabla \mathbf{u} + \nabla \mathbf{u}^T) - p\mathbf{I}$, rather than simplifying to $\mu \nabla^2 \mathbf{u} + \mathbf{f} = \nabla p$ using the incompressibility condition. While this reduces the sparseness of our system, it simplifies the treatment of traction boundary conditions $\boldsymbol{\sigma}\mathbf{n} = \mathbf{b}$. We assume a single fluid phase, so that μ is constant.

Let $\mathbf{w}$ be a test function chosen from the same function space as the velocity $\mathbf{u}$. Then, the weak form of the momentum equation may be written as

$$
\begin{aligned}
0 = \int_\Omega \mathbf{w} \cdot (\nabla \cdot \boldsymbol{\sigma} + \mathbf{f})\, dV = \int_\Omega \nabla \cdot (\mathbf{w} \cdot \boldsymbol{\sigma}) - \nabla \mathbf{w} : \boldsymbol{\sigma} + \mathbf{w} \cdot \mathbf{f}\, dV = \int_{\partial\Omega} \mathbf{w} \cdot \boldsymbol{\sigma}\mathbf{n}\, dA - \int_\Omega \nabla \mathbf{w} : \boldsymbol{\sigma}\, dV + \int_\Omega \mathbf{w} \cdot \mathbf{f}\, dV \\
\iff -\int_{\partial\Omega} \mathbf{w} \cdot \boldsymbol{\sigma}\mathbf{n}\, dA + \int_\Omega \nabla \mathbf{w} : \boldsymbol{\sigma}\, dV = \int_\Omega \mathbf{w} \cdot \mathbf{f}\, dV \\
\iff -\int_{\partial\Omega} \mathbf{w} \cdot \boldsymbol{\sigma}\mathbf{n}\, dA + \int_\Omega \nabla \mathbf{w} : \left(\mu(\nabla \mathbf{u} + \nabla \mathbf{u}^T) - p\mathbf{I}\right) dV = \int_\Omega \mathbf{w} \cdot \mathbf{f}\, dV,
\end{aligned}
$$

where $\mathbf{f}$ is the external force. Letting N_i and P_i be bases for velocity and pressure,

$$
\mathbf{u} = \sum_i N_i \mathbf{u}_i \qquad \mathbf{w} = \sum_i N_i \mathbf{w}_i \qquad p = \sum_i P_i p_i \qquad \phi = \sum_i P_i q_i \qquad \mathbf{f} = \sum_i N_i \mathbf{f}_i \qquad \mathbf{b} = \sum_i N_i \mathbf{b}_i.
$$

Noting the identities

$$
\nabla(f\mathbf{c}) = \mathbf{c}\nabla f^T \qquad (\mathbf{w}\mathbf{z}^T) : (\mathbf{u}\mathbf{v}^T) = (\mathbf{w}^T\mathbf{u})(\mathbf{z}^T\mathbf{v}) \qquad (\mathbf{w}\mathbf{z}^T) : (\mathbf{v}\mathbf{u}^T) = \mathbf{w}^T(\mathbf{v}\mathbf{z}^T)\mathbf{u}
$$

and using the definitions

$$\mathbf{D}_{ij} = \mu \int_\Omega \frac{\partial N_i}{\partial \mathbf{x}} \left(\frac{\partial N_j}{\partial \mathbf{x}} \right)^T dV \qquad \text{tr}(\mathbf{D}_{ij}) = \mu \int_\Omega \left(\frac{\partial N_i}{\partial \mathbf{x}} \right)^T \frac{\partial N_j}{\partial \mathbf{x}} \, dV \qquad \mathbf{A}_{ij} = \text{tr}(\mathbf{D}_{ij})\mathbf{I} + \mathbf{D}_{ij},$$

the integrals can be written as

$$\int_\Omega \mu \nabla \mathbf{w} : (\nabla \mathbf{u} + \nabla \mathbf{u}^T) \, dV = \int_\Omega \mu \nabla \left(\sum_i N_i \mathbf{w}_i \right) : \left(\nabla \left(\sum_j N_j \mathbf{u}_j \right) + \nabla \left(\sum_j N_j \mathbf{u}_j \right)^T \right) dV \tag{1}$$

$$= \int_\Omega \mu \left(\sum_i \mathbf{w}_i \left(\frac{\partial N_i}{\partial \mathbf{x}} \right)^T \right) : \left(\sum_j \mathbf{u}_j \left(\frac{\partial N_j}{\partial \mathbf{x}} \right)^T + \sum_j \left(\frac{\partial N_j}{\partial \mathbf{x}} \right) \mathbf{u}_j^T \right) dV \tag{2}$$

$$= \sum_{ij} \int_\Omega \mu \left(\mathbf{w}_i \left(\frac{\partial N_i}{\partial \mathbf{x}} \right)^T \right) : \left(\mathbf{u}_j \left(\frac{\partial N_j}{\partial \mathbf{x}} \right)^T \right) dV \tag{3}$$

$$+ \sum_{ij} \int_\Omega \mu \left(\mathbf{w}_i \left(\frac{\partial N_i}{\partial \mathbf{x}} \right)^T \right) : \left(\left(\frac{\partial N_j}{\partial \mathbf{x}} \right) \mathbf{u}_j^T \right) dV \tag{4}$$

$$= \sum_{ij} \mathbf{w}_i^T \mathbf{u}_j \left(\mu \int_\Omega \left(\frac{\partial N_i}{\partial \mathbf{x}} \right)^T \frac{\partial N_j}{\partial \mathbf{x}} \, dV \right) + \sum_{ij} \mathbf{w}_i^T \left(\mu \int_\Omega \frac{\partial N_i}{\partial \mathbf{x}} \left(\frac{\partial N_j}{\partial \mathbf{x}} \right)^T dV \right) \mathbf{u}_j \tag{5}$$

$$= \sum_{ij} \mathbf{w}_i^T \left(\text{tr}(\mathbf{D}_{ij})\mathbf{I} + \mathbf{D}_{ij} \right) \mathbf{u}_j \tag{6}$$

$$= \sum_{ij} \mathbf{w}_i^T \mathbf{A}_{ij} \mathbf{u}_j. \tag{7}$$

Using the definitions

$$\mathbf{g}_{ij} = -\int_\Omega \frac{\partial N_i}{\partial \mathbf{x}} P_j \, dV \qquad k_{ij} = \int_\Omega N_i N_j \, dV \qquad m_{ij} = \int_{\partial \Omega_n} N_i N_j \, dA$$

we have

$$\int_\Omega \nabla \mathbf{w} : p\mathbf{I} \, dV = \int_\Omega \nabla \cdot \mathbf{w} p \, dV = \int_\Omega \left(\sum_i \mathbf{w}_i^T \frac{\partial N_i}{\partial \mathbf{x}} \right) \left(\sum_j P_j p_j \right) dV \tag{8}$$

$$= \sum_{ij} \mathbf{w}_i^T p_j \int_\Omega \frac{\partial N_i}{\partial \mathbf{x}} P_j \, dV = -\sum_{ij} \mathbf{w}_i^T \mathbf{g}_{ij} p_j \tag{9}$$

$$\int_\Omega \mathbf{w} \cdot \mathbf{f} \, dV = \int_\Omega \left(\sum_i N_i \mathbf{w}_i \right)^T \left(\sum_j N_j \mathbf{f}_j \right) dV \tag{10}$$

$$= \sum_{ij} \mathbf{w}_i^T \mathbf{f}_j \int_\Omega N_i N_j \, dV = \sum_{ij} \mathbf{w}_i^T \mathbf{f}_j k_{ij} \tag{11}$$

$$\int_{\partial \Omega} \mathbf{w} \cdot \boldsymbol{\sigma} \mathbf{n} \, dA = \int_{\partial \Omega_n} \mathbf{w} \cdot \mathbf{b} \, dA = \int_{\partial \Omega_n} \left(\sum_i N_i \mathbf{w}_i \right)^T \left(\sum_j N_j \mathbf{b}_j \right) dA \tag{12}$$

$$= \sum_{ij} \mathbf{w}_i^T \mathbf{b}_j \int_{\partial \Omega_n} N_i N_j \, dA = \sum_{ij} \mathbf{w}_i^T \mathbf{b}_j m_{ij}. \tag{13}$$

For the incompressibility equation, we begin with $\nabla \cdot \mathbf{u} = -l$, where l is a source term that we include to simplify analytic testing. Physically, $l = 0$ (See Section 6.2). Then, the weak form of the incompressibility equation is just

$$-\int_\Omega \phi \nabla \cdot \mathbf{u}\, dV = \int_\Omega \phi l\, dV. \tag{14}$$

Substituting in our basis using the definition

$$c_i = \int_\Omega P_i l\, dV \tag{15}$$

produces the integrals

$$\int_\Omega \phi \nabla \cdot \mathbf{u}\, dV = \sum_{ij} q_i \left(\int_\Omega P_i \frac{\partial N_j}{\partial \mathbf{x}}\, dV \right) \mathbf{u}_j = -\sum_{ij} q_i \mathbf{g}_{ji}^T \mathbf{u}_j \tag{16}$$

$$\int_\Omega \phi l\, dV = \sum_i q_i \int_\Omega P_i l\, dV = \sum_i q_i c_i. \tag{17}$$

Let $\mathbf{A}$, $\mathbf{G}$, $\mathbf{K}$, and $\mathbf{M}$ denote block matrices whose blocks are given by $\mathbf{A}_{ij}$, $\mathbf{g}_{ij}$, k_{ij}, and m_{ij}. Similarly, let $\mathbf{u}$, $\mathbf{p}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{f}$ be block vectors whose blocks are given by $\mathbf{u}_i$, p_i, $\mathbf{b}_i$, c_i, and $\mathbf{f}_i$. More precisely,

$$(\mathbf{A})_{id+\alpha,jd+\beta} = (\mathbf{A}_{ij})_{\alpha\beta} \qquad (\mathbf{G})_{id+\alpha,r} = (\mathbf{g}_{ir})_\alpha \qquad (\mathbf{u})_{id+\alpha} = (\mathbf{u}_i)_\alpha \tag{18}$$

$$(\mathbf{K})_{id+\alpha,jd+\beta} = k_{ij}\delta_{\alpha\beta} \qquad (\mathbf{p})_r = p_r \qquad (\mathbf{f})_{id+\alpha} = (\mathbf{f}_i)_\alpha \tag{19}$$

$$(\mathbf{M})_{id+\alpha,qd+\beta} = m_{iq}\delta_{\alpha\beta} \qquad (\mathbf{c})_r = c_r \qquad (\mathbf{b})_{qd+\alpha} = (\mathbf{b}_q)_\alpha, \tag{20}$$

where d is the spatial dimension (2 or 3), indices α, β are used for spatial indices, indices i, j are used for velocity sample locations, index r is used for pressure sample locations, and index q is used for boundary velocity sample locations. Then, we can express the full system as

$$\begin{pmatrix} \mathbf{A} & \mathbf{G} \\ \mathbf{G}^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{p} \end{pmatrix} = \begin{pmatrix} \mathbf{Mb} + \mathbf{Kf} + \mathbf{d} \\ \mathbf{c} \end{pmatrix}, \tag{21}$$

where $\mathbf{d}$ is a vector of velocity boundary conditions. This vector is obtained by eliminating velocity dofs associated with velocity boundary conditions from the system and moving them to the right hand side.

3.2. Taylor-Hood element

The choice of basis functions N_i and P_i plays an important role in the numerical stability of a discretization; the Stokes equations require a stable finite element pair to avoid serious numerical problems [66]. We adopt the $(\mathcal{P}_2, \mathcal{P}_1)$ Taylor-Hood element (See Figure 4), which is known to be a stable pair for the Stokes equations [67]. This choice is not essential, and other elements may be preferred [68]. We have found this choice to provide a favorable tradeoff between discretization accuracy, implementation complexity, and numerical stability.

Figure 4: $(\mathcal{P}_2, \mathcal{P}_1)$ Taylor-Hood elements for 2D and 3D. The filled circles (●) are velocity degrees of freedom and hollow circles (○) are pressure degrees of freedom.

3.3. Component construction

For the purposes of this work, we assume that the input geometry is already broken into labeled components. That is, we know what portions of the fluid domain are pipes, joints of various connectivities (bends, tees, crosses), mixers, etc. This design decision greatly simplifies the implementation of the algorithm, and it reflects the way in which these devices are constructed in practice.

The first step in our meshing and discretization process is to transform each component into a canonical coordinate system (with respect to translations, rotations, and optionally scale). We refer to these as *canonical components*. This facilitates the identification of reused components; components are equivalent when their canonical components are the same. Only unique canonical components are represented explicitly. Components are represented as a pointer to a canonical component, a transformation, and connectivity information. The transformations will be used for assembling matrix blocks and transforming the right hand side and solution vectors, as described in Section 3.9. Only canonical components are passed forward to the later stages of the meshing and discretization process (block construction, meshing, and integration). Connectivity information will be used to assemble the final matrix blocks and global system.

The connectivity between components is important for matrix construction, since connections correspond to off-diagonal matrix blocks in the final system. We do this in terms of *connections*. Components have *sockets*, which are places along their boundary where they connect to neighboring components. A pipe has a socket at each end; a tee-junction has three sockets. Connections have a well-defined *cross-sectional shape*, which may differ from connection to connection; these are also canonicalized. Connections consist of (a) the two components that are being connected, (b) which socket of each component is involved, and (c) the canonicalized cross section shape.

We will use the canonical cross section shapes later to ensure consistent mesh generation and discretization between components and blocks. In our 2D implementation, cross sections are line segments. In our 3D implementation, we assume rectangular cross sections between components with fixed depth but potentially different widths. This is consistent with how many microfluidic devices are manufactured, but other cross section shapes may be more appropriate in other contexts (e.g., circular for plumbing). The algorithm is not sensitive to the shapes of cross sections; our use of rectangular cross sections is purely for convenience.

3.4. Block construction

The block construction phase has three primary goals: (a) divide canonical components into *geometry blocks*, (b) identify duplicated geometry blocks, and (c) construct a block-level connectivity graph from the component-level connectivity graph. Geometry blocks are small geometric regions of the fluid domain that will be meshed and discretized and will correspond to block matrices in the final global system. Geometry blocks are the level of granularity at which meshing and finite element integration are performed. Ideally, the domain will be divided into large numbers of small blocks, most of which are identical and have few neighbors.

In our implementation, we used simple rules to divide components into blocks. Pipes are divided into cross sections (geometry blocks) of a fixed characteristic width h (the triangle edge length). Since pipes may have any length, the last geometry block in a pipe may have an irregular width, which we limit to the range $[\frac{h}{2}, \frac{3h}{2}]$. This simple strategy ensures that all but one geometry block within each pipe will be identical, and these blocks will also be identical to the geometry blocks of other pipes with the same cross section. We divide irregular canonical components into strips of width approximately equal to h; strips may run parallel or perpendicular to the pipe direction. Geometry blocks are constructed in a canonical frame to identify duplicates. As with canonical components, we perform all per-block operations on these canonical blocks. Each physical block stores a transform and a pointer to its canonical block.

Once canonical components have been divided into geometry blocks, we must update our connectivity graph. Nodes of the graph are blocks, which store a transform and point to a canonical block. Edges of the graph represent connections between blocks. These connections store the same information as their component-wise counterparts: the blocks being connected, the socket of each block being connected, and the canonical cross section. Note that connections may involve many blocks.

The interiors of geometry blocks are disjoint (they do not overlap). When the boundaries of two geometry blocks intersect, we call the blocks neighbors. Based on this, we divide geometry blocks into three types: *regular blocks*, *irregular blocks*, and *separator blocks*. Separator blocks occur at the boundaries of components; one of the blocks adjacent to each connection is designated as a separator block. In practice, one of these components will be a pipe (or at least pipe-like); we designate the outermost blocks of these pipes as the separator blocks. As many of the remaining blocks are classified as regular as possible, subject to the rule that regular blocks may have at most two neighboring regular blocks. The remaining blocks are classified as irregular blocks. We illustrate different types of geometry blocks in Figure 5. The three types of geometry blocks will be treated differently during the elimination stage of the algorithm.

3.5. Canonical mesh construction

Once we have divided our geometry into geometry blocks, we need to construct meshes on those blocks. We use tetrahedral meshes (triangle meshes in 2D). We divide the algorithm into two stages.

Canonical cross section meshing. The first stage is to mesh the connections between blocks. We independently mesh each canonical cross section. Although the meshing of these cross sections can be performed arbitrarily, special con-

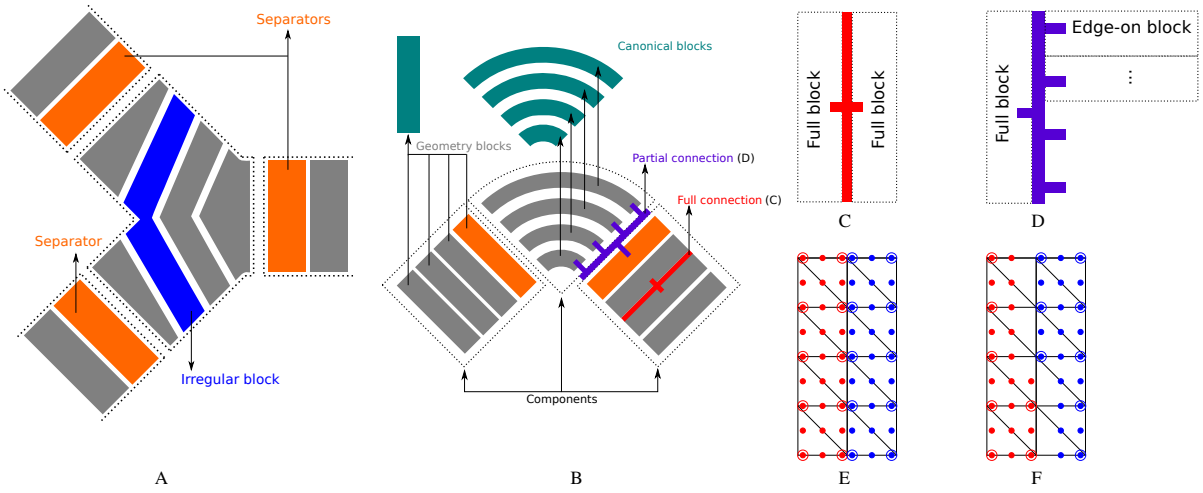


Figure 5: Illustration of terminology. In A and B we show two example domains, where components are enclosed in dotted lines. Components are divided into smaller pieces called geometry blocks; most steps of the algorithm function at this level of granularity. Geometry blocks are classified by their connectivity. Geometry blocks that are on the end of a pipe and touch another component are designated as separator blocks, or separators (■). Non-separator geometry blocks with at most two non-separator neighbor blocks are called regular blocks (■). All remaining geometry blocks have three or more non-separator neighbor blocks and are called irregular blocks (■). Between geometry blocks we might have full (■) or partial (■) connections, shown in B and illustrated separately in C and D. We call the block adjacent to a connection either a full block or an edge-on block based on the connection type, as shown in C and D. In B, we identify blocks with unique shapes as canonical blocks (■). Then we triangulate the canonical blocks and assign the degrees of freedom. When two geometry blocks are next to each other, their canonical degrees of freedom will be duplicated on their boundary, as shown in E. We resolve these to get the global degrees of freedom in F.

siderations are needed to make sure that the final meshes will be consistent. We achieve this by choosing an interface mesh that is reversible. That is to say that the interface mesh looks the same when viewed from either side.

Canonical block meshing. Meshing the interfaces between blocks first based on canonical cross sections gives us a number of important benefits. The interface between blocks is fixed, so we can construct meshes for each block independently. The mesh for the geometry block must conform to the interface mesh, but its generation is otherwise flexible. Since canonical blocks share the same geometry and the same canonical cross sections (and thus the same interface meshes), we can also give them the same mesh. This allows us to construct meshes independently per canonical block. Since the number of canonical blocks is typically much less than the number of geometry blocks, we typically only need to construct and store a mesh for a small fraction of the total fluid domain. We refer to the meshes constructed for canonical blocks as *canonical meshes*.

Meshing restrictions. Although the meshing strategies are generally flexible, we do impose a few extra requirements. We require that each element have at least one edge that is not on the boundary. This would be violated by a tetrahedron at a corner of the domain with three of its faces on the domain boundary; all six of the edges of this tetrahedron are on the boundary. Note that an edge that lies in the interior of a cross section between two blocks is not considered to be a

boundary edge. That is, it is on the boundary of the block but not on the boundary of the full fluid domain mesh. This topology restriction is needed to prevent a numerical nullspace in our final discretization [69]. The second requirement that we impose on block meshes is a connectivity requirement based on the assignment of cross section degrees of freedom to blocks; we address this in Section 3.7.

3.6. Canonical block matrix assembly

Once we have constructed our canonical meshes, we can begin the process of matrix assembly. The first step of our matrix assembly process is to compute the finite element integrals within each canonical mesh. We allocate degrees of freedom according to our Taylor-Hood finite element basis (See Section 3.2). We have a pressure degree of freedom at each vertex and co-located velocity degrees of freedom at each vertex and each edge of the mesh (See Figure 4). The Stokes equations are assembled into a symmetric indefinite linear system following the formulation in Section 3.1. We refer to these matrices as *canonical block matrices*.

Canonical block matrix assembly may be performed independently per canonical mesh (i.e., per canonical block). Since many blocks often share the same canonical block, matrix assembly is typically only performed for a subset of blocks. Matrix assembly occurs in the configuration of the canonical blocks, not in the configuration of the actual blocks, which allows reuse of canonical blocks that differ in orientation. We represent our canonical block matrices as dense matrices; the number of degrees of freedom within each block should be kept small. See Section 5.2 for a discussion of block size and the use of dense matrix blocks. In practice, we delay canonical block matrix assembly until the execution stage. Matrices are assembled when they are first required to improve memory and cache usage.

3.7. Global degrees of freedom assignment

At this stage of the algorithm, we have a notion of canonical degrees of freedom, which are defined from the canonical mesh that we have computed for each canonical block. The canonical block matrices that we have assembled are indexed in terms of the canonical degrees of freedom. Canonical degrees of freedom do not correspond to physical degrees of freedom per se; canonical blocks are assembled in a reference coordinate system, and a single canonical block may correspond to many different geometry blocks within the fluid domain.

Geometry blocks naturally inherit degrees of freedom from their canonical block; we refer to these degrees of freedom as *geometry block degrees of freedom*. Geometry block degrees of freedom do correspond to physical degrees of freedom, but a single physical degree of freedom may belong to more than one block. This occurs for all degrees of freedom which occur along the connections between blocks, as shown in Figure 5. Our task is to assign each physical degree of freedom to one of geometry blocks that contains it. When doing so, we must be careful to avoid numerical problems later in the algorithm (see the note on stability restrictions below and Section 5.1 for details). We will call these *global block degrees of freedom*; they exist in one-to-one correspondence with physical degrees of freedom.

The degree of freedom mapping must be performed on geometry blocks and not canonical blocks. It is sometimes not possible to assign ownership of degrees of freedom to all instances of a canonical block in the same way. Blocks

that are not indexed the same way do not produce duplicated matrix blocks in the final system, so it is desirable for the mapping to be done the same way whenever possible.

Boundary conditions. Boundary conditions affect the assignment of global block degrees of freedom. Velocity degrees of freedom are not allocated where velocity boundary conditions are being enforced; these velocity samples are instead moved to the right hand side. Pressure degrees of freedom are allocated where velocity boundary conditions are being enforced.

Stability restrictions. In order to avoid breakdowns during the elimination process, we divide the degrees of freedom along each connection between two blocks evenly between the two blocks. The reasons for this are discussed in detail in Section 5.1.

Connection types. When assigning parent/child at each connection (see below), it is helpful to distinguish between connections with one block on each side (*full connection*) and connections where a single block on one side of the connection touches multiple blocks on the other (*partial connection*). When a block occupies one entire side of a connection, we call the block a *full block*. Otherwise, the block is considered an *edge-on block*. Full connections have two full blocks. Partial connections have one full block and many edge-on blocks. We illustrate these concepts in Figure 5.

Ownership convention. A simple convention in 2D to resolve ownership of degrees of freedom on connections is to walk around the perimeter of a geometry block in counterclockwise order. When you encounter a connection at which you are a full block, the half of the connection that you encounter first is the half owned by that block. The degrees of freedom on the other half are owned by the block or blocks on the other side of the connection. With this convention, both full blocks at a full connection agree on which half of the degrees of freedom are owned by each block. The only ambiguity is the degree of freedom in the middle (which may be at a vertex or an edge). We must establish a globally-consistent rule for the ownership of this middle degree of freedom. We refer to the block that owns the middle degree of freedom as the *parent* and the block that does not own it as the *child*. Note that a block may (and usually is) a child at one connection and a parent at another. We employ two rules:

1. At partial connections, the full block is always the parent.
2. Blocks that are full blocks with respect to exactly two connections are the parent of one connection and the child of the other.

When the two rules come into conflict, the first rule wins. The purpose of these rules is to avoid creating matrix dependencies between non-neighbor blocks (See Section 3.7.1). The assignment is otherwise arbitrary.

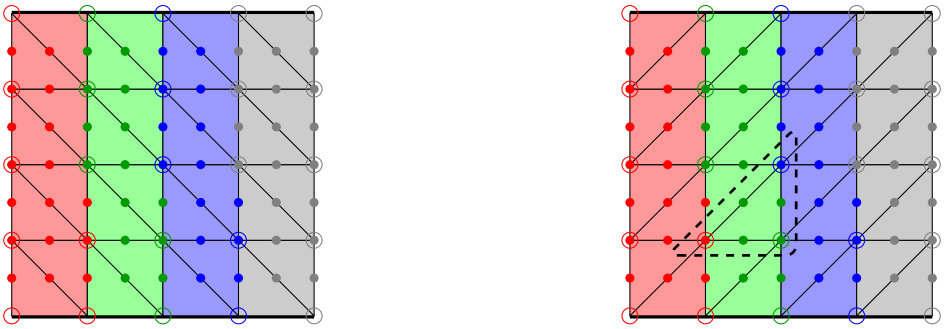


Figure 6: Block meshing with different diagonal edge directions. The background colors indicate the territory of blocks. The colors on edges and vertices indicate which block owns the degrees of freedom. The filled and hollowed circles are velocities and pressures respectively. The triangulation on the left is able to separate non-adjacent blocks. However swapping the direction of the diagonal edges will allow the non-adjacent blocks (blue and red blocks shown on the right) interact. The dashed triangle shows an element containing the blue and red vertices that would introduce a non-zero matrix entry.

Triple junctions. At partial connections (P), there are degrees of freedom shared by three blocks. One of these blocks (A) is the full block with respect to the partial connection. The other two are edge-on blocks (B, C) with respect to this connection; they always connect to each other through a full connection (Q). The partial connection P takes precedence; block A owns the same degrees of freedom that it would if P were a full connection. If the degree of freedom is not owned by block A, we decide whether the degree of freedom belongs to block B or C by looking at connection Q.

3.7.1. Spurious connectivity

When we constructed geometry blocks, we did so in a way that ensured that most blocks only touch two neighbors. Geometry blocks were considered neighbors only if their boundaries intersected. With respect to degrees of freedom, however, the notion of connectivity is somewhat different. Two degrees of freedom are connected if they share an element; pairs of degrees of freedom belonging to the same element correspond to nonzero matrix entries. We want to make sure that the matrix notion of connectivity corresponds to the geometry notion. As shown in Figure 6, it may be possible for degrees of freedom of non-neighboring blocks to be connected if care is not taken. This occurs whenever an element of a geometry block has vertices belonging to two different blocks. (This also occurs at triple junctions, but in this case the blocks involved are already neighbors.) In the case of our simple triangulation strategy, this problem is avoided by (a) choosing the diagonal directions carefully and (b) preventing the block from being the child on both connections. In rare cases, spurious connectivity is still not eliminated; we resolve this by merging blocks.

3.8. System assembly

During canonical block matrix assembly, we perform finite element integration on each canonical block to compute canonical block matrices. This gives us a matrix and right hand side for each canonical block as in (21). We will denote the matrix for block a as $\mathbf{B}_a$ and the right hand side as $\mathbf{b}_a$. These quantities are indexed by canonical block degrees

of freedom. Observe that $\mathbf{B}_a$ is a symmetric matrix. We will ignore transformations in this section; we show how to include them Section 3.9.

The global system that we must solve has matrix blocks that are indexed with global degrees of freedom. Global indices are unique (each degree of freedom belongs to exactly one global matrix block), while a single degree of freedom may exist within multiple canonical blocks. This means that integral contributions may have been calculated for a particular degree of freedom within multiple canonical blocks. These contributions must be added up while calculating global matrix blocks.

We introduce index mapping matrices $\mathbf{P}_{ab}$ to denote the correspondences between degrees of freedom in global blocks and canonical blocks. We define $(\mathbf{P}_{ab})_{ij} = 1$ if the *global* degree of freedom i within block a corresponds to the same degree of freedom as the *canonical* degree of freedom j within block b. $(\mathbf{P}_{ab})_{ij} = 0$ otherwise. Note that $\mathbf{P}_{aa}$ is just the canonical-to-global index map for block a. Since all dofs in a global block exist inside the corresponding canonical block, $\mathbf{P}_{aa}\mathbf{P}_{aa}^T = \mathbf{I}$.

Let $\mathbf{E}_{ab}$ be the global matrix block corresponding to block-row a and block-column b. Let the corresponding right hand side blocks be denoted as $\mathbf{h}_a$. These blocks can be computed from canonical matrix blocks as

$$\mathbf{E}_{ab} = \sum_c \mathbf{P}_{ac}\mathbf{B}_c\mathbf{P}_{bc}^T \qquad\qquad \mathbf{h}_a = \sum_c \mathbf{P}_{ac}\mathbf{b}_c,$$

where c runs over adjacent blocks. If a and c are not neighboring blocks (they do not share any degrees of freedom), then $\mathbf{P}_{ac} = \mathbf{0}$.

3.9. Transforms

To introduce transforms into our matrix blocks, we must first determine how integrals transform over individual elements. We assume that all transforms are affine (per block).

Element-wise transforms. Consider a single element. A world space coordinate $\mathbf{x}$ can be transformed from its canonical space version $\hat{\mathbf{x}}$ by $\mathbf{x} = \mathbf{F}\hat{\mathbf{x}} + \mathbf{c}$, where $\mathbf{F}$ is a constant transform matrix and $\mathbf{c}$ is a constant displacement. Note that we are requiring that the transformation be affine, and we will later restrict it further to a composition of translations, rotations, and uniform scale. Let $J = \det \mathbf{F}$. The basis functions in world space (without a hat) and in canonical space (with a hat) and their derivatives are related by

$$N_i(\mathbf{x}) = \hat{N}_i(\hat{\mathbf{x}}) = \hat{N}_i(\mathbf{F}^{-1}(\mathbf{x} - \mathbf{c})) \qquad P_i(\mathbf{x}) = \hat{P}_i(\hat{\mathbf{x}}) = \hat{P}_i(\mathbf{F}^{-1}(\mathbf{x} - \mathbf{c}))$$
$$\frac{\partial N_i}{\partial \mathbf{x}}(\mathbf{x}) = \mathbf{F}^{-T}\frac{\partial \hat{N}_i}{\partial \hat{\mathbf{x}}}(\mathbf{F}^{-1}(\mathbf{x} - \mathbf{c})) \qquad \frac{\partial P_i}{\partial \mathbf{x}}(\mathbf{x}) = \mathbf{F}^{-T}\frac{\partial \hat{P}_i}{\partial \hat{\mathbf{x}}}(\mathbf{F}^{-1}(\mathbf{x} - \mathbf{c})).$$

Our canonical matrix blocks $\mathbf{B}_a$ consist of viscosity blocks $\mathbf{A}$ and gradient blocks $\mathbf{G}$, as in (21). The blocks $\mathbf{A}$ are comprised of per-element blocks $\mathbf{A}_{ij} = \operatorname{tr}(\mathbf{D}_{ij})\mathbf{I} + \mathbf{D}_{ij}$, which are defined in (1). The blocks $\mathbf{G}$ are comprised of per-element vectors $\mathbf{g}_{ij}$, which are defined in (8). These transform as

$$\mathbf{g}_{ij} = J\mathbf{F}^{-T}\hat{\mathbf{g}}_{ij} \qquad\qquad \mathbf{D}_{ij} = J\mathbf{F}^{-T}\hat{\mathbf{D}}_{ij}\mathbf{F}^{-1}.$$

The matrix $\mathbf{A}_{ij}$, however, does not generally transform in a simple way, unless $\mathbf{F}^{-T}\mathbf{F}^{-1} = \mathbf{I}\,\mathrm{tr}(\mathbf{F}^{-T}\mathbf{F}^{-1})$. This is true if our transform is comprised of a combination of rotation, uniform scale, and translation. This restriction is why we were limited to transformations of this type when computing canonical blocks. With this assumption, we also have

$$\mathbf{A}_{ij} = J\mathbf{F}^{-T}\hat{\mathbf{A}}_{ij}\mathbf{F}^{-1}.$$

Blockwise transforms. Blocks are composed by combining element-wise $\mathbf{A}_{ij}$ and $\mathbf{g}_{ij}$ into block-wise versions $\mathbf{A}$ and $\mathbf{G}$. We can express these in world space and canonical space matrices (for block a) as

$$\mathbf{B}_a = \begin{pmatrix} \mathbf{A} & \mathbf{G} \\ \mathbf{G}^T & \mathbf{0} \end{pmatrix} \qquad \hat{\mathbf{B}}_a = \begin{pmatrix} \hat{\mathbf{A}} & \hat{\mathbf{G}} \\ \hat{\mathbf{G}}^T & \mathbf{0} \end{pmatrix} \qquad \mathbf{B}_a = \mathbf{H}_a^T\hat{\mathbf{B}}_a\mathbf{H}_a,$$

where $\mathbf{H}_a$ is defined as

$$\mathbf{H}_a = \sqrt{J}\begin{pmatrix} \mathbf{F}^{-1} & & & & & \\ & \ddots & & & & \\ & & \mathbf{F}^{-1} & & & \\ & & & 1 & & \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix}.$$

Observe that $\mathbf{H}_a$ applies the matrix transform $\mathbf{F}^{-1}$ to each co-located velocity degree of freedom and only scales pressure degrees of freedom. Since both pressure and viscosity blocks are scaled by J, this scaling is split between the row and column scaling. This allows us to use the same $\mathbf{H}_a$ on both sides. Note that $\mathbf{H}_a$ is indexed using canonical indexing. Note also that the index a refers to the block a of the fluid domain, not a canonical block. Canonical blocks do not have transformations associated with them. Rather, each block stores a pointer to a canonical block and the transformation of the block relative to its canonical block. The matrix $\mathbf{H}_a$ is used to transform from the canonical coordinate system into world space. Although $\mathbf{H}_a$ is defined using canonical indexing, we can define a globally-indexed version by $\overline{\mathbf{H}}_a = \mathbf{P}_{aa}\mathbf{H}_a\mathbf{P}_{aa}^T$. This matrix has the same form as $\mathbf{H}_a$, but it contains a different number of dofs.

Global matrix blocks in canonical coordinates. We can define a canonical-space version $\hat{\mathbf{E}}_{aa}$ of the world space block $\mathbf{E}_{aa}$, which is naturally defined according to $\mathbf{E}_{aa} = \overline{\mathbf{H}}_a^T\hat{\mathbf{E}}_{aa}\overline{\mathbf{H}}_a$ or $\hat{\mathbf{E}}_{aa} = \overline{\mathbf{H}}_a^{-T}\mathbf{E}_{aa}\overline{\mathbf{H}}_a^{-1}$. We can extend this to off-diagonal blocks as

$$\hat{\mathbf{E}}_{ab} = \overline{\mathbf{H}}_a^{-T}\mathbf{E}_{ab}\overline{\mathbf{H}}_b^{-1}.$$

Then,

$$
\begin{aligned}
\mathbf{E}_{ab} &= \sum_c \mathbf{P}_{ac}\mathbf{B}_c\mathbf{P}_{bc}^T \\
\hat{\mathbf{E}}_{ab} &= \sum_c \overline{\mathbf{H}}_a^{-T}\mathbf{P}_{ac}\mathbf{B}_c\mathbf{P}_{bc}^T\overline{\mathbf{H}}_b^{-1} \\
&= \sum_c \overline{\mathbf{H}}_a^{-T}\mathbf{P}_{ac}\mathbf{H}_c^T\hat{\mathbf{B}}_c\mathbf{H}_c\mathbf{P}_{bc}^T\overline{\mathbf{H}}_b^{-1} \\
&= \sum_c \overline{\mathbf{P}}_{ac}\hat{\mathbf{B}}_c\overline{\mathbf{P}}_{bc}^T \\
\overline{\mathbf{P}}_{ac} &= \overline{\mathbf{H}}_a^{-T}\mathbf{P}_{ac}\mathbf{H}_c^T.
\end{aligned}
$$

This directly relates global matrix blocks in canonical coordinates with the canonical block matrices.

Transformation invariance. The matrix $\overline{\mathbf{P}}_{ac}$ maps degrees of freedom (as $\mathbf{P}_{ac}$ does) but also applies a transformation along the way. This transformation is $\sqrt{J_a^{-1}J_c}\mathbf{F}_a^T\mathbf{F}_c^{-T}$ for co-located velocity degrees of freedom and $\sqrt{J_a^{-1}J_c}$ for pressure degrees of freedom. If two blocks are connected, then the relative orientation between the two blocks is fixed. If one block is rotated, scaled, or translated, then the connected block must be rotated, scaled, and translated by the same amount in order to remain connected. This would replace $\mathbf{F}_a \to \mathbf{R}\mathbf{F}_a$ and $\mathbf{F}_c \to \mathbf{R}\mathbf{F}_c$, so that $\mathbf{F}_a^T\mathbf{F}_c^{-T} \to \mathbf{F}_a^T\mathbf{R}^T\mathbf{R}^{-T}\mathbf{F}_c^{-T} = \mathbf{F}_a^T\mathbf{F}_c^{-T}$ is unchanged. Similarly, $J_a^{-1}J_c = \det(\mathbf{F}_a^T\mathbf{F}_c^{-T})^{-1}$ must remain unchanged.

Duplicated matrix blocks. Noting that $\overline{\mathbf{P}}_{ac}$ does not depend on block orientation suggests that $\hat{\mathbf{E}}_{ab}$ may be computed once for each canonical block, but this is not the case. We will have $\hat{\mathbf{E}}_{ab} = \hat{\mathbf{E}}_{cd}$ if (a) all of the blocks involved in the sum correspond to the same canonical blocks, (b) are connected through the same sockets, and (c) are indexed the same way in global indexing. For example, $\hat{\mathbf{E}}_{aa} \neq \hat{\mathbf{E}}_{cc}$ if blocks a and c are connected to different types of blocks, even though a and c have the same canonical block. Requirement (c) is actually somewhat stronger than is required, since not all indices of the blocks involved may participate in the computation of $\hat{\mathbf{E}}_{ab}$. Nevertheless, we use rule (c) since it is easy to check during global degree of freedom assignment. Identifying copies of $\hat{\mathbf{E}}_{ab}$ that are the same is critical, as this the only form of redundancy that will be passed to the final linear system that must be solved.

Transformed system. Consider a simple geometry consisting of three blocks (1, 2, and 3) connected in sequence. The world-space global system that must be solved looks like

$$
\begin{pmatrix}
\mathbf{E}_{11} & \mathbf{E}_{12} & \\
\mathbf{E}_{12}^T & \mathbf{E}_{22} & \mathbf{E}_{23} \\
 & \mathbf{E}_{23}^T & \mathbf{E}_{33}
\end{pmatrix}
\begin{pmatrix}
\mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3
\end{pmatrix}
=
\begin{pmatrix}
\mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3
\end{pmatrix},
$$

Here, $\mathbf{x}_a$ are the degrees of freedom assigned to block a (including velocity and pressure). In general, the blocks $\mathbf{E}_{ab}$ will vary with the orientations of the blocks. We can replace these with canonical-space versions

$$\begin{pmatrix} \overline{\mathbf{H}}_1^T \hat{\mathbf{E}}_{11} \overline{\mathbf{H}}_1 & \overline{\mathbf{H}}_1^T \hat{\mathbf{E}}_{12} \overline{\mathbf{H}}_2 & \\ \overline{\mathbf{H}}_2^T \hat{\mathbf{E}}_{12}^T \overline{\mathbf{H}}_1 & \overline{\mathbf{H}}_2^T \hat{\mathbf{E}}_{22} \overline{\mathbf{H}}_2 & \overline{\mathbf{H}}_2^T \hat{\mathbf{E}}_{23} \overline{\mathbf{H}}_3 \\ & \overline{\mathbf{H}}_3^T \hat{\mathbf{E}}_{23}^T \overline{\mathbf{H}}_2 & \overline{\mathbf{H}}_3^T \hat{\mathbf{E}}_{33} \overline{\mathbf{H}}_3 \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{pmatrix}$$

$$\begin{pmatrix} \hat{\mathbf{E}}_{11} & \hat{\mathbf{E}}_{12} & \\ \hat{\mathbf{E}}_{12}^T & \hat{\mathbf{E}}_{22} & \hat{\mathbf{E}}_{23} \\ & \hat{\mathbf{E}}_{23}^T & \hat{\mathbf{E}}_{33} \end{pmatrix} \begin{pmatrix} \overline{\mathbf{H}}_1 \mathbf{x}_1 \\ \overline{\mathbf{H}}_2 \mathbf{x}_2 \\ \overline{\mathbf{H}}_3 \mathbf{x}_3 \end{pmatrix} = \begin{pmatrix} \overline{\mathbf{H}}_1^{-T} \mathbf{b}_1 \\ \overline{\mathbf{H}}_2^{-T} \mathbf{b}_2 \\ \overline{\mathbf{H}}_3^{-T} \mathbf{b}_3 \end{pmatrix},$$

This amounts to solving for transformed degrees of freedom $\mathbf{y}_a = \overline{\mathbf{H}}_a \mathbf{x}_a$ with a canonical-space matrix and a transformed right hand side. The solution is easily transformed back into world space with $\mathbf{x}_a = \overline{\mathbf{H}}_a^{-1} \mathbf{y}_a$. Solving this canonical-space version of the problem is preferable since it will normally contain many more duplicate matrices than the global space version of the problem.

As with canonical block matrix assembly, we delay the assembly of the global system matrix blocks until the task execution phase. These matrix blocks are assembled from canonical block matrices as they are needed.

4. Elimination algorithm

With our global system fully assembled, our next task is to solve the resulting linear system. The system is block sparse. Many of the blocks in the system are duplicates of other matrix blocks, possibly with transpose. The elimination algorithm proceeds in four phases.

Regular blocks. Regular blocks, by definition, have at most two neighboring regular blocks. If all other blocks are removed, the global system decomposes into separate components (each geometric component corresponds to a connected component in the resulting matrix). Each of these connected components is tridiagonal and can be eliminated efficiently with cyclic reduction. We use this cyclic reduction order to eliminate all regular blocks from the global system. Note that regular blocks may have more than two neighbors in the global system due to the presence of irregular blocks and separators. Irregular blocks and separators result in fill-in during elimination, but they also bound this fill-in by preventing it from spreading outside of a component. The use of cyclic reduction exposes a large number of tasks that can be executed in parallel. In the presence of duplicate matrix blocks (especially pipes), it is also very effective at exposing caching opportunities. The vast majority of blocks are regular, so relatively few blocks remain after this stage of elimination.

Irregular blocks. After regular blocks are eliminated, only irregular blocks and separators remain. Irregular blocks are eliminated next in arbitrary order. Eliminating irregular blocks creates fill-in, but separators bound the fill-in by preventing it from spreading to other components. As long as care is taken to ensure that the order of elimination is

the same every time the blocks in a component are eliminated (start cyclic reduction from the same side, and eliminate irregular blocks in the same order), *nearly all* of the computations involved in eliminating one copy of a component will coincide exactly with the computations needed to eliminate another copy.

Separators I. Separators are eliminated in two phases. During the first phase, separators with at most two neighbors are eliminated in arbitrary order. These blocks are easy to detect, and their elimination never produces fill-in.

Separators II. The remaining separators are eliminated from the system. We use COLAMD order [17] to reduce fill-in. This is the only elimination stage where the amount of fill-in produced is not readily bounded. This is compensated by the fact that only a small fraction of the original number of blocks remain in the system. The planar topology if typical microfluidic devices also tends to limit fill-in. After this stage, all rows of the system have been eliminated.

4.1. Planning and optimization

Each phase of elimination proceeds by repeated application of block-row elimination. Each row operation consists of a sequence of basic linear algebra operations (See Figures 1 and 2). The elimination stages are treated as planning stages; rather than performing the operations required, we instead treat each operation as a tasks. The dependency relationships between the tasks form a directed acyclic graph.

Forward and backward substitution. The row elimination operation also emits tasks for both forward and backward substitution. Note that the operations that will be required for backward substitution are known during elimination stage, even though these tasks would not be able to execute until after the forward phase.

Matrix and vector IDs. To facilitate handling duplicates, we store a sparse matrix of matrix IDs. Each *essentially unique* block matrix is assigned a unique ID. Two matrices are not considered essentially unique if they differ only by negation or transpose. We reserve a bit for negation and a bit for transpose to represent matrices that are essentially the same as another matrix. We reserve two special IDs to indicate a zero matrix and an identity matrix; since the block row and block column in which the matrix is stored uniquely identifies its dimensions, it is unnecessary to distinguish special matrices of different sizes. A similar ID scheme is applied to vectors.

Caching. Each task consists of a simple linear algebra operation and produces an intermediate matrix or vector as output. Each of these intermediate quantities is assigned an ID. A simple hashing scheme is used to detect that an intermediate quantity is being computed twice. The hashing is aware of negation, transpose, associativity, and (for addition) commutativity. We do not include distributivity, as this would make the problem very difficult. This simple hashing scheme allows us to detect and eliminate duplicate calculations and simply reuse the results of the earlier computation. This simple idea is the basis for the majority of the performance benefits observed from the proposed algorithm.

Operation simplification. One benefit of representing identity and zero objects with special indices is that we are able to simplify or eliminate many operations during the planning stage. For example, during an elimination step a matrix-vector multiply by a zero vector simply results in the ID for the zero vector; no task is produced. Similarly, when a zero matrix is added to another matrix, the ID for the second matrix is returned without generating a task. These simplifications can be quite dramatic. For many problems, nearly all initial right hand side vectors are zero, and the vast majority of vector operations that occur during forward elimination will be optimized away. When it does not prevent caching opportunities, we merge tasks to correspond to BLAS operations. This allows us, for example, to merge some sequences of operations ($A = B + C$, $B = -D$, $C = EF$, $E = G^T$) into a single BLAS operation ($A = -D + G^T F$). This also allows us to use the same memory location for A and D. We use the Intel MKL-BLAS for our basic linear algebra and LAPACK for our matrix inverses (and final-block pseudo-inverse when required).

Stability and pseudo-inverse for the last block. Our stability considerations ensure that our elimination procedure does not break down during elimination. That is, we will never be required to invert a matrix block that is singular. The one exception to this is the very last block. If the whole system contains a constant pressure nullspace due to the absence of traction boundary conditions, the last block to be inverted will be singular. We perform a pseudo-inverse on this singular block using the singular value decomposition, which we compute using the appropriate LAPACK gesvd routine. This is done at most once and does not affect performance.

4.2. Task execution

During the task execution phase, we perform all of the actual computations required for elimination. In addition, we also assemble the block matrices for the global system matrix when they are required by elimination calculations (See Sections 3.6 and 3.8). These tasks are both computationally intensive and memory intensive, and they benefit from the parallelism, load balancing, and memory management that we perform during task execution. In addition to a core computation, tasks are also responsible for allocating and assembling finite element matrices (if required and not already available), allocating space for their output (unless it shares space with an input), and freeing memory for intermediates that are no longer required.

We assign a priority to each task equal to the length of the longest dependency chain starting at the task [70, 71, 72]. The time required to complete the most expensive dependency chain places a lower bound on the time required to complete a set of tasks, even if unlimited processors are available to complete them. These priorities tend to encourage long dependency chains to be executed quickly. Indeed, favorable scaling to 16 cores is observed with the proposed method (See Section 6.4).

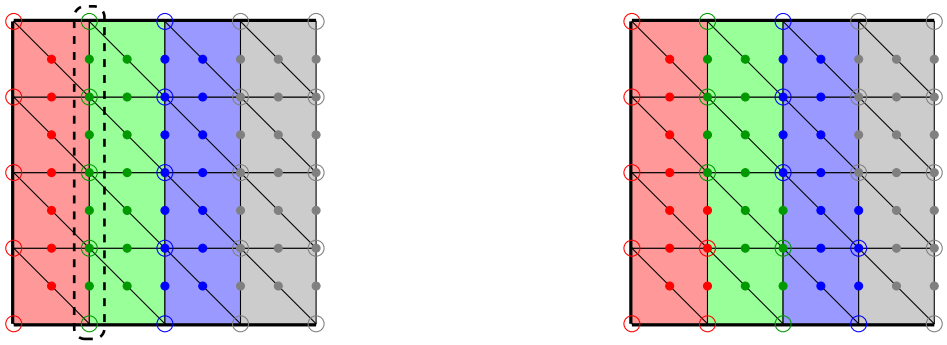


Figure 7: Here we show how different placements of degrees of freedom affect the elimination. The background colors indicate the territory of blocks. The colors on edges and vertices indicate which block owns the degrees of freedom. The filled and hollowed circles are velocities and pressures respectively. The left, bottom, and top sides are with velocity boundary conditions. On the left eliminating the red block would fail. This is because the degrees of freedom on the edge (marked by dashed rectangle) are all owned by the green block, and the elimination of the red block can be regarded as solving a smaller system with solely velocity boundary conditions (three from the original domain and right side being the effective one). The constant pressure nullspace makes the system singular. We solve this by dividing the degrees of freedom between the adjacent blocks (shown on the right).

5. Analysis

5.1. Stability

Being equivalent to un-pivoted Gaussian elimination on a permuted system [73], breakdowns (zeros on the diagonal) and entry growth are in general possible [39]. Cyclic reduction has been extensively studied for the solution of the Poisson equation, which is symmetric and positive definite. It has been shown that cyclic reduction is stable for diagonally dominant and symmetric positive definite systems [42], where off-diagonal entries even tend to become smaller in subsequent iterations [36, 74]. Since our systems are symmetric indefinite, we must take care to avoid these problems.

We place a restriction on block meshing (See Section 3.4) to avoid numerical nullspaces and elimination breakdowns. Assignment of global degrees of freedom also plays an important role in preventing breakdowns. Consider the elimination of an individual row. As the first step we invert the row's diagonal block matrix. This block is just a Stokes flow discretization of the corresponding geometry block with some effective boundary conditions. When the effective boundary conditions correspond to velocity boundary conditions, this discretization has the constant-pressure nullspace. This situation occurs when we assign all shared degrees of freedom to one of the neighboring blocks. We illustrate one example in Figure 7. Our solution to this problem is to split the shared degrees of freedom between the neighbor blocks.

5.2. Scaling

One of the limitations of the proposed method is its scaling with resolution. In the absence of caching opportunities, we will have n geometry blocks, each with m degrees of freedom. Let s be the number of separators (similar to the

number of components).

Memory usage. Each block is dense and requires $O(m^2)$ storage, for $O(m^2 n)$ overall storage. The first three phases of elimination create at most a constant amount of fill-in, so total memory use through the end of these phases is $O(m^2 n)$. This leaves us with $s \ll n$ separators. During this phase, fill-in is possible. Based on the planar topology, the number of operations should grow as $O(s^{1.5})$, which leads to $O(m^2 s^{1.5})$ additional storage. Under refinement by a factor of k, $n \to kn$, $m \to km$ ($m \to k^2 m$ in 3D), and $s \to s$. This leads to $O(k^3)$ (2D) or $O(k^5)$ (3D) scaling in memory usage. In practice, actual memory requirements are significantly better than these predictions, since many blocks are duplicates and need not be stored. It is worth noting that the reduced memory usage may also indirectly improve performance by reducing memory bandwidth requirements, since many block matrices will be reused. Nevertheless, this is noticeably worse than the optimal scaling of $O(k^2)$ (2D) and $O(k^3)$ (3D). Indeed, memory is the limiting factor of our method in 3D, where we start reaching our memory limitations at around 10M degrees of freedom on realistic geometry.

Computational cost. Computational cost closely follows memory usage, except that operations on our blocks scale as $O(m^3)$. This gives us $O(m^3 n + m^3 s^{1.5})$ computational cost. The factor of $s^{1.5}$ accounts for fill-in during the final elimination step and follows the asymptotics of reordered sparse elimination for matrices with planar topology. This scales with resolution k as $O(k^4)$ in 2D and $O(k^7)$ in 3D, which compares poorly with optimal at $O(k^2)$ and $O(k^3)$. In practice, this scaling is not observed as long as channel is not too wide. Over the relevant range of resolutions, our tests suggest 2D scaling of $O(k^a)$ with a between 2.1 and 2.6 on suitable geometry and 3D scaling with a between 5.2 and 5.4. (See Section 6.5 for details.) The proposed method is performance competitive with state-of-the-art methods even at around 1M degrees of freedom in both 2D and 3D. (See Section 6.6 for timing comparisons.)

Impacts of cross section size. The proposed method scales poorly with block size. This suggests that blocks should have as few degrees of freedom as possible while satisfying connectivity requirements. The detrimental effects of large cross sections may be observed in the "wide" test case (in 2D and 3D), where a very wide cross section in a portion of the fluid domain causes global performance deterioration. (See Section 6.6.) The isolating effects of separator blocks and the independence of components means that components with large cross sections may be eliminated using an alternative sparse direct solver (such as MUMPS); the sparse LU or LDL^T factorization may then be used in lieu of dense blocks for the component. This would allow the method to overcome the effects of such components while retaining the benefits in other regions. Observe that the caching benefits are retained; if the same large component is repeated in the device's design, it need only be eliminated once.

6. Numerical results

6.1. Sample device geometries

We use six different geometry templates for our numerical tests.

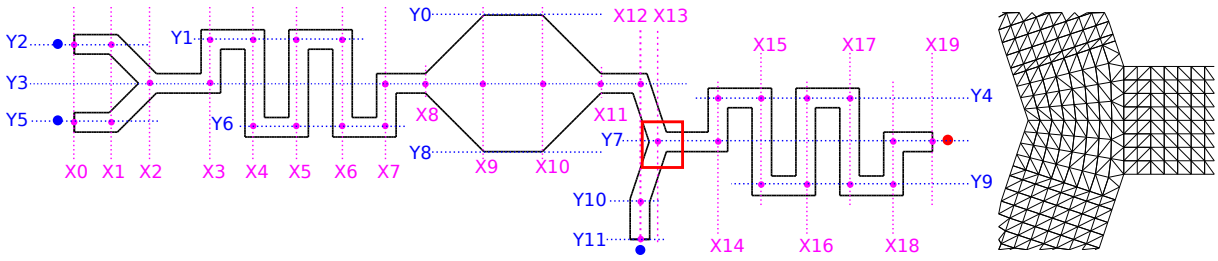


Figure 8: Domain of the test case "wide". The blue dots (•) indicate inflow ports, and the red dots (•) indicate free surface outflow. On the right we show the mesh inside the red box at resolution 8. The x and y coordinates of nodes are labeled with "X#" and "Y#" respectively. Their values can be read from Table 1.

- **"wide"** is an example of a relatively simple microfluidic device. The device and its precise geometry are shown in Figure 8. This geometry includes a component with very large cross sections to illustrate the performance degradation that occurs in this case. Precise coordinates are provided in Table 1.

- **"grid20"** is a large regular grid of pipes (See Figure 9). This example benefits heavily from the regularity of the geometry, resulting in lots of caching opportunities; this tends to accelerate the earlier elimination stages. On the other hand, it has a large number of separator blocks, which makes the final stage of elimination more expensive. Because of the large number of pipes, this example also has the highest degree of freedom count relative to the resolution of the pipes.

- **"rgrid0"** and **"rgrid1"** were selected from a large database of grid-like automatically-generated microfluidic devices [75]. In that work, it was very expensive to solve the Stokes equations for the devices in this database. The geometry for these devices is shown in Figure 9.

- **"voronoi-s4"** and **"voronoi-s15"** are randomly generated from Voronoi diagrams clipped to the circle centered at the origin with radius 0.5 (See Figure 10). In these tests all pipes are joined at different angles; this prevents

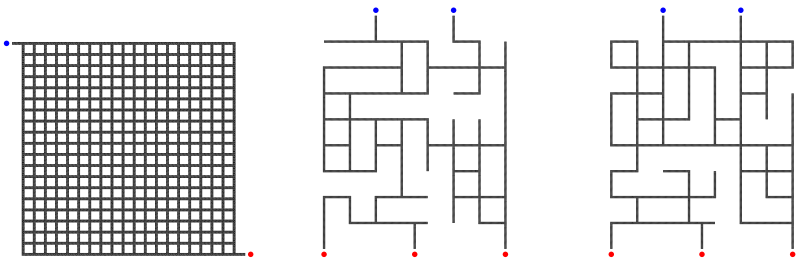


Figure 9: Domains of grid-shaped tests. The blue dots (•) indicate inflow ports, and the red dots (•) indicate free surface outflow. For simplicity, we draw each pipe as a filled stroke, by connecting the central vertices at the ends of the pipe. In "grid20" The coordinates for the bottom left and top right vertices are $(0, 0)$ and $(0.95, 0.95)$ respectively. In tests "rgrid0" and "rgrid1" The coordinates of vertices are contained in a box with bottom left corner $(0, 0)$ and top right corner $(1.575, 2.025)$. In all of these tests a uniform cross section of 0.0125 is used.

X0 = 0.000	X1 = 0.050	X2 = 0.100	X3 = 0.175
X4 = 0.231	X5 = 0.288	X6 = 0.344	X7 = 0.400
X8 = 0.450	X9 = 0.525	X10 = 0.600	X11 = 0.675
X12 = 0.725	X13 = 0.750	X14 = 0.825	X15 = 0.881
X16 = 0.938	X17 = 0.994	X18 = 1.050	X19 = 1.100
Y0 = 0.088	Y1 = 0.056	Y2 = 0.050	Y3 = 0.000
Y4 = -0.019	Y5 = -0.050	Y6 = -0.056	Y7 = -0.075
Y8 = -0.088	Y9 = -0.131	Y10 = -0.150	Y11 = -0.200

Table 1: Coordinates for the test case "wide".

any blocks (other than the pipes) from being cached. Precise coordinates are provided in Table 2.

In each example, a fixed channel width w is used for all pipes. In 3D tests, we extrude the domain along the z direction by w, which produces a square cross section for pipes. We use a characteristic block width of $h = \frac{w}{r}$, where r is the resolution. That is, each pipe is r elements wide. At all inflow regions, we enforce velocity boundary conditions with a quadratic Poiseuille flow velocity profile in 2D. In 3D, the input velocity profile is quadratic in both the horizontal and vertical directions; the velocity is zero at the walls and greatest in the middle. We use flow rates of $0.005m^2s^{-1}$ (2D) or $0.005m^3s^{-1}$ (3D) at all inflows on all non-analytic tests.

6.2. Analytic convergence tests

We begin by performing convergence tests on all of our devices. Since analytic solutions to the Stokes equations are known only for simple geometry setups, we instead use the method of manufactured solutions to perform our analytic tests [76]. In the method of manufactured solutions, one chooses arbitrary analytic velocity and pressure fields and then applies boundary conditions and body forces that make these fields the analytic solution. We choose velocity and

A0 = (-0.231,0.050)	A1 = (0.427,0.156)	A2 = (0.142,0.476)	A3 = (0.225,0.152)
A4 = (-0.048,0.173)	A5 = (0.033,-0.342)	A6 = (0.202,-0.363)	A7 = (-0.109,0.321)
A8 = (-0.332,0.218)	A9 = (-0.077,-0.045)	A10 = (-0.377,-0.104)	A11 = (0.440,-0.131)
A12 = (0.111,-0.004)	A13 = (0.096,0.303)	A14 = (-0.155,-0.268)	A15 = (-0.471,0.062)
A16 = (0.252,-0.127)	A17 = (0.007,-0.493)		
B0 = (0.349,0.314)	B1 = (0.417,-0.019)	B2 = (0.058,0.210)	B3 = (0.146,0.022)
B4 = (0.168,0.418)	B5 = (-0.115,0.447)	B6 = (-0.243,0.302)	B7 = (-0.151,0.100)
B8 = (-0.089,-0.124)	B9 = (0.278,-0.248)	B10 = (-0.403,-0.035)	B11 = (0.094,-0.435)
B12 = (-0.135,-0.348)	B13 = (-0.314,-0.260)	B14 = (0.085,-0.230)	

Table 2: Coordinates for the test cases "voronoi-s4" and "voronoi-s15".

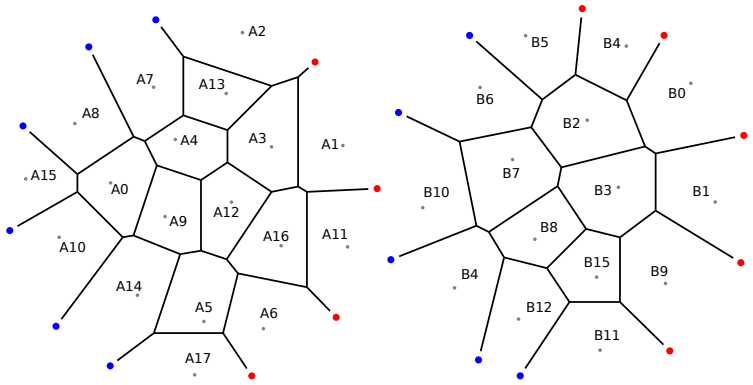


Figure 10: Domains of tests "voronoi-s4" (left) and "voronoi-s15" (right). The blue dots (•) indicate inflow ports, and the red dots (•) indicate free surface outflow. The Voronoi cell centers are labeled by "A#" and "B#" in "voronoi-s4" and "voronoi-s15" respectively; they are listed in Table 2.

pressure fields that are combinations of trigonometry and polynomials and have oscillations on the length scale of about 0.4, which is small enough to be well-resolved at all resolutions and yet high enough to exhibit significant nonlinearity. In all tests, we used the analytic fields below.

Field	2D	3D
$\mathbf{u}(\mathbf{x})$	$\begin{pmatrix} \sin(12x)y + \cos(15y) + xy \\ \cos(14x)\cos(13y) + \sin(16y)x + x^2 - 1 \end{pmatrix}$	$\begin{pmatrix} \sin(14x)y + \cos(15y)z + xy \\ \cos(14x)\cos(16y) + \sin(15y)x + x^2 + yz - 1 \\ \sin(17z)y + \cos(15x)z \end{pmatrix}$
$p(\mathbf{x})$	$\sin(15x + 10y + 1)$	$\sin(15x + 14y + 1) + \cos(16z)$

Note that the velocity fields are not divergence free, do not follow the domain geometry, and do not satisfy the Stokes equations. Instead, we use a right hand side term for divergence (See Section 3.1), enforce velocity boundary conditions at all inflows and pipe walls, enforce traction boundary conditions at outflows, and use a forcing term to make the analytic solution satisfy the Stokes equations. This allows us to do very precise refinement studies even with our very irregular geometry. We use a viscosity of $\mu = 1$.

We compute L^∞ and L^2 errors of computed pressures p and velocities $\mathbf{u}$ using

$$L_p^\infty = \max_i \left| p(\mathbf{x}_i) - p(\mathbf{x}_i) \right| \qquad L_p^2 = \sqrt{\frac{1}{N_p} \sum_i \left(p(\mathbf{x}_i) - p(\mathbf{x}_i) \right)^2}$$

$$L_\mathbf{u}^\infty = \max_j \left\| \mathbf{u}(\mathbf{x}_j) - \mathbf{u}(\mathbf{x}_j) \right\|_\infty \qquad L_\mathbf{u}^2 = \sqrt{\frac{1}{N_\mathbf{u}} \sum_j \left\| \mathbf{u}(\mathbf{x}_j) - \mathbf{u}(\mathbf{x}_j) \right\|_2^2},$$

where N_p is the total number of pressure degrees of freedom, and $N_\mathbf{u}$ is the total number of vertices and edges with velocity degrees of freedom. We conduct the refinement study by changing the resolution r, which is the number of elements along the cross section of a pipe.

The results of the refinement tests are shown in Figure 11 for 2D and Figure 12 for 3D. In all cases, we observe second order convergence in pressure and third order convergence in velocity in both L^∞ and L^2. This is the optimal

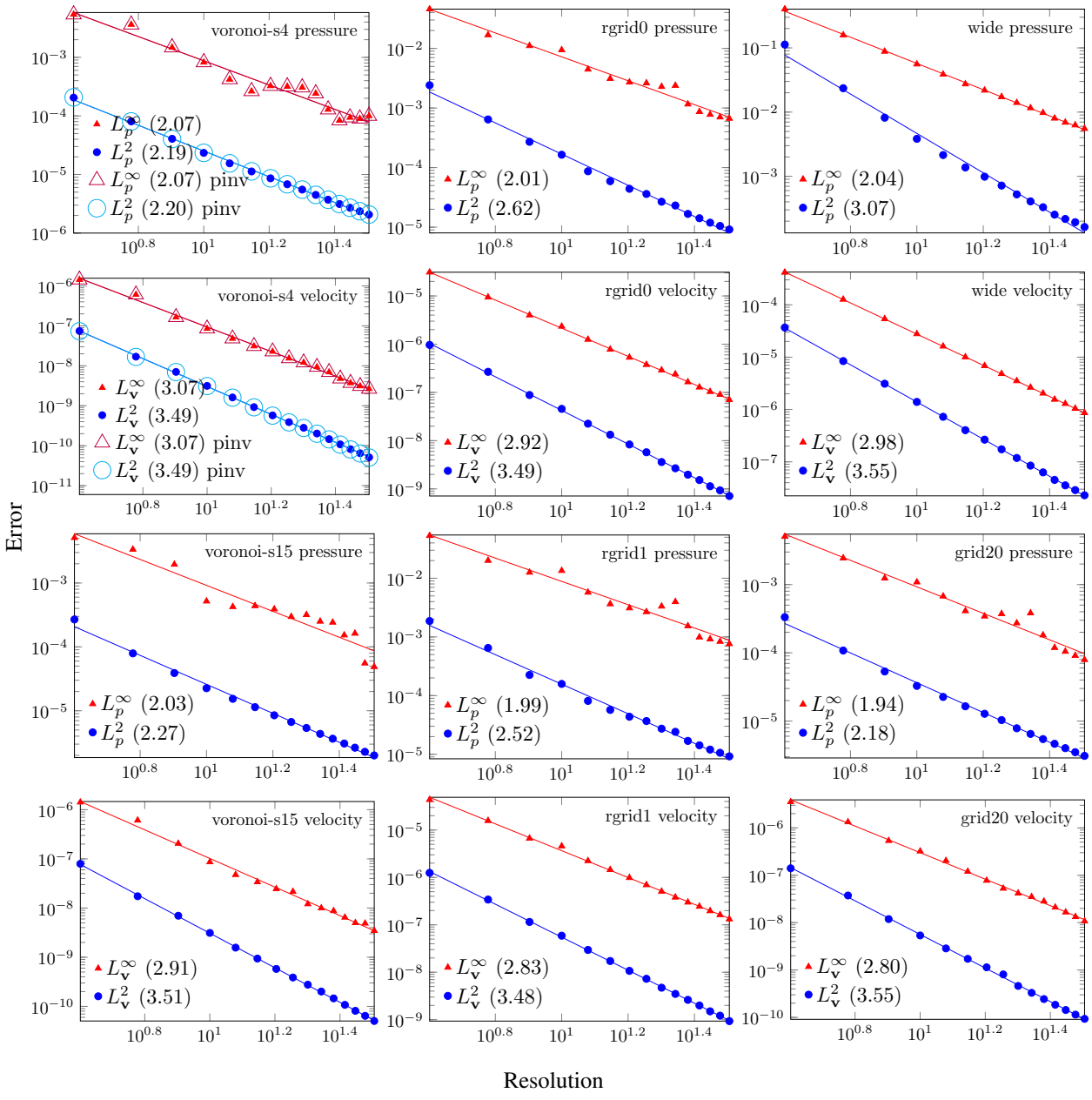


Figure 11: **Analytic** convergence tests for L^∞ and L^2 error measures in **2D**. The markers indicate the computed errors. The solid lines are least square regression lines used to compute the convergence rates. The convergence rates are shown in the legends. The resolution is the number of elements across the width of a regular channel. We also run tests on a modified version of "voronoi-s4", which contains velocity boundary conditions only. In that case, the pseudo-inverse is used to eliminate the last block; these tests are indicated with "pinv."

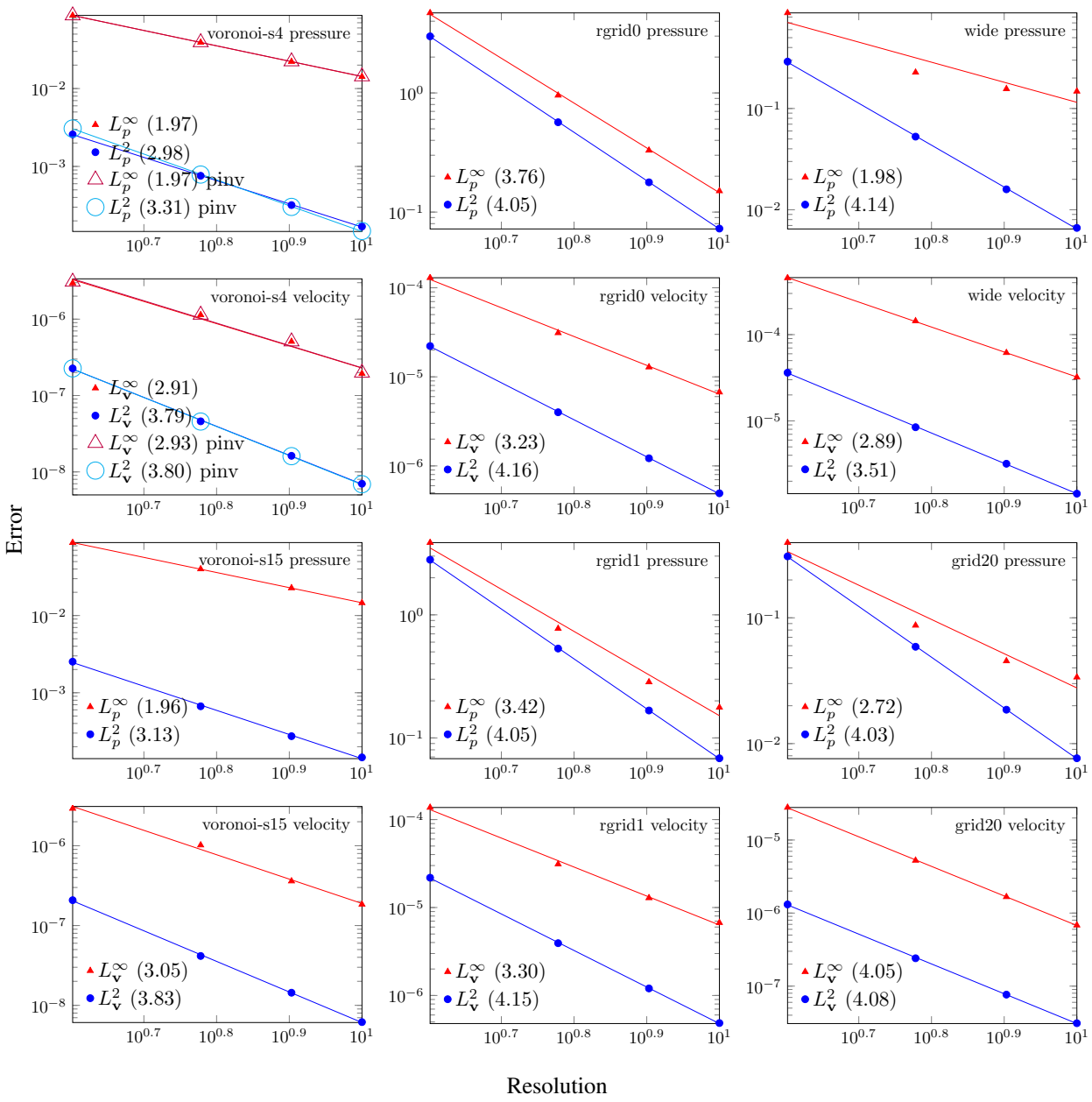


Figure 12: **Analytic** convergence tests for L^∞ and L^2 error measures in **3D**. The markers indicate the computed errors. The solid lines are least square regression lines used to compute the convergence rates. The convergence rates are shown in the legends. The resolution is the number of elements across the width of a regular channel. We also run tests on a modified version of "voronoi-s4", which contains velocity boundary conditions only. In that case, the pseudo-inverse is used to eliminate the last block; these tests are indicated with "pinv."

convergence order for the Taylor-Hood elements that we use in our discretization.

In 3D, memory usage restricts the resolutions to $r = 10$. At this resolution, simulations contain on the order of 10M degrees of freedom. (See Table 4 for precise numbers.)

Nullspace. We repeat test "voronoi-s4" in both 2D and 3D with all traction boundary conditions replaced by velocity boundary conditions. These boundary conditions result in a constant pressure nullspace. This nullspace is handled as described in Section 4.1. The convergence results are shown alongside the original boundary conditions in Figures 11 and 12 and are indicated by "pinv" in the legend. The pressure nullspace has no significant effect on the accuracy, convergence, or performance of the method.

6.3. Convergence tests using real boundary conditions

In the analytic convergence tests, we considered test cases with smooth velocity profiles and pressure profiles everywhere. Real flows around sharp corners, however, may have high velocity gradients in these regions. High velocity gradients are also observed at corners where a Poiseuille flow profile must conform to a traction-free outflow boundary condition. These gradients reduce the convergence order in L^∞, especially with the regular element sizes used in this study. In all tests, inflow ports have a flow rate of $0.005\, m^d\, s^{-1}$ where d is the dimension. The viscosity is $8.9 \times 10^{-4}\, kg\, m^{2-d}\, s^{-1}$. We use the solution at a fine resolution ($r = 60$ for 2D and $r = 16$) as our reference to compute the error. The error is normalized by the maximum magnitude seen in the reference solution. The results of convergence tests for 2D and 3D are shown in Figure 13 and Figure 14. We show the distribution of errors and solution gradients in Figure 15.

6.4. Parallel scaling

In this section we run the tests "grid20", "rgrid0" and "voronoi-s4" at fixed resolution ($r = 16$ in 2D, $r = 6$ in 3D) with different numbers of cores (1-16) to evaluate how well the method scales with available cores. The physical properties are the same as in Section 6.3. Input parameters of the tests are shown in Table 3, and results are shown in Figure 16. A speedup of 9-13 times is observed in 3D when increasing cores from 1 to 16. In 2D, a more modest factor of 5-8 is observed instead. The reduced scaling in 2D is due to the lower computational cost of tasks in 2D (and thus scheduling overhead is relatively more expensive). Test "grid20" is more expensive since it is a larger test with more degrees of freedom. The numbers included in the plots are fit line slopes; a slope of s indicates runtime scaling as $O(c^s)$, where c is the number of cores. $s = -1$ is ideal.

6.5. Scaling with refinement

In this section, we evaluate the scaling of the method with resolution. We again run tests "grid20", "rgrid0" and "voronoi-s4." This time, we fix the core count at 16 and instead vary the resolution. Test parameters are shown in Table 4, and results are shown in Figure 17. The numbers in the plots represent the slope of the fit line. A slope of s indicates runtime $O(r^s)$, where r is the resolution. In 2D, observed scaling is $O(r^{2.1}) - O(r^{2.6})$; this compares very

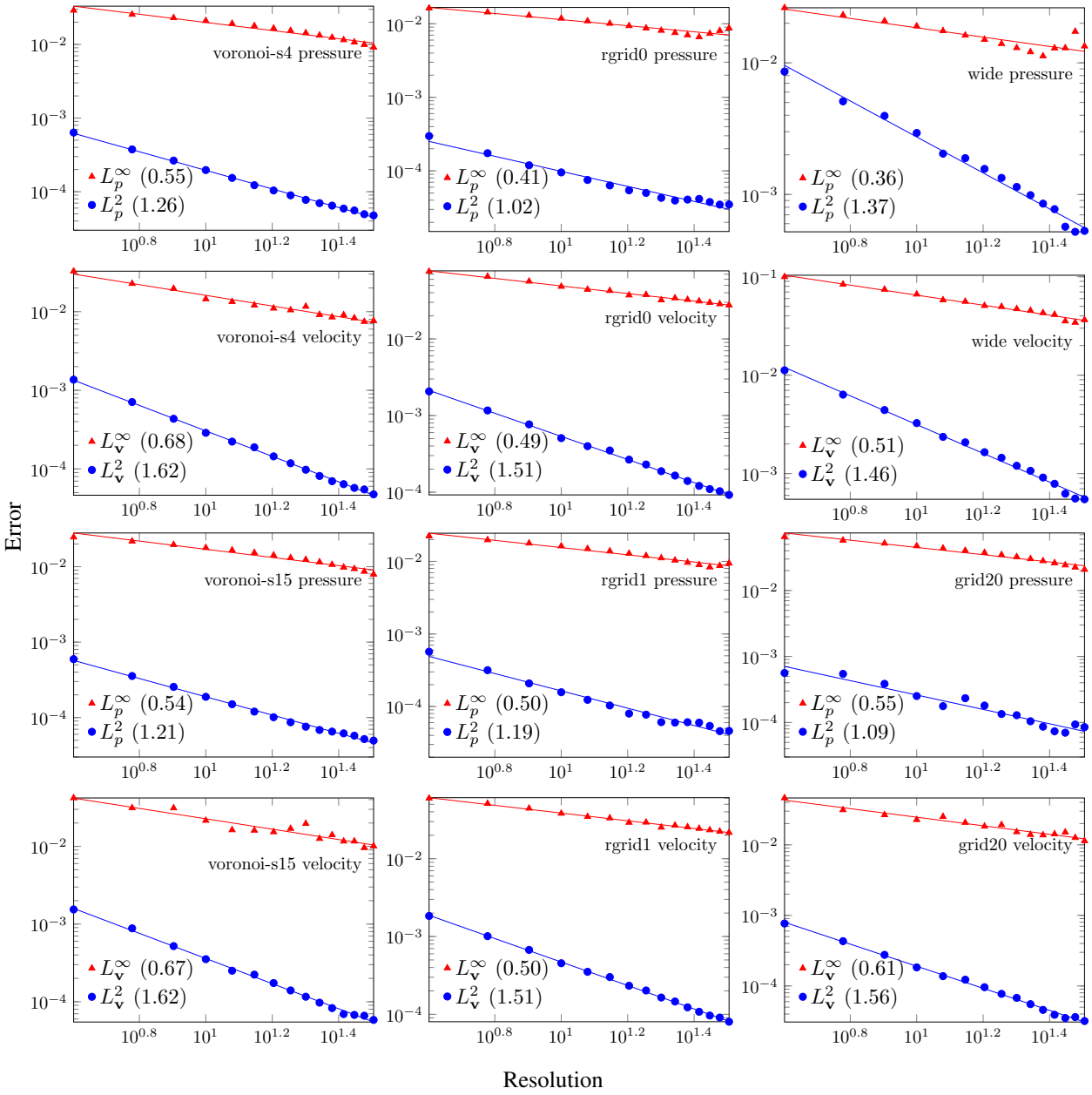


Figure 13: Convergence tests with **real boundary conditions** for L^∞ and L^2 error measures in **2D**. The markers indicate the computed errors. The solid lines are least square regression lines used to compute the convergence rates. The convergence rates are shown in the legends. The resolution is the number of elements across the width of a regular channel.

favorable with ideal $O(r^2)$ and is much better than the predicted $O(r^5)$. In 3D, observed scaling is $O(r^{5.2})$–$O(r^{5.4})$; this is significantly worse than the optimal $O(r^3)$ but also much better than the predicted $O(r^7)$. In each case, observed scaling is far better than one would predict based on the analysis of the algorithm in Section 5.2. Caching is certainly a major factor in the improved scaling, but this alone can only explain a factor of $O(\frac{r}{\ln r})$ improvement. (Optimal asymptotic

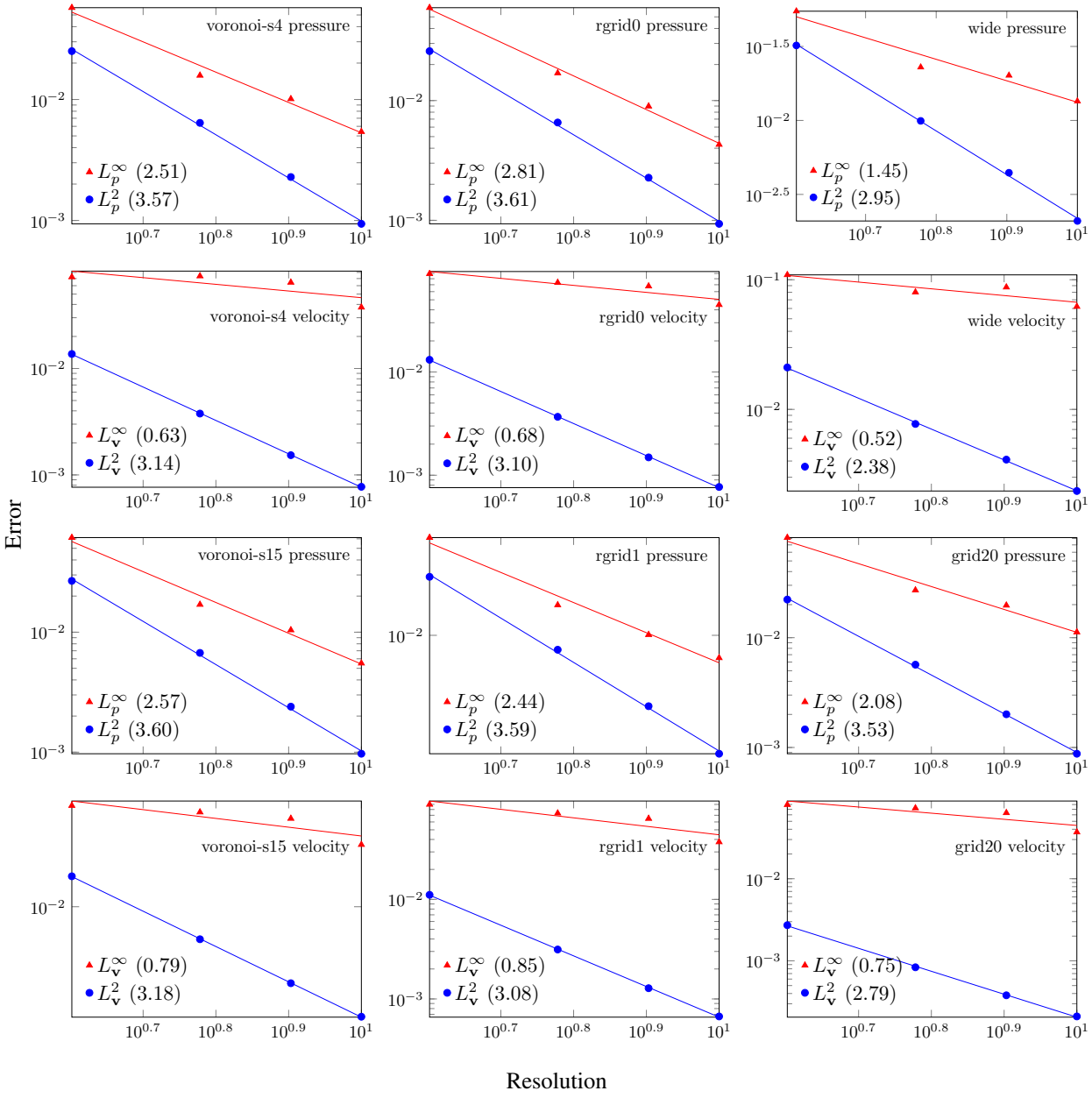
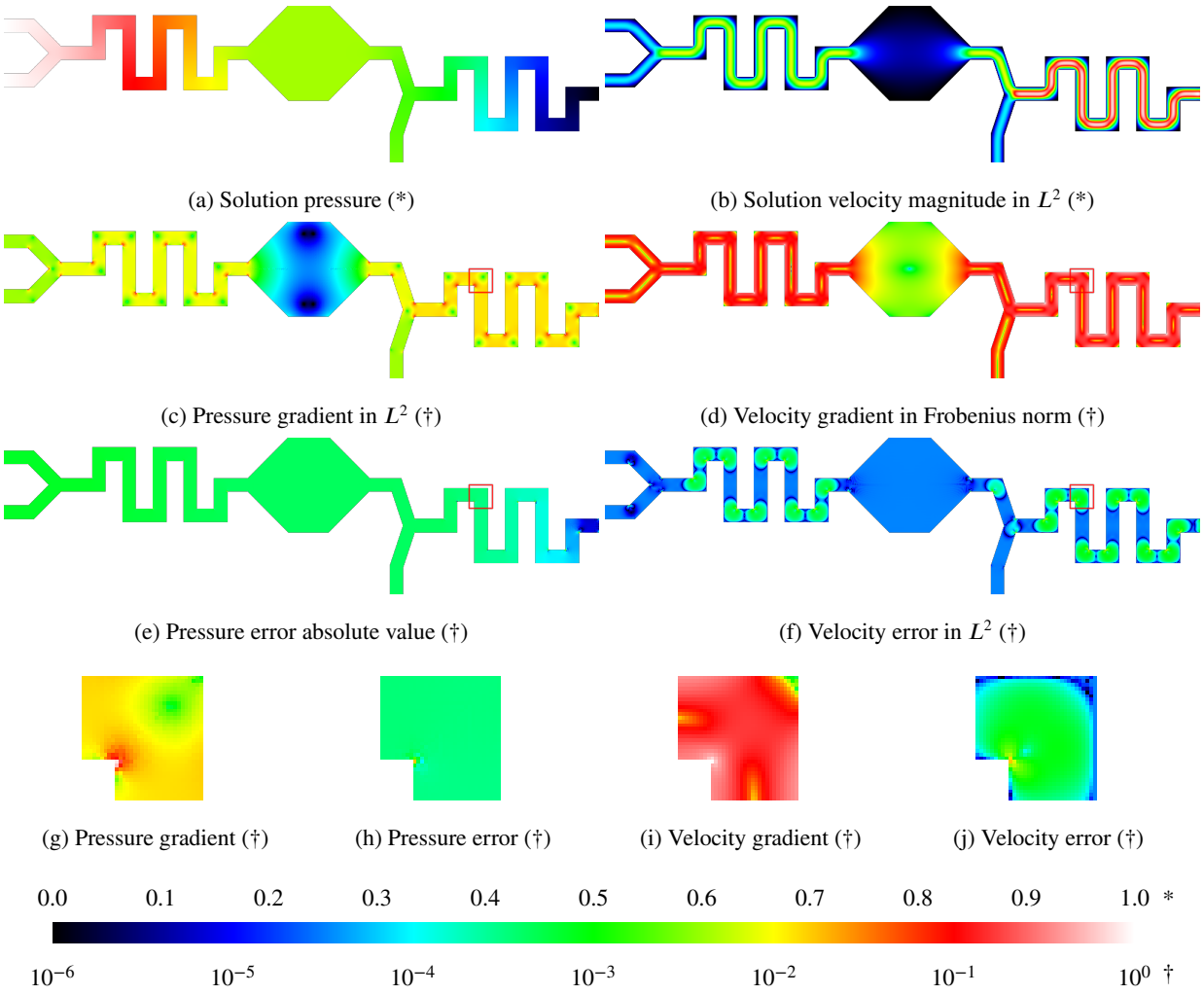


Figure 14: Convergence tests with **real boundary conditions** for L^∞ and L^2 error measures in **3D**. The markers indicate the computed errors. The solid lines are least square regression lines used to compute the convergence rates. The convergence rates are shown in the legends. The resolution is the number of elements across the width of a regular channel.

improvement is obtained for a long pipe, which will have $O(r)$ blocks. The number block operations required for the elimination procedure with caching scales with $O(\ln r)$ compared to $O(r)$ without caching. The optimal asymptotic improvement is thus $O(\frac{r}{\ln r})$, which would be expected for geometry dominated by long pipes.) Improved exploitation of parallelism (SIMD, threading) and better utilization of available memory bandwidth (since larger matrix blocks



(a) Solution pressure (*)

(b) Solution velocity magnitude in L^2 (*)

(c) Pressure gradient in L^2 (†)

(d) Velocity gradient in Frobenius norm (†)

(e) Pressure error absolute value (†)

(f) Velocity error in L^2 (†)

(g) Pressure gradient (†)

(h) Pressure error (†)

(i) Velocity gradient (†)

(j) Velocity error (†)

Figure 15: Solutions (first row), gradients (second row) and errors (third row) for test "wide" at resolution 16. The solutions are in linear scale (*), and the gradients and errors are in logarithmic scale (†). The solutions and gradients are normalized by the maximum values that are evaluated at the center of each element. To compute the errors, we compare the results with the solution at resolution 32. The errors are also normalized by the maximum velocity or pressure magnitude. The gradients and errors enclosed in the red rectangle are shown in the fourth row.

result in more FLOPS per byte of data) with larger problem sizes may also contribute to the improved performance. The near perfect scaling in 2D is quite surprising but very noticeable. The rather poor scaling in 3D is also quite clear. The resolutions for which 3D is manageable are adequate to produce results accurate to 1-3 decimal places (depending on the variable evaluated and the norm chosen), which is likely adequate for prototyping purposes. At higher resolutions, the method is best used as a coarse-grid solver for multigrid. The proposed method effectively performs an LU factorization; subsequent iterations require only the forward/backward substitution.

Name	Resolution	Total dofs	Blocks	Tasks	Avg dofs/block
grid20	16	6.0 M	43080	174.0 K	140.2
rgrid0	16	2.0 M	14334	35.5 K	138.0
voronoi-s4	16	1.3 M	9798	34.7 K	136.2
grid20-3d	6	5.8 M	15470	108.0 K	373.3
rgrid0-3d	6	1.9 M	5281	14.9 K	358.5
voronoi-s4-3d	6	1.3 M	3582	13.8 K	354.5

Table 3: Parameters and task decomposition of **parallel scaling** tests.

6.6. Comparison with general direct sparse solvers

In this section we compare our method with two direct solvers used for general sparse system: MUMPS[31, 32] and UMFPACK[34]. We use test cases "wide", "grid20", "rgrid0" and "voronoi-s4" in both 2D and 3D. We choose a resolution for each test so that there are around one million degrees of freedom. The parameters are listed in Table 5. We compare the methods based on runtime and also scaling with threads.

MUMPS (version `5.2.1`) uses MPI for parallelism. Each instance calls sequential LAPACK routines. We call MUMPS so that it is aware that our system is symmetric indefinite. (MUMPS also supports the shared memory parallelism through OpenMP. We tried this setup with one MPI instance and let the LAPACK implementation spawn threads. This was not as efficient as the MPI approach, so we do not show these results here.)

UMFPACK (version `5.7.8`) supports threading through a parallel LAPACK implementation. In the setup of UMFPACK, we specify the OpenMP threads number. The default parameters are used for the UMFPACK solver.

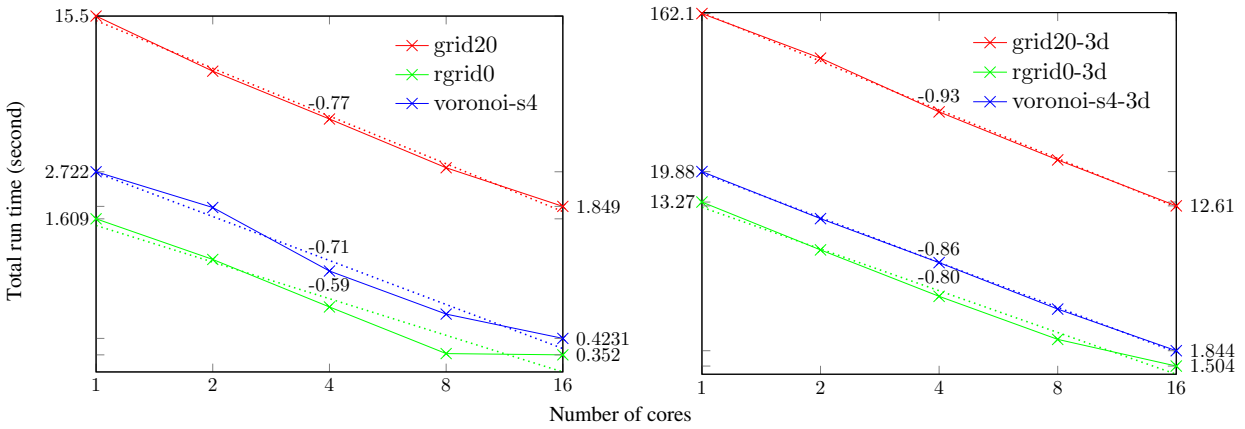


Figure 16: Total solution time as a function of the number of threads plotted on a logarithmic scale. We measure the total run time including meshing, system construction, elimination, and backsolve. Regression lines are shown as dotted lines labeled with their slopes. The run time for 1 and 16 threads are shown on the left and right vertical axes.

Name	Resolution	Total dofs	Blocks	Tasks	Avg dofs/block
grid20	18	7.6 M	48602	187.0 K	157.2
grid20	36	30.9 M	98300	305.7 K	314.4
rgrid0	18	2.5 M	16134	42.8 K	154.9
rgrid0	36	10.1 M	32495	78.3 K	311.7
voronoi-s4	18	1.7 M	11041	38.6 K	153.1
voronoi-s4	36	6.8 M	22226	75.9 K	307.7
grid20-3d	4	1.5 M	9948	94.9 K	154.1
grid20-3d	10	29.7 M	26514	134.3 K	1118.4
rgrid0-3d	4	0.5 M	3438	10.1 K	145.2
rgrid0-3d	10	9.7 M	8855	23.3 K	1089.9
voronoi-s4-3d	4	0.3 M	2340	9.8 K	143.3
voronoi-s4-3d	10	6.5 M	6066	22.1 K	1066.4

Table 4: Parameters and task decomposition of **scaling with refinement** tests. Only the smallest and largest resolutions are shown.

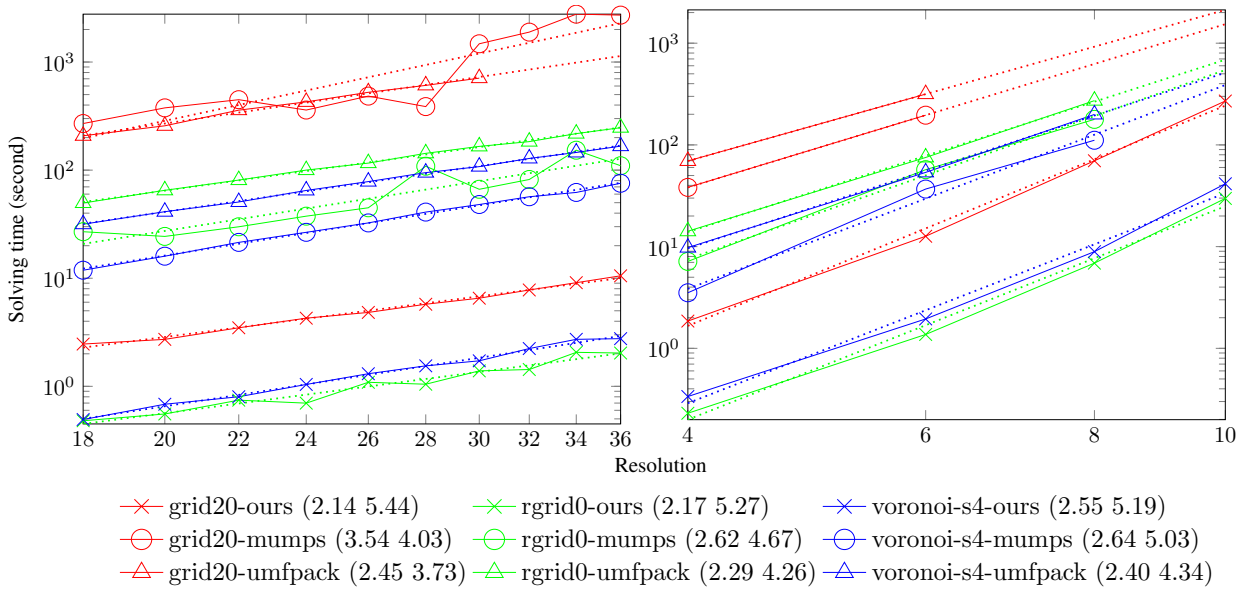


Figure 17: Scaling with refinement using our method (×), MUMPS (○), and UMFPACK (△). Different colors indicate the test cases ("grid20", "rgrid0", or "voronoi-s4"). We run these tests in both 2D (left) and 3D (right). Some data points are missing for MUMPS and UMFPACK because we run out of memory for those resolutions. The resolution refers to the number of edges that each pipe cross section has been divided into, so that the triangles (or tetrahedra) have edge length $h = \frac{w}{r}$, where w is the pipe width and r is the resolution. The dotted lines are least square regression lines used to compute the increasing order. The orders for 2D (first entry) and 3D (second entry) are shown along with the corresponding legend items. An order of s indicates a complexity of $O(n^s)$ as discussed in Section 5.2.

Name	Resolution	Total dofs	Blocks	Tasks	Avg dofs/block
wide	32	1.0 M	3036	16.0 K	314.4
grid20	8	1.5 M	20992	121.3 K	71.9
rgrid0	12	1.1 M	10702	27.2 K	103.9
voronoi-s4	16	1.3 M	9798	34.7 K	136.2
wide-3d	8	0.6 M	714	4.1 K	795.0
grid20-3d	4	1.5 M	9948	94.9 K	154.1
rgrid0-3d	6	1.9 M	5281	14.9 K	358.5
voronoi-s4-3d	6	1.3 M	3582	13.8 K	354.5

Table 5: Parameters and task decomposition for **comparison** tests with MUMPS, UMFPACK, and Krylov solvers. The resolutions are chosen so that the total number of dofs is around one million.

For both MUMPS and UMFPACK, the total solving time is measured for symbolic analysis, numeric factorization, and final numeric solve steps. For both solvers, we solve the world space system, thus avoiding the extra transform passes on the solution and right hand side required for our method. The time required to set up the systems is excluded in all cases; only linear system solve time is being compared. Results are shown in Figure 18. We were surprised to observe that MUMPS and UMFPACK did not scale well with increasing core count. Our method is significantly faster on all tests except "wide." The test case "wide" demonstrates a limitation of our method (See Section 7), though even in this example we eventually catch up with increasing numbers of cores.

6.7. Comparison with iterative solver

The family of Krylov subspace-based solvers is also commonly used for solving general sparse linear systems. In the case of symmetric indefinite matrices, MINRES is frequently used. The cost of Krylov solvers varies considerably, with system conditioning and the effectiveness and cost of the preconditioner being major factors. Rather than compare the cost of solving the system with particular choices of preconditioner, we instead compare the cost of solving the systems with our algorithm with the cost of performing *one iteration of unpreconditioned MINRES*. The cost of a MINRES iteration was estimated by running 10 iterations of MINRES on the linear system and taking the average. Our MINRES implementation uses MKL-BLAS for the vector operations and MKL's sparse matrix-vector routines for the matrix-vector multiply. All of the MINRES linear algebra operations are threaded. We use the same set of tests and setup as in Section 6.6.

The test results are shown in Figure 19. With the exception of the "wide" test, our method converges for the price of about 20 (in 2D) or 60 (in 3D) unpreconditioned Krylov iterations for a Stokes systems with approximately 1M degrees of freedom. On the "wide" test, our cost is equivalent to a bit less than 300 (in 2D) or 1000 (in 3D) unpreconditioned Krylov iterations. Unpreconditioned MINRES would make little progress on these problems in 60 iterations and does

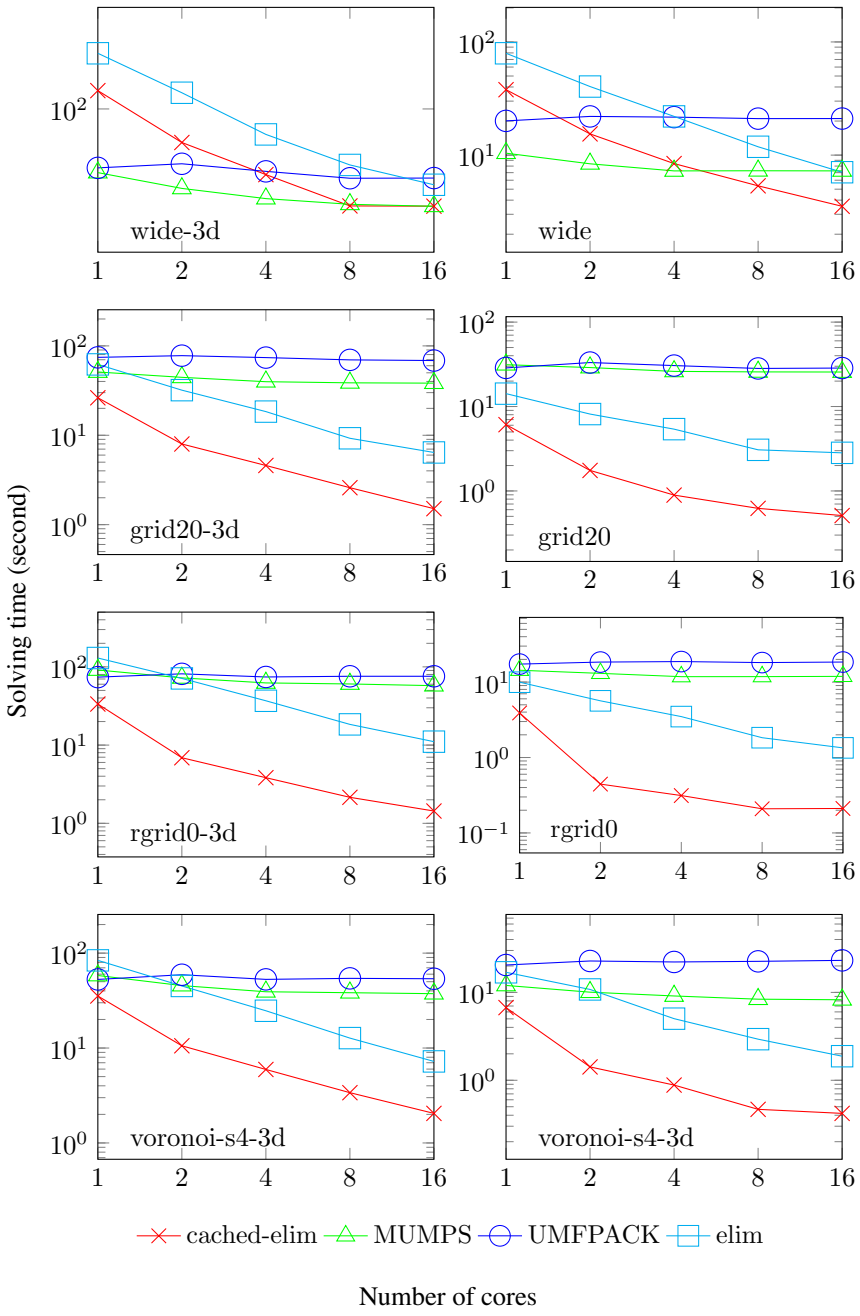


Figure 18: Solving time of our method ("cached-elim"), our method without caching ("elim"), MUMPS, and UMFPACK with different number of threads.

not even converge in 1000; a preconditioner should always be used on these problems. Preconditioners compete for time with the rest of the Krylov iteration. While effective preconditioners exist which can converge in fewer than 60 iterations, doing so at the *cost* of 60 unpreconditioned iterations would be quite difficult. Nevertheless, fair comparisons with iterative methods are tricky. The cost of an iterative method depends on many factors, including the convergence

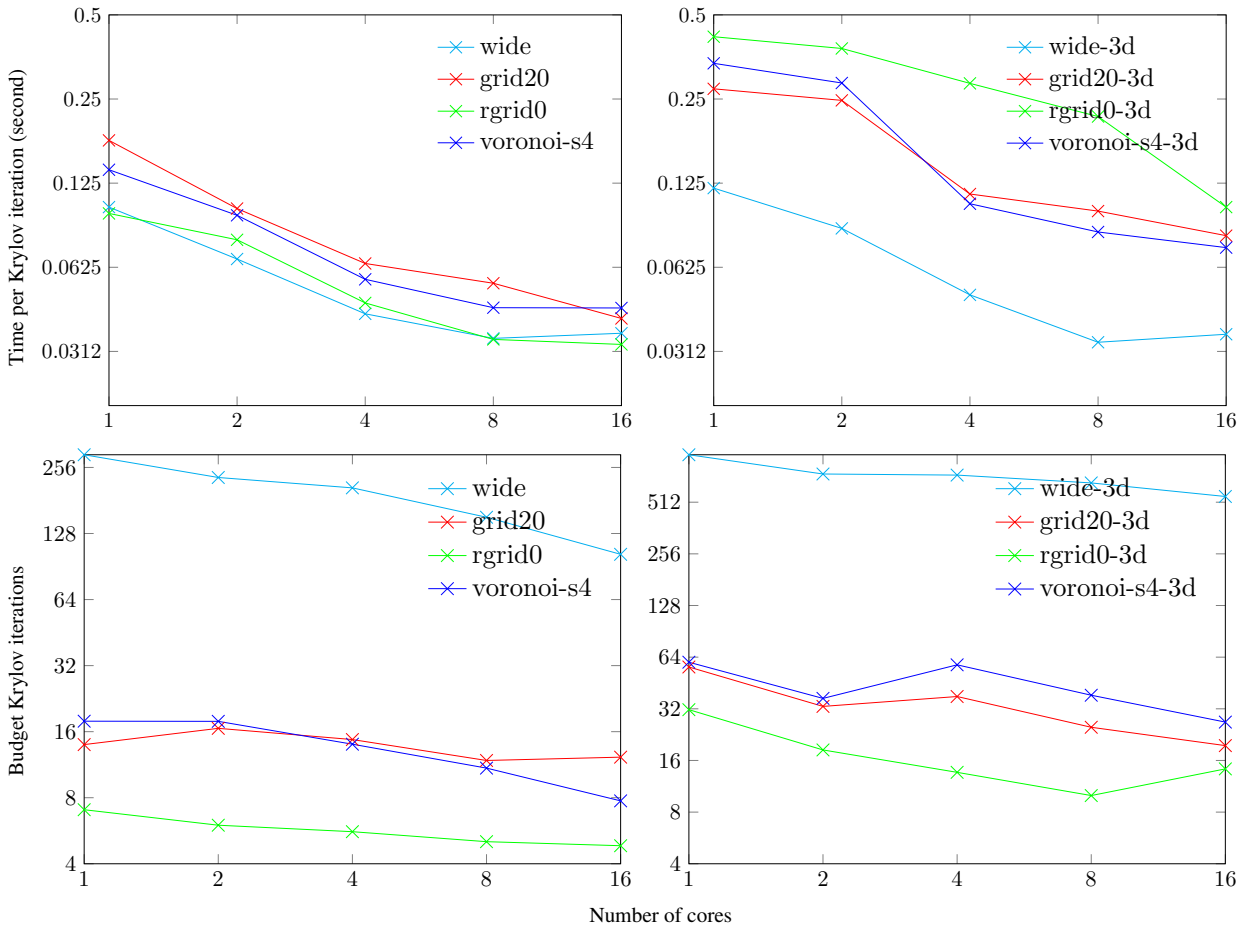


Figure 19: Comparison with Krylov solver. The solving time of our method is converted to number of Krylov iterations based on the time per iteration observed in our reference MINRES solver.

tolerance (full convergence is usually not required), the condition number of the linear system, and the effectiveness and cost of the preconditioner. A good multigrid preconditioner can reduce the residual by an order of magnitude each iteration, but generally there is a trade-off between the effectiveness and cost of the preconditioner.

7. Limitations and future work

Although the proposed method can in principle be applied to arbitrary fluid problems, in practice it is only efficient for fluid domains that have special geometrical properties. Irregularities are well-tolerated, provided they are local and do not lead to blocks with excessive numbers of degrees of freedom. Geometry without repetitions (either in the form of repeated components or straight pipes) produces no caching opportunities and thus no speedup over existing solvers. Though sensitive to the geometry, the method is relatively flexible with respect to PDE (Navier-Stokes, Poisson, heat equation) and other discretization choices (finite volume, finite difference; triangles vs. quads).

Due to the need for repetitions in geometry to be passed on to the linear solver, some amount of special-purpose meshing is required. Pipes must be broken into geometry blocks directly, and canonical components and blocks must still be identified. One is still free to use existing meshing libraries for non-pipe components and for geometry blocks. If a component was meshed with a general purpose library, a simple breadth-first-search traversal of the mesh elements could be used to automatically generate geometry blocks and canonical meshes. The meshing algorithm employed uses uniformly-sized elements, which do not accurately capture the large velocity and pressure gradients that occur in isolated parts of the fluid domain. There is no fundamental reason that adaptive mesh generation could not be employed, and we leave this for future work.

Although our algorithm remains quite efficient at interesting resolutions, the algorithm scales poorly with block matrix size. For high enough resolutions, the algorithm will eventually become slower than many competing methods, especially iterative methods. The algorithm, however, only scales poorly with respect to feature width (channel width). It scales very well with respect to channel length. As long as cross sections can be accurately resolved with at most a few hundred degrees of freedom (components should be not too much wider than pipes, unless they can be accurately resolved with larger elements), the algorithm will scale to devices of very high complexity (hundreds of components) and effectively unlimited total pipe length.

There are many promising avenues for extending and improving the proposed method. The method may be coupled with multigrid to scale to higher resolutions or with other direct solvers to handle wide components more efficiently. Our implementation of the algorithm is quite simple and does not extend naturally to other problem domains besides our specific application, even though other domains exhibit similar geometrical features (e.g., plumbing). Our extension from 2D to 3D assumes that geometry is simply extruded, but we do not exploit this, such as by applying FFTs in this direction as is done in the FACR algorithm [77]. Such a method would improve the 3D scaling almost to the level of 2D scaling both in terms of memory and computational complexity.

8. Conclusions

We have demonstrated how the Stokes equations can be meshed and discretized in microfluidic devices with thin and repetitive geometry. This leads to a linear algebra problem with repeated matrix blocks. We have also constructed an algorithm to efficiently solve linear equations with this structure. The algorithm is very efficient up to moderate resolutions and is competitive with existing methods over this range of resolutions. The proposed method is the most efficient algorithm we are aware of for solving the Stokes flow equations on microfluidic chips. Discretizations with around 1M degrees of freedom can be solved in about one second on a workstation with 16 cores.

Acknowledgements

This work was funded partially through an initial funding package from the University of California Riverside. Ounan Ding was partially funded through a Graduate Division Fellowship from the University of California Riverside.

[1] D. A. Hoang, V. van Steijn, L. M. Portela, M. T. Kreutzer, C. R. Kleijn, Benchmark numerical simulations of segmented two-phase flows in microchannels using the volume of fluid method, Computers & Fluids 86 (2013) 28–36.

[2] W. S. Low, N. A. Kadri, W. Abas, et al., Computational fluid dynamics modelling of microfluidic channel for dielectrophoretic biomems application, The Scientific World Journal 2014.

[3] T. Zhou, T. Liu, Y. Deng, L. Chen, S. Qian, Z. Liu, Design of microfluidic channel networks with specified output flow rates using the cfd-based optimization method, Microfluidics and Nanofluidics 21 (1) (2017) 11.

[4] C. Chang, C. Lai, C.-K. Chung, Numerical analysis and experiments of capillarity-driven microfluid chip, in: 2011 6th IEEE International Conference on Nano/Micro Engineered and Molecular Systems, IEEE, 2011, pp. 1032–1035.

[5] S. Cito, J. Pallares, A. Fabregat, I. Katakis, Numerical simulation of wall mass transfer rates in capillary-driven flow in microchannels, International Communications in Heat and Mass Transfer 39 (8) (2012) 1066–1072.

[6] M. Wörner, Numerical modeling of multiphase flows in microfluidics and micro process engineering: a review of methods and applications, Microfluidics and nanofluidics 12 (6) (2012) 841–886.

[7] T. Glatzel, C. Litterst, C. Cupelli, T. Lindemann, C. Moosmann, R. Niekrawietz, W. Streule, R. Zengerle, P. Koltay, Computational fluid dynamics (cfd) software tools for microfluidic applications–a case study, Computers & Fluids 37 (3) (2008) 218–235.

[8] A. Grimmer, M. Hamidović, W. Haselmayr, R. Wille, Advanced simulation of droplet microfluidics, ACM Journal on Emerging Technologies in Computing Systems (JETC) 15 (3) (2019) 26.

[9] J. Wang, V. G. Rodgers, P. Brisk, W. H. Grover, Instantaneous simulation of fluids and particles in complex microfluidic devices, PloS one 12 (12) (2017) e0189429.

[10] N. Gleichmann, D. Malsch, P. Horbert, T. Henkel, Toward microfluidic design automation: a new system simulation toolkit for the in silico evaluation of droplet-based lab-on-a-chip systems, Microfluidics and Nanofluidics 18 (5-6) (2015) 1095–1105.

[11] N. Gleichmann, D. Malsch, M. Kielpinski, W. Rossak, G. Mayer, T. Henkel, Toolkit for computational fluidic simulation and interactive parametrization of segmented flow based fluidic networks, Chemical Engineering Journal 135 (2008) S210–S218.

[12] A. Grimmer, R. Wille, Designing Droplet Microfluidic Networks: A Toolbox for Designers, Springer, 2019.

[13] J. A. Talbert, A. R. Parkinson, Development of an automatic, two-dimensional finite element mesh generator using quadrilateral elements and bezier curve boundary definition, International Journal for Numerical Methods in Engineering 29 (7) (1990) 1551–1567.

[14] S.-W. Chae, J.-H. Jeong, Unstructured surface meshing using operators, in: Proceedings, 6th International Meshing Roundtable, 1997, pp. 281–291.

[15] T. A. Davis, S. Rajamanickam, W. M. Sid-Lakhdar, A survey of direct methods for sparse linear systems, Acta Numerica 25 (2016) 383–566.

[16] J. Liu, A. George, Computer solution of large sparse positive definite systems, Prentice-Hall, Inc. Englewood Cliffs. NJ 7632 (1981) 1981.

[17] T. A. Davis, J. R. Gilbert, S. I. Larimore, E. G. Ng, Algorithm 836: Colamd, a column approximate minimum degree ordering algorithm, ACM Transactions on Mathematical Software (TOMS) 30 (3) (2004) 377–380.

[18] A. George, J. W. Liu, The evolution of the minimum degree ordering algorithm, Siam review 31 (1) (1989) 1–19.

[19] J. R. Gilbert, C. Moler, R. Schreiber, Sparse matrices in matlab: Design and implementation, SIAM Journal on Matrix Analysis and Applications 13 (1) (1992) 333–356.

[20] A. George, Nested dissection of a regular finite element mesh, SIAM Journal on Numerical Analysis 10 (2) (1973) 345–363.

[21] J. W. Liu, The minimum degree ordering with constraints, SIAM Journal on Scientific and Statistical Computing 10 (6) (1989) 1136–1145.

[22] B. M. Irons, A frontal solution program for finite element analysis, International Journal for Numerical Methods in Engineering 2 (1) (1970) 5–32.

[23] P. Hood, Frontal solution program for unsymmetric matrices, International Journal for Numerical Methods in Engineering 10 (2) (1976) 379–399.

[24] I. S. Duff, J. K. Reid, The multifrontal solution of indefinite sparse symmetric linear, ACM Transactions on Mathematical Software (TOMS) 9 (3) (1983) 302–325.

[25] I. S. Duff, Parallel implementation of multifrontal schemes, Parallel computing 3 (3) (1986) 193–204.

[26] I. S. Duff, The parallel solution of sparse linear equations, in: International Conference on Parallel Processing, Springer, 1986, pp. 18–24.

[27] P. R. Amestoy, L. S. Duff, Vectorization of a multiprocessor multifrontal code, The International Journal of Supercomputing Applications 3 (3) (1989) 41–59.

[28] P. R. Amestoy, I. S. Duff, J.-Y. L'excellent, Multifrontal parallel distributed symmetric and unsymmetric solvers, Computer methods in applied mechanics and engineering 184 (2-4) (2000) 501–520.

[29] C. Ashcraft, R. Grimes, The influence of relaxed supernode partitions on the multifrontal method, ACM Transactions on Mathematical Software (TOMS) 15 (4) (1989) 291–309.

[30] C. Cleveland Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, H. D. Simon, P. E. Bjørstad, Progress in sparse matrix methods for large linear systems on vector supercomputers, The International Journal of Supercomputing Applications 1 (4) (1987) 10–30.

[31] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, J. Koster, A fully asynchronous multifrontal solver using distributed dynamic scheduling, SIAM Journal on Matrix Analysis and Applications 23 (1) (2001) 15–41.

[32] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, S. Pralet, Hybrid scheduling for the parallel solution of linear systems, Parallel Computing 32 (2) (2006) 136–156.

[33] T. A. Davis, I. S. Duff, An unsymmetric-pattern multifrontal method for sparse lu factorization, SIAM Journal on Matrix Analysis and Applications 18 (1) (1997) 140–158.

[34] T. A. Davis, Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method, ACM Transactions on Mathematical Software (TOMS) 30 (2) (2004) 196–199.

[35] T. A. Davis, A column pre-ordering strategy for the unsymmetric-pattern multifrontal method, ACM Transactions on Mathematical Software (TOMS) 30 (2) (2004) 165–195.

[36] R. W. Hockney, A fast direct solution of poisson's equation using fourier analysis, Journal of the ACM (JACM) 12 (1) (1965) 95–113.

[37] B. L. Buzbee, G. H. Golub, C. W. Nielson, On direct methods for solving poissonâĂŹs equations, SIAM Journal on Numerical analysis 7 (4) (1970) 627–656.

[38] K. Hwang, Advanced computer architecture: Parallelism, scalability, programmability:, mcgraw-hill book co. international edition.

[39] P. Arbenz, M. Hegland, et al., The stable parallel solution of general narrow banded linear systems.

[40] G. H. Golub, J. M. Ortega, Scientific computing: an introduction with parallel computing, Elsevier, 2014.

[41] Y. Zhang, J. Cohen, J. D. Owens, Fast tridiagonal solvers on the gpu, ACM Sigplan Notices 45 (5) (2010) 127–136.

[42] W. Gander, G. H. Golub, Cyclic reduction–history and applications, Scientific computing (Hong Kong, 1997) (1997) 73–85.

[43] M. R. Hestenes, E. Stiefel, Methods of conjugate gradients for solving linear systems, Vol. 49, NBS Washington, DC, 1952.

[44] C. C. Paige, M. A. Saunders, Solution of sparse indefinite systems of linear equations, SIAM journal on numerical analysis 12 (4) (1975) 617–629.

[45] Y. Saad, M. H. Schultz, Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM Journal on scientific and statistical computing 7 (3) (1986) 856–869.

[46] S. Kaniel, Estimates for some computational techniques in linear algebra, Mathematics of Computation 20 (95) (1966) 369–378.

[47] A. Van der Sluis, H. A. van der Vorst, The rate of convergence of conjugate gradients, Numerische Mathematik 48 (5) (1986) 543–560.

[48] G. H. Golub, C. F. Van Loan, Matrix computations, Vol. 3, JHU Press, 2012.

[49] B. Smith, P. Bjorstad, W. D. Gropp, W. Gropp, Domain decomposition: parallel multilevel methods for elliptic partial differential equations, Cambridge university press, 2004.

[50] V. Dolean, P. Jolivet, F. Nataf, An introduction to domain decomposition methods: algorithms, theory, and parallel implementation, Vol. 144, SIAM, 2015.

[51] W. H. Press, Numerical recipes 3rd edition: The art of scientific computing, Cambridge university press, 2007.

[52] M. T. Heath, Scientific computing, McGraw-Hill New York, 2002.

[53] G. Wittum, Multi-grid methods for stokes and navier-stokes equations, Numerische Mathematik 54 (5) (1989) 543–563.

[54] A. McAdams, E. Sifakis, J. Teran, A parallel multigrid poisson solver for fluids simulation on large grids, in: Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, Eurographics Association, 2010, pp. 65–74.

[55] U. Ghia, K. N. Ghia, C. Shin, High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method, Journal of computational physics 48 (3) (1982) 387–411.

[56] D. J. Mavriplis, A. Jameson, Multigrid solution of the navier-stokes equations on triangular meshes, AIAA journal 28 (8) (1990) 1415–1425.

[57] D. J. Mavriplis, Multigrid solution of the two-dimensional euler equations on unstructured triangular meshes, AIAA journal 26 (7) (1988) 824–831.

[58] X. Lv, Y. Zhao, X. Huang, G. Xia, X. Su, A matrix-free implicit unstructured multigrid finite volume method for simulating structural dynamics and fluid–structure interaction, Journal of Computational Physics 225 (1) (2007) 120–144.

[59] J. Bolz, I. Farmer, E. Grinspun, P. Schröoder, Sparse matrix solvers on the gpu: conjugate gradients and multigrid, in: ACM transactions on graphics (TOG), Vol. 22, ACM, 2003, pp. 917–924.

[60] G. Haase, M. Liebmann, C. C. Douglas, G. Plank, A parallel algebraic multigrid solver on graphics processing units, in: High performance computing and applications, Springer, 2010, pp. 38–47.

[61] H. Liu, N. Mitchell, M. Aanjaneya, E. Sifakis, A scalable schur-complement fluids solver for heterogeneous compute platforms, ACM Transactions on Graphics (TOG) 35 (6) (2016) 201.

[62] M. Kronbichler, A. Diagne, H. Holmgren, A fast massively parallel two-phase flow solver for microfluidic chip simulation, The International Journal of High Performance Computing Applications 32 (2) (2018) 266–287.

[63] D. Di Carlo, Inertial microfluidics, Lab on a Chip 9 (21) (2009) 3038–3046.

[64] D. Assêncio, D. Teran, A second order virtual node algorithm for stokes flow problems with interfacial forces, discontinuous material parameters and irregular domains, J. Comput. Phys. 250 (1) (2013) 77–105.

[65] C. Schroeder, A. Stomakhin, R. Howes, J. Teran, A second order virtual node algorithm for navier-stokes flow problems with interfacial forces and discontinuous material properties, J. Comput. Phys. 265 (2014) 221–245.

[66] D. Boffi, F. Brezzi, M. Fortin, et al., Mixed finite element methods and applications, Vol. 44, Springer, 2013.

[67] A. Ern, J.-L. Guermond, Theory and practice of finite elements, Vol. 159, Springer Science & Business Media, 2013.

[68] D. Boffi, N. Cavallini, F. Gardini, L. Gastaldi, Local mass conservation of stokes finite elements, Journal of scientific computing 52 (2) (2012) 383–400.

[69] M. Bercovier, O. Pironneau, Error estimates for finite element method solution of the stokes problem in the primitive variables, Numerische Mathematik 33 (2) (1979) 211–224.

[70] T. L. Adam, K. M. Chandy, J. Dickson, A comparison of list schedules for parallel processing systems, Communications of the ACM 17 (12) (1974) 685–690.

[71] A. Gerasoulis, T. Yang, A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors, Journal of Parallel and Distributed Computing 16 (4) (1992) 276–291.

[72] I. Ahmad, Y.-K. Kwok, M.-Y. Wu, Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors, in: Proceedings Second International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'96), IEEE, 1996, pp. 207–213.

[73] J. J. Lambiotte Jr, R. G. Voigt, The solution of tridiagonal linear systems on the cdc star 100 computer, ACM Transactions on Mathematical Software (TOMS) 1 (4) (1975) 308–329.

[74] D. Heller, Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems, SIAM Journal on Numerical Analysis 13 (4) (1976) 484–496.

[75] J. Wang, P. Brisk, W. H. Grover, Random design of microfluidics, Lab Chip 16 (21) (2016) 4212–4219. doi:10.1039/c6lc00758a.

[76] P. J. Roache, Code verification by the method of manufactured solutions, Journal of fluids engineering 124 (1) (2002) 4–10.

[77] P. N. Swarztrauber, The methods of cyclic reduction, fourier analysis and the facr algorithm for the discrete solution of poisson's equation on a rectangle, Siam Review 19 (3) (1977) 490–501.