DELFT UNIVERSITY OF TECHNOLOGY

REPORT 06-01

An Efficient Deflation Method applied on 2-D and 3-D Bubbly Flow Problems

J.M. Tang, C. Vuik

ISSN 1389-6520

Reports of the Department of Applied Mathematical Analysis

Delft 2006

AN EFFICIENT DEFLATION METHOD APPLIED ON 2-D AND 3-D BUBBLY FLOW PROBLEMS

J.M. TANG AND C. VUIK

Abstract. Simulating bubbly flows is a very popular topic in CFD. These bubbly flows are governed by the Navier-Stokes equations. In many popular operator splitting formulations for these equations, solving the linear system coming from the discontinuous Poisson equation takes the most computational time, despite of its elliptic nature. ICCG is widely used for this purpose, but for complex bubbly flows this method shows slow convergence.

As alternative for ICCG, we apply a deflated variant of ICCG which is called DICCG. This new method incorporates the eigenmodes corresponding to the components which caused the slow convergence of ICCG. Some implementation issues of DICCG are discussed in this paper and some considerations about the singularity are made. Moreover, coarse linear systems have to be solved within DICCG. We discuss methods to do this efficiently which results in two approaches DICCG1 and DICCG2. In exact arithmetics, we prove that these variants lead to the same convergence results. Thereafter we show with numerical experiments that both DICCG approaches are very efficient. Compared to ICCG, DICCG decreases significantly the number of iterations and the computational time as well, which are required for solving Poisson equation in applications of 2-D and 3-D bubbly flows.

Keywords. deflation, conjugate gradient method, preconditioning, Poisson equation, symmetric positive semi-definite matrices, bubbly flow problems.
AMS subject classifications. 65F10, 65F50, 65N22.

Date: March 17, 2006.

Contents

List of Figures

List of Tables

1. Introduction

Numerical simulation of bubbly flows are relevant to problems found in various disciplines such as oil, nuclear and chemical industry. These bubbly flows are governed by the incompressible Navier-Stokes equations. Efficient solution of these Navier-Stokes equations in complex domains depends upon the availability of fast solvers for sparse linear systems. Most popular in use are the Krylov subspace iterative solvers, like the CG method with the incomplete Cholesky preconditioner denoted by ICCG [5].

Figure 1. An example of a bubbly flow problem: a water splash.

For bubbly flows, the pressure operator is often the leading contributor to stiffness. This pressure operator is coming from the discontinous Poisson equation. In the context of operator splitting formulations for the Navier-Stokes equations, it is the Poisson solve which is the most computationally challenging despite its elliptic origins, see also [8, 9, 12].

We seek to improve the ICCG method for the pressure solve in order to overcome the slow convergence, frequently observed in the presence of highly refined grids and flows with high density ratio's or with many bubbles. The presence of small eigenvalues have a harmful influence on the convergence of ICCG. These slow converging components are not cured by preconditioning. A significant improvement consists of the removal of the eigenmodes corresponding to these small eigenvalues out of the system. This leads to the DICCG method. This method is equal to ICCG which is extended with a deflation technique [7].

In many applications, DICCG has been proven to be an efficient method, such as in applications of porous media flows [6] and ground water flows [15]. However, in these cases, the interfaces in the domain can be described explicitly so that applying the deflation technique is straightforward. Due to the possible appearance of complex geometries in our problems of bubbly flows, the interfaces of the bubbles can only be described implicitly in general, making the deflation technique more sophisticated to apply.

Recently, DICCG applied on bubbly flows has been studied by the authors [13, 14]. In that paper theoretical considerations of DICCG are given with respect to the singularity of the linear system. Moreover, some 3-D numerical experiments have been performed, where ICCG and DICCG have been compared by considering the number of iterations required for convergence to the solution. However, it is known that a reduction of the number of iterations does not guarantee a reduction of the required computational time, since the work per iteration may increase in the new method. Therefore, in this paper we present new results by investigating both the number of iterations and the required CPU time. To do so, DICCG has to be programmed efficiently which is not straightforward. Hence, in this paper we will consider some implementation issues in more detail in order to obtain an efficient programming code.

This paper is organized as follows. First the problem setting of the bubbly flow is given in Section 2. Subsequently, we describe shortly the ICCG and DICCG methods in Section 3. After that, in Section 4, DICCG will be treated in more detail, especially the operations with the deflation matrix will be investigated. Next, efficient methods to solve the coarse linear systems with DICCG will be proposed in Section 5. Thereafter, a theoretical comparison of these efficient methods will be made in Section 6. Section 7 and 8 are devoted to the 2-D and 3-D numerical experiments to investigate the performance of both ICCG and DICCG. Moreover, a real-life applications of a rising bubble in water and a falling droplet in air will be considered in Section 9, where both ICCG and DICCG will be compared. Finally, the conclusions will be presented in Section 10.

2. Problem Setting of the Bubbly Flow

We consider the singular SPSD (symmetric and positive semi-definite) linear system

$$Ax = b, \quad A \in \mathbb{R}^{n \times n}. \tag{1}$$

The linear system (1) is derived after a second-order finite-difference discretization of the 2-D or 3-D Poisson equation with Neumann boundary conditions, which is

$$\left\{ \begin{array}{rcll} -\nabla \cdot \left(\frac{1}{\rho(\mathbf{x})} \nabla p(\mathbf{x}) \right) & = & f(\mathbf{x}), & \mathbf{x} \in \Omega, \\ \frac{\partial}{\partial \mathbf{n}} p(\mathbf{x}) & = & g(\mathbf{x}), & \mathbf{x} \in \partial\Omega, \end{array} \right. \tag{2}$$

where $p, \rho, \mathbf{x}$ and $\mathbf{n}$ denote the pressure, density, spatial coordinates and the unit normal vector to the boundary $\partial\Omega$, respectively. In the 2-D case, domain Ω is chosen to be a unit box, whereas Ω is a unit cube in 3-D. We apply the computations on a uniform Cartesian grid, so that $n = N_x \cdot N_y$ in 2-D and $n = N_x \cdot N_y \cdot N_z$ in 3-D, where N_x, N_y and N_z are the grid sizes in each spatial direction. Furthermore, we consider two-phase bubbly flows with air and water. In this case, ρ is piecewise constant with a relatively high contrast:

$$\rho = \left\{ \begin{array}{ll} \rho_0 = 1, & \mathbf{x} \in \Lambda_0, \\ \rho_1 = 10^{-3}, & \mathbf{x} \in \Lambda_1, \end{array} \right. \tag{3}$$

where Λ_0 is water, the main fluid of the flow around the air bubbles, and Λ_1 is the region inside the bubbles. In the numerical experiments, we will take bubbles with equal radius and which are well-structured in Λ_0. In Figure 2 one can find a plot in the case of such a problem with eight bubbles. In the numerical experiments, we will also consider other two-phase flows instead of the air-water flows. In these cases, the contrast between the densities ρ_0 and ρ_1 will be larger, resulting in a more ill-conditioned linear system (1).

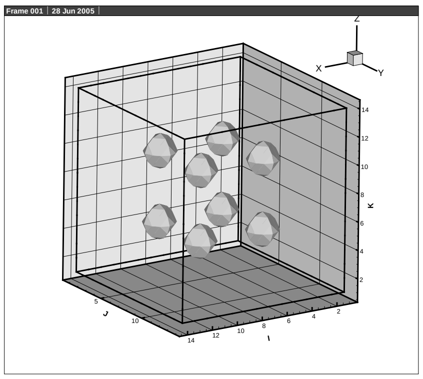


Figure 2. Geometry of a 3-D bubbly flow problem with eight air bubbles in the unit domain filled with water.

Subsequently, inhomogeneous Neumann boundary conditions (i.e., $g(\mathbf{x}) \neq 0$) and no source term (i.e., $f(\mathbf{x}) = 0$) are imposed, so that vector b has only contributions from the boundary conditions.

Finally, define $\mathbf{1}_p$ and $\mathbf{0}_p$ by the all-one and all-zero vector with p elements, respectively. Then, through this paper the following assumption holds which follows implicitly from the above problem setting.

Assumption 1. *The singular SPSD matrix A and the vector b have the following properties.*

(*i*) $A\mathbf{1}_n = \mathbf{0}_n$.
(*ii*) *The algebraic multiplicity of the zero-eigenvalue of A is equal to one.*
(*iii*) $b^T\mathbf{1}_n = 0$.

Obviously, from (*i*) and (*iii*), it follows that the $Ax = b$ is compatible. Hence, although A is singular, this linear system is always consistent and an infinite number of solutions x exists.

3. ICCG and DICCG algorithms

3.1. **ICCG.**

The construction of the incomplete Cholesky (IC) preconditioner $M = C^TC$ can be found in e.g. [3, Sect. 10.3.2]. Matrix C is lower triangular with the same sparsity pattern as A. The resulting linear system which has to be solved is

$$M^{-1}Ax = M^{-1}Ab.$$

Although A is singular, it can be shown that the IC preconditioner M is invertible [2, Th. 3.2]. Next, the ICCG algorithm is given in Algorithm 1.

Algorithm 1 ICCG Algorithm solving $Ax = b$

```
1: compute $r_0 := b - Ax_0$,
   solve $Mv_1 = r_0$ and compute $p_1 := v_1$
2: for $j := 1, \ldots,$ until convergence do
3:    $w_j := Ap_j$
4:    $\alpha_j := (r_j, v_j)/(p_j, w_j)$
5:    $x_{j+1} := x_j + \alpha_j p_j$
6:    $r_{j+1} := r_j - \alpha_j w_j$
7:    solve $Mv_{j+1} = r_{j+1}$
8:    $\beta_j := (r_j, v_j)/(r_{j-1}, v_{j-1})$
9:    $p_{j+1} := v_{j+1} + \beta_j p_j$
10: end for
11: $x := x_{j+1}$
```

3.2. **DICCG.**

Before giving the algorithm of DICCG, we start with defining and giving properties of the deflation subspace matrix Z, the coarse matrix E and the deflation matrix P, see also [14].

3.2.1. *Definition of Z.*

Let the computational domain Ω be divided into open subdomains Ω_j, $j = 1, 2, \ldots, k$, such that $\Omega = \cup_{j=1}^k \overline{\Omega}_j$ and $\cap_{j=1}^k \Omega_j = \emptyset$ where $\overline{\Omega}_j$ is Ω_j including its adjacent boundaries. The discretized domain and subdomains are denoted by Ω_h and Ω_{h_j}, respectively. Then, for each Ω_{h_j} with $j = 1, 2, \ldots, k$, we introduce a deflation vector z_j as follows:

$$(z_j)_i := \begin{cases} 0, & x_i \in \Omega_h \setminus \overline{\Omega}_{h_j}; \\ 1, & x_i \in \Omega_{h_j}, \end{cases}$$

where x_i is a grid point in the discretized domain Ω_h. Then we define

$$Z := [z_1 \; z_2 \; \cdots \; z_k] \in \mathbb{R}^{n \times k}.$$

This implies immediately

$$Z\mathbf{1}_k = \mathbf{1}_n. \tag{4}$$

Hence, Z consists of orthogonal disjunct piecewise-constant vectors.

3.2.2. *Definition of P and E.* Matrices E and P are defined as follows [7]:

$$P := I - AZE^{-1}Z^T \in \mathbb{R}^{n\times n}, \quad E := Z^TAZ \in \mathbb{R}^{k\times k}. \tag{5}$$

Note that E^{-1} is just a notation, since it will never be determined explicitly in the computations. In addition, determining E^{-1} is also not possible in our case, since A is singular and therefore E is singular as well due to

$$E\mathbf{1}_k = Z^TAZ\mathbf{1}_k = Z^TA\mathbf{1}_n = Z^T\mathbf{0}_n = \mathbf{0}_k, \tag{6}$$

where we have applied Eq. (4) and Assumption 1. This means that direct computations with E requires an invertible matrix A and iterative computations with E have to satisfy consistency conditions. In fact, matrix E^{-1} is known as the Moore-Penrose inverse or pseudo-inverse [1]. This will be investigated extensively in Section 4.

Now, in DICCG we have to solve the resulting linear system

$$M^{-1}PA\tilde{x} = M^{-1}Pb.$$

From $\tilde{x}$ we can find solution x using the following expression (see e.g. [6]):

$$x = ZE^{-1}Z^Tb + P^T\tilde{x}.$$

3.2.3. *The DICCG Algorithm.* Now, the algorithm of DICCG is presented in Algorithm 2.

Algorithm 2 DICCG Algorithm solving $Ax = b$

```
 1: compute $r_0 := b - Ax_0$ and $\hat{r}_0 := Pr_0$,
    solve $Mv_1 = \hat{r}_0$ and compute $p_1 := v_1$
 2: for $j := 1, \ldots,$ until convergence do
 3:   $w_j := PAp_j$
 4:   $\alpha_j := (\hat{r}_j, v_j)/(p_j, w_j)$
 5:   $\tilde{x}_{j+1} := \tilde{x}_j + \alpha_j p_j$
 6:   $\hat{r}_{j+1} := \hat{r}_j - \alpha_j w_j$
 7:   solve $Mv_{j+1} = \hat{r}_{j+1}$
 8:   $\beta_j := (\hat{r}_j, v_j)/(\hat{r}_{j-1}, v_{j-1})$
 9:   $p_{j+1} := v_{j+1} + \beta_j p_j$
10: end for
11: $x := ZE^{-1}Z^Tb + P^T\tilde{x}_{j+1}$
```

[1]Since A is symmetric, it is also diagonalizable where we can write

$$A = V\Lambda V^T, \quad V = [v_1\ v_2\ \cdots\ v_n], \quad \Lambda = \mathrm{diag}(0, \lambda_2, \ldots, \lambda_n),$$

with Λ is the diagonal matrix consisting of the eigenvalues of A on the diagonal satisfying $\lambda_2, \ldots, \lambda_n > 0$ and V is an orthonormal matrix consisting of the corresponding eigenvectors. Then, the real inverse can be written as $A^{-1} = V\Sigma V^T$ with $\Sigma = \Lambda^{-1}$, if A is invertible. In our case, A is singular, so that the real inverse is not defined. Instead, the Moore-Penrose inverse or pseudo-inverse is defined by

$$A^{-1} := V\Sigma^+V^T, \quad \Sigma^+ = \mathrm{diag}(0, 1/\lambda_2, \ldots, 1/\lambda_n).$$

Note that in this paper we use the same notation for both the real and pseudo-inverse when there is no ambiguity. Now, it can easily be shown that a solution of the singular and consistent linear system $Ax = b$ is

$$x = A^{-1}b = V\Sigma^+V^Tb.$$

Note from Algorithm 1 and 2 that v_j represents the updated preconditioned residuals in ICCG, whereas v_j are the updated deflated-preconditioned residuals in DICCG.

Furthermore, operations with P in Algorithm 2 should be treated carefully during the implementation, since the success of DICCG depends strongly on the way of implementation of these operations. In the next sections, we treat this topic in more detail.

3.3. **Termination Criterions for ICCG and DICCG.** In the original method ICCG, we terminate the iterative process if the relative preconditioned residuals are smaller than a stopping tolerance $\epsilon > 0$, i.e., for ICCG we take the following termination criterion:

$$\frac{||M^{-1}(b - Ax_k)||_2}{||M^{-1}(b - Ax_0)||_2} < \epsilon. \tag{7}$$

This is equivalent to the following termination criterion in DICCG:

$$\frac{||M^{-1}P(b - A\tilde{x}_k)||_2}{||M^{-1}(b - Ax_0)||_2} < \epsilon, \tag{8}$$

since it is easy to see that $b - Ax_k = P(b - A\tilde{x}_k)$. Note that the deflation matrix P appears only once in Eq. (8).

4. Treatment of Operations with Deflation Matrix

Since the iterative method DICCG should be implemented efficiently in a programming code to obtain a fast solver, some implementation issues considering DICCG will be treated below. Details on flop calculations, which are given below, can be found in Appendix C and D.

4.1. **Construction of AZ.** The matrix-vector product AZ is computed by only determining the non-zero elements, which will be stored as a sparse matrix denoted by S_{AZ}. Denote γ the number of non-zero elements of the full matrix AZ. Then, S_{AZ} is a $\gamma \times 3$ matrix, where the first and second columns are filled with the row and column number of the non-zero elements of AZ, respectively. The third column of S_{AZ} is devoted to the corresponding value of these non-zero elements. Therefore, each row of S_{AZ} consists of two integers and one double.

Determining the elements of S_{AZ} can be done efficiently, by noting that in the 2-D case and in the 3-D case Z represents non-overlapping blocks and cubes, respectively. Moreover, AZ has only contributions near the interfaces of these blocks and hence it consists of relatively many zeros. So the participating elements of AZ are known beforehand. Using Assumption 1, we can determine the value of each of those elements immediately.

For an extensive treatment of the efficient construction of S_{AZ} and the computation of γ for both the 2-D and 3-D case, we refer to Appendix A and B. In addition, considering the number of floating point operations, it appears that constructing S_{AZ} requires $\mathcal{O}(\sqrt{nk})$ flops for the 2-D case, while $\mathcal{O}(n^{2/3}k^{1/3})$ flops are required in the 3-D case.

4.2. **Construction of E and E^{-1}.** The coarse matrix $E := Z^T AZ$ can be formed easily during the construction of AZ. Each non-zero element of AZ has exactly one contribution to E by simply adding the value to the corresponding position in E. If this position is in the interior of the participating block than the contribution is to the main diagonal of E, otherwise it contributes to one of the subdiagonals of E.

It depends on the choice of solving the linear system $Ey_2 = y_1$ in which sparse way E is stored resulting in S_E. Moreover, E^{-1} will never be determined explicitly, as earlier mentioned. We treat this extensively in Section 5. Some implementation remarks about S_E, which are filled with doubles, are also made in Appendix A and B.

4.3. **Construction of Matrix-Vector Products Py and P^Ty.** In contrast to AZ and E, the deflation matrix P is not constructed explicitly. Instead, the matrix-vector product $Py := y - AZE^{-1}Z^Ty$ will be computed in several steps, which are summed up in Algorithm 3. Similarly, P^Ty can be treated in the same way, see Algorithm 4. Both algorithms require $\mathcal{O}(n)$ flops in the 2-D and 3-D case.

Algorithm 3 Algorithm computing Py

```
1: $y_1 := Z^T y$
2: solve $Ey_2 = y_1$
3: $y_3 := AZy_2$
4: $Py := y - y_3$
```

Algorithm 4 Algorithm computing P^Ty

```
1: $y_1 := (AZ)^T y$
2: solve $Ey_2 = y_1$
3: $y_3 := Zy_2$
4: $P^Ty := y - y_3$
```

Note that Z is not stored explicitly, since the matrix-vector products Z^Ty and Zy_2 can be simply determined from y, requiring $\mathcal{O}(n)$ flops. Furthermore, since S_{AZ} is known, the products AZy_2 and $(AZ)^Ty$ can also be computed easily using $\mathcal{O}(\sqrt{nk})$ flops in 2-D and $\mathcal{O}(n^{2/3}k^{1/3})$ flops in 3-D. Finally, solving the linear system $Ey_2 = y_1$ will be treated in the next section.

5. Solving the Coarse Linear system Efficiently

In each iterate of DICCG, we have to solve the coarse linear system $Ey_2 = y_1$. Below, we give two alternatives to do this. We define k_x to be the number of grid points in each spatial direction of a block or cube from Z, i.e., $k_x := \sqrt{n/k}$ in 2-D and $k_x := \sqrt[3]{n/k}$ in 3-D, by assuming that k is a divisor of n and that the number of grid points are equal in each spatial direction.

5.1. **Solving $Ey_2 = y_1$ directly.** If k is relatively small, we can solve $Ey_2 = y_1$ with a direct method, using the band-Cholesky decomposition [3, Sect. 4.3.5] and thereafter the band-back/forward substitution [3, Sect. 4.3.2]. In this case, S_E is a full matrix with dimensions $k \times (k_x + 1)$ in the 2-D case and $k \times (k_x^2 + k_x + 1)$ in 3-D. Therefore, determining the band-Cholesky is especially efficient in the 2-D case, since the bandwidth is just k_x. In the 3-D, the bandwidth is $k_x^2 + k_x$ making the decomposition more expensive, but it can still be efficient for relatively small k and/or n.

In the previous section we noted that E^{-1} does not exist in Eq. (5). Hence, the band-Cholesky decomposition does also not exist due to the singularity of E. Now, instead of solving the singular SPSD linear system $Ax = b$, we attempt to solve the invertible SPD linear system

$$\widetilde{A}x = b, \tag{9}$$

where $\widetilde{A}$ is exactly equal to A except for the last element which is

$$\tilde{a}_{n,n} = (1 + \sigma)\, a_{n,n}, \quad \sigma > 0.$$

In other words, matrix A is forced to be invertible by modifying one of its diagonal elements. Usually, this modification will cause slow convergence of the iterative process, since the condition number of $\widetilde{A}$ is always larger than the effective condition number of A. However, it has been shown that the modification does not affect the condition numbers and the convergence of DICCG at all [14]. Thus, even in the case of a singular A, we can apply the definition of P and E as given in Eq. (5) by simply replacing A by $\widetilde{A}$. The resulting deflation matrix is denoted by $\widetilde{P}$, i.e.,

$$\widetilde{P} := I - \widetilde{A}Z\widetilde{E}^{-1}Z^T, \quad \widetilde{E} := Z^T\widetilde{A}Z. \tag{10}$$

Since it is allowed to replace A by $\widetilde{A}$ in the DICCG method, the band-Cholesky decomposition exists. The resulting DICCG, which is based on $\widetilde{A}x = b$ and where $Ey_2 = y_1$ is solved directly, will be denoted by DICCG1.

Considering the floating point operations, we note that in the 2-D case constructing the Cholesky decomposition and the backward and forward substitutions require $\mathcal{O}(k^2)$ and $\mathcal{O}(k\sqrt{k})$ flops, respectively. In the 3-D case, constructing the Cholesky decomposition requires $\mathcal{O}(k^{7/3})$ flops, whereas the backward and forward substitutions takes $\mathcal{O}(k^{5/3})$ flops, see Appendix C and D for more details.

5.2. **Solving $Ey_2 = y_1$ iteratively.** If k or n is relatively large, it is not benificial to solve the coarse system $Ey_2 = y_1$ in a direct manner. In this case, it is more efficient to use an iterative method like ICCG. This is possible since E has more or less the same properties as A. Obviously, E is SPSD and has the same sparsity pattern as A. Moreover, E has an effective condition number which is smaller than the effective condition number of A, due to Theorem 1.

Theorem 1. *Let the eigenvalues of both A and E sorted increasingly, i.e.,* $0 = \mu_1(E) \leq \mu_2(E) \leq \ldots \leq \mu_k(E)$ *and* $0 = \lambda_1(A) \leq \lambda_2(A) \leq \ldots \leq \lambda_n(A)$. *Let Z be as defined in Section 3, where Z is scaled with $\sqrt{n/k}$ such that it satisfies $Z^TZ = I$. Then,*

$$\lambda_2(A) \leq \mu_2(E) \leq \ldots \leq \mu_k(E) \leq \lambda_n(A). \tag{11}$$

Proof. The theorem can be derived from the Courant-Fischer Minimax Theorem, see e.g. [4, Th. 4.2.11]. From this theorem, we obtain in particular

$$\lambda_2(A) = \min_{x^Tx=1,\ x\perp u_1(A)} x^TAx, \quad \lambda_n(A) = \max_{x^Tx=1} x^TAx, \tag{12}$$

where $u_1(A)$ is the eigenvector corresponding to $\lambda_1(A)$, see for details [4, Sect. 4.2].

Note first that the identities $u_1(A) = \mathbf{1}_n$ and $u_1(E) = \mathbf{1}_k$ hold due to Assumption 1 and Eq. (6). In addition, we have $x^TAx = (Zy)^TAZy = y^TZ^TAZy = y^TEy$, $(Zy)^T(Zy) = y^TZ^TZy = y^Ty$ and $(Zy)^T\mathbf{1}_n = y^TZ^T\mathbf{1}_n = y^T\mathbf{1}_k$, using the fact that $Z^T\mathbf{1}_n = \mathbf{1}_k$. Hence, this implies

$$\min_{(Zy)^T(Zy)=1,\ Zy\perp\mathbf{1}_n} (Zy)^TA(Zy) = \min_{y^Ty=1,\ y\perp\mathbf{1}_k} y^TEy. \tag{13}$$

Now, combining Eqs. (12) and (13) gives us

$$\lambda_2(A) = \min_{x^Tx=1,\ x\perp\mathbf{1}_n} x^TAx \leq \min_{y^Ty=1,\ y\perp\mathbf{1}_k} y^TEy = \mu_2(E),$$

which is the left inequality of (11). For the right inequality of (11) it follows in a similar way

$$\mu_k(E) = \max_{y^Ty=1} y^TEy \leq \max_{x^Tx=1} x^TAx = \lambda_n(A),$$

where we have applied $\max_{(Zy)^T(Zy)=1}(Zy)^TA(Zy) = \max_{y^Ty=1} y^TEy$.

□

Note that applying DICCG with the scaling of Z as given in Theorem 1 does not affect the method at all, since the column space of Z and as well the deflation matrix P do not change [6, Lemma 2.9]. It appears that DICCG depends on the column space of Z rather than on the exact deflation vectors.

Next, there is no need to force invertibility of E, since ICCG can deal with singular matrices. Thus, in contrast to DICCG1, we can solve the original singular linear system $Ax = b$. This variant of DICCG is denoted by DICCG2.

Note that we have to ensure the consistency of all coarse linear systems during the whole process of DICCG2. Theorem 2 guarantees this.

Theorem 2. *All coarse linear systems within DICCG2 are consistent if $Ax = b$ is consistent.*

Proof. Note first that for $r_0 = b - Ax_0$ we have

$$(Z^T r_0)^T \mathbf{1}_k = r_0^T Z \mathbf{1}_k = r_0^T \mathbf{1}_n = b^T \mathbf{1}_n - x_0^T A \mathbf{1}_n = \mathbf{0}_n - x_0^T \mathbf{0}_n = \mathbf{0}_n, \tag{14}$$

and, moreover, for all arbitrary vectors $q \in \mathbb{R}^n$ we obtain

$$(Z^T A q)^T \mathbf{1}_k = q^T A Z \mathbf{1}_k = q^T A \mathbf{1}_n = \mathbf{0}_n, \tag{15}$$

where we have applied Eq. (4) and Assumption 1.

The coarse linear systems $Ey_2 = y_1$ appears three times in Algorithm 2 (Lines 1, 3 and 11). We consider each of these cases.

(i) In the matrix-vector product Pr_0 we have to solve the coarse linear system

$$Ey_2 = Z^T r_0.$$

This system is consistent, since it is compatible due to Eqs. (6) and (14).

(ii) In PAp_j we have

$$Ey_2 = Z^T A p_0.$$

This system is consistent, since it is compatible due to Eqs. (6) and (15) by substituting $q := p_0$.

(iii) Using the same argument as in (ii), we conclude that $P^T \tilde{x}_{j+1}$ is also consistent, since $P^T \tilde{x}_{j+1} = \tilde{x}_{j+1} - ZE^{-1}Z^T A \tilde{x}_{j+1}$.

□

So it is convenient to solve the coarse systems iteratively, since ICCG guarantees solutions for the coarse systems. Each of these ICCG steps costs $\mathcal{O}(k)$ and the efficiency of this method depends on the number of required inner ICCG iterations.

Note that in DICCG2 we have an inner-outer iterative process with DICCG and ICCG, so we need two different termination criteria. The inner and outer tolerance ϵ are called ϵ_{out} and ϵ_{in}, respectively. We will take

$$\epsilon_{in} = \omega \ \epsilon_{out}, \quad \omega > 0. \tag{16}$$

For large $\omega \geq 1$, DICCG2 will not converge, since the operations with P are not computed sufficiently accurate, see also [6, Sect. 3]. However, for small $\omega < 1$, DICCG2 will not converge as well, since the termination criterion is too severe with respect to the machine precision. Therefore, ω should be chosen carefully to obtain an accurate and efficient method. From numerical experiments it appears that

$$\omega = 10^{-2} \tag{17}$$

is a proper choice. We refer to [10, 11] for more information about inner-outer iterative processes and their termination criteria.

We end this section with the remark that for large problems it can be advantageous to solve the inner iterations also with DICCG instead of ICCG. The inner iterations can even be solved by recursively application of DICCG. This is in analogy with multigrid, see also [1, Sect. 3]. It is however left for future research.

6. Theoretical Comparison of DICCG1 and DICCG2

In the previous section we have introduced the two deflation variants DICCG1 and DICCG2. In this section we will make a theoretical comparison between these two methods. We will show that DICCG1 and DICCG2 perform the same theoretically, since it can be proven that the system $\widetilde{P}\widetilde{A}$ in DICCG1 is equal to PA in DICCG2.

The proof is given in Subsection 6.3, after introducing some notations and giving some auxiliary results in the next subsections.

6.1. **Notations.** We adopt the notation from [14], where we denote P with k deflation vectors by P_k, i.e.,

$$P_k := I - AZ_kE_k^{-1}Z_k^T, \quad P_{k-1} := I - AZ_{k-1}E_{k-1}^{-1}Z_{k-1}^T,$$

where

$$E_k := Z_k^TAZ_k, \quad E_{k-1} = Z_{k-1}^TAZ_{k-1},$$

and

$$Z_{k-1} = [z_1 \; z_2 \; \cdots z_{k-1}], \quad Z_k = [Z_{k-1} \; \mathbf{1}_n].$$

Note that replacing $Z_k = [Z_{k-1} \; \mathbf{1}_n]$ by $Z = [Z_{k-1} \; z_k]$ would lead to exactly the same deflation matrix P_k, since the column space of Z_k and Z are equal [6, Lemma 2.9].

Furthermore, we can observe that E_{k-1} is invertible whereas E_k is singular. Therefore, E_k^{-1} is the pseudo-inverse while E_{k-1}^{-1} is the real inverse. Recall that in DICCG2 the computations considering E_k^{-1} are done using the ICCG method. Note that $E_{k-1}^{-1}Z_{k-1}^T$ can not be computed using CG, since it is not compatible, in contrast to $E_{k-1}^{-1}Z_{k-1}^TA$ which is compatible (cf. Theorem 2). Therefore, P_k and P_{k-1} can not be compared, while a comparison between P_kA and $P_{k-1}A$ can be made without problems.

6.2. **Auxiliary Results.** In [14, Lemma 5.5], we have proven the following lemma.

Lemma 1. *Let $\widetilde{A}$ and Z_k be defined as above. Then there exists a matrix $Y \in \mathbb{R}^{n\times n-k}$ such that*

- *matrix $X := [Y \; Z_k]$ is invertible;*
- *identity $Z_k^T\widetilde{A}Y = \mathbf{0}_{k,n-k}$ holds.*

Note that $\widetilde{A}$ can not be replaced by A, since in the proof of the lemma we have used the fact that the $\widetilde{A}$-inner product is an inner product, while the A-inner product is not an inner product: $(z, z)_A$ can be zero for $z \neq \mathbf{0}_n$.

From Lemma 1 we can find the following corollary.

Corollary 1. *Let Y be as given as in Lemma 1. Then,*

(i) *the last row of Y is $\mathbf{0}_n^T$;*
(ii) *Y satisfies $Z_k^TAY = \mathbf{0}_{k,n-k}$.*

Proof. (i) Note that from [14, Cor. 2] we have

$$\widetilde{A}\mathbf{1}_n = \sigma a_{n,n}\mathbf{e}_n^{(n)}, \tag{18}$$

resulting in a last row of $Z_k^T\widetilde{A}$ which is equal to $\sigma a_{n,n}\mathbf{e}_n^{(n)}$. Hence, $Z_k^T\widetilde{A}Y = \mathbf{0}_{k,n-k}$ can only be satisfied if all elements of the last row of Y are zero, i.e., the last row of Y is $\mathbf{0}_n^T$.

(ii) We have (cf. [14, Th. 1])

$$A = \widetilde{A} - \tau cc^T, \quad c = \mathbf{e}_n^{(n)}, \quad \tau = \sigma \cdot a_{n,n}.$$

Moreover, note that

$$Z_k^Tcc^T = cc^T.$$

Then,

$$Z_k^T cc^T Y = cc^T Y = \mathbf{0}_{k,n-k},$$

since the last row of Y is $\mathbf{0}_n^T$. Finally, this yields

$$Z_k^T AY = Z_k^T(\widetilde{A} - \tau cc^T)Y = Z_k^T \widetilde{A} Y - \tau Z_k^T cc^T Y = \mathbf{0}_{k,n-k}^T - \mathbf{0}_{k,n-k}^T = \mathbf{0}_{k,n-k}^T.$$

□

6.3. **Identical Systems in DICCG1 and DICCG2.** First we will prove that the deflated system of A with k deflation vectors is the same as the deflated system of A with $k-1$ deflation vectors, i.e., $P_k A = P_{k-1}A$. This result is somewhat suprising, since $\mathrm{Col}(Z_{k-1}) \subset \mathrm{Col}(Z_k)$. Therefore, a more favorable spectrum of $P_k A$ will be expected compared to $P_{k-1}A$. However, this is not the case; the spectra of both systems $P_k A$ and $P_{k-1}A$ are equal. More strongly, the identity $P_k A = P_{k-1}A$ holds, see Theorem 3.

Theorem 3. *The following identity holds:*

$$P_k A = P_{k-1} A. \tag{19}$$

Proof. Let $X = [Z_k \ Y] \in \mathbb{R}^{n,n}$ where Z_k and Y be matrices as defined in Lemma 1. We prove that

$$(P_k - P_{k-1})AX = \mathbf{0}_{n,n} \tag{20}$$

holds. Then after right-multiplying both sides with X^{-1}, Eq. (19) follows immediately.

We know $Z_k^T AY = \mathbf{0}_{k,n-k}$ from Corollary 1. In particular we have $Z_{k-1}^T AY = \mathbf{0}_{k-1,n-k}$, which gives immediately

$$\begin{aligned} (P_k - P_{k-1})AY &= AZ_{k-1}E_{k-1}^{-1}Z_{k-1}^T AY - AZ_k E_k^{-1} Z_k^T AY \\ &= AZ_{k-1}E_{k-1}^{-1}\mathbf{0}_{k-1,n-k} - AZ_k E_k^{-1}\mathbf{0}_{k,n-k} \\ &= \mathbf{0}_{n,n-k} \end{aligned}$$

Hence,

$$(P_k - P_{k-1})AY = \mathbf{0}_{n,n-k}. \tag{21}$$

Next, it appears that $AZ_k E_k^{-1} Z_k^T AZ_k = AZ_k$, since $E_k^{-1}Z_k^T AZ_k = I$. Moreover, note also that

$$\begin{aligned} AZ_{k-1}E_{k-1}^{-1}Z_{k-1}^T AZ_k &= AZ_{k-1}E_{k-1}^{-1}Z_{k-1}^T A \cdot [Z_{k-1} \quad \mathbf{1}_n] \\ &= [AZ_{k-1}E_{k-1}^{-1}Z_{k-1}^T AZ_{k-1} \quad AZ_{k-1}E_{k-1}^{-1}Z_{k-1}^T A\mathbf{1}_n] \\ &= [AZ_{k-1} \quad AZ_{k-1}E_{k-1}^{-1}Z_{k-1}^T \mathbf{0}_n] \\ &= [AZ_{k-1} \quad \mathbf{0}_n] \\ &= AZ_k. \end{aligned}$$

Combining the latter expressions we obtain

$$\begin{aligned} (P_k - P_{k-1})AZ_k &= AZ_{k-1}E_{k-1}^{-1}Z_{k-1}^T AZ_k - AZ_k E_k^{-1} Z_k^T AZ_k \\ &= AZ_k - AZ_k \\ &= \mathbf{0}_{n,k}. \end{aligned} \tag{22}$$

Finally, Eq. (20) follows immediately from Eqs. (21) and (22), i.e.,

$$\begin{aligned} (P_k - P_{k-1})AX &= (P_k - P_{k-1})A[Z_k \ Y] \\ &= [\mathbf{0}_{n,k} \ \mathbf{0}_{n,n-k}] \\ &= \mathbf{0}_{n,n}. \end{aligned}$$

□

As a consequence, Corollary 2 follows.

Corollary 2. $P_k A = \widetilde{P}_k \widetilde{A}$.

Proof. Theorem 3 of [14] gives

$$\widetilde{P}_k \widetilde{A} = P_{k-1} A. \tag{23}$$

Combining Eq. (23) with Theorem 3 leads to the corollary. □

Hence, we obtain that DICCG1 and DICCG2 give exactly the same convergence results in exact arithmetics, since their preconditioned deflated systems $M^{-1}P_kA$ and $M^{-1}\widetilde{P}_k\widetilde{A}$ are completely equal in these methods. In the next sections, we will investigate whether DICCG1 and DICCG2 are also equal in practice.

We end this section with Corollary 3, which is a generalization of Theorem 2.10 of [6].

Corollary 3. *Let A and M be matrices as defined above. Let $V \in \mathbb{R}^{n\times s}$ and $W \in \mathbb{R}^{n\times t}$ with rank $V = s$ and rank $W = t$. Let $Q_V := I - AV(V^TAV)^{-1}V^T$ and $Q_W := I - AW(W^TAW)^{-1}W^T$ where $(V^TAV)^{-1}$ and $(W^TAW)^{-1}$ denote the pseudo-inverses of V^TAV and W^TAW, respectively. If Col $(V) \subseteq$ Col (W), then*

$$\begin{array}{rcl} \lambda_n(M^{-1}Q_VA) & \geq & \lambda_n(M^{-1}Q_WA); \\ \lambda_{s+1}(M^{-1}Q_VA) & \leq & \lambda_{t+1}(M^{-1}Q_WA). \end{array} \tag{24}$$

Moreover,

$$\kappa_{eff}(M^{-1}Q_VA) \geq \kappa_{eff}(M^{-1}Q_WA), \tag{25}$$

where κ_{eff} denotes the effective condition number.

Proof. The corollary follows immediately from Theorem 2.12 of [6] and Corollary 2. □

Corollary 3 states that the effective condition number of the deflated preconditioned system corresponding to the singular matrix A decreases if we increase the number of deflation vectors. This means that theoretically the more deflation vectors are taken, the faster the convergence of the deflation method with respect to the number of iterations, although more work is needed for solving the coarse systems.

7. 2-D Numerical Experiments

In this section we present the results of some 2-D numerical experiments done with FORTRAN. The computations are performed on a serial Pentium 4 (2.80 GHz) computer with a memory capacity of 1GB. Moreover, the code is compiled with FORTRAN g77 on LINUX.

7.1. Test Problems.

We apply the 2-D variant of the problem setting as given in Section 2. These experiments will illustrate the theoretical results obtained in the previous sections. We consider three test problems:

- (2D-TP1) no bubbles;
- (2D-TP2) one bubble;
- (2D-TP3) nine bubbles,

in the unit domain Ω filled with water. Moreover, we vary the contrast $\theta := \rho_1/\rho_0$ between the phases: $\theta = 10^{-3}, 10^{-6}, 10^{-8}$. Finally, various number of grid sizes N_x and N_y and number of deflation vectors will be used.

We apply ICCG and DICCG1$-k$ to solve the linear system, where DICCG1$-k$ denotes DICCG1 with k deflation vectors. Note that for ICCG we consider the singular system $Ax = b$ while we apply the invertible system $\widetilde{A}x = b$ for DICCG1$-k$, as mentioned before. In both methods, a random starting vector will be used and we choose the relative termination criterion with tolerance $\epsilon = 10^{-8}$. At the end of each test problem we compute the relative exact residuals, i.e.,

$$\phi := \frac{||b - Ax||_2}{||b - Ax_0||_2},$$

as a measure of the accuracy of the solutions.

7.2. **Results with $N_x = N_y = 100$.** We present the results of the three test problems with grid sizes $N_x = N_y = 100$.

7.2.1. *Table of the Results.* The results considering the CPU time (in seconds), the number of required iterations and the relative exact residuals for $\theta = 10^{-3}$ are given in Table 1.

Test Problem	Method	# Iter.	CPU (sec)	ϕ ($\times 10^{-9}$)
2D-TP1 (No Bubbles)	ICCG	109	0.12	2.3
	DICCG1-5^2	49	0.08	2.5
	DICCG1-10^2	32	0.06	2.0
	DICCG1-20^2	21	0.05	1.7
	DICCG1-25^2	19	0.06	0.8
	DICCG1-50^2	12	0.11	0.5
2D-TP2 (One Bubble)	ICCG	128	0.14	0.3
	DICCG1-5^2	51	0.08	0.4
	DICCG1-10^2	34	0.07	0.3
	DICCG1-20^2	22	0.06	0.6
	DICCG1-25^2	20	0.06	0.4
	DICCG1-50^2	13	0.11	0.1
2D-TP3 (Nine Bubbles)	ICCG	247	0.26	0.2
	DICCG1-5^2	70	0.11	0.4
	DICCG1-10^2	44	0.08	0.4
	DICCG1-20^2	27	0.07	0.4
	DICCG1-25^2	23	0.06	0.5
	DICCG1-50^2	14	0.11	0.1

TABLE 1. Convergence Results of ICCG and DICCG$-k$ for all test problems with $N_x = N_y = 100$.

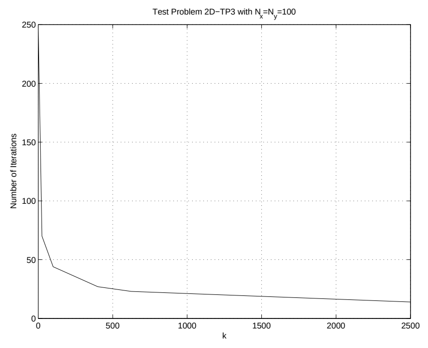
Considering Table 1 we can make the following observations.

- In all test problems it is obvious that DICCG1$-k$ reduces the number of iterations. The larger k the smaller the required number of iterations of DICCG1$-k$ to converge.
- Besides the number of iterations, the CPU times decreases as well for larger k, except for the case DICCG1-50^2. The optimal choices are $k = 20^2, 25^2$ in the most cases.
- The more bubbles the worse the performance of ICCG and the larger the differences between ICCG and DICCG1$-k$ becomes in favor of DICCG$-k$.
- The relative exact residuals ϕ are comparable with the relative update residuals, since they are both of the same order.

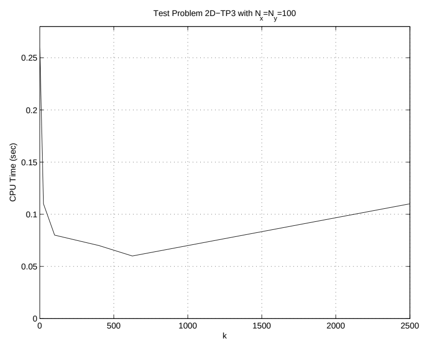
7.2.2. *Visualization of the Results.* To visualize the results in Table 1, we present those of Test Problem 2D-TP3 in Figure 3.

Figure 3 can be interpreted as follows:

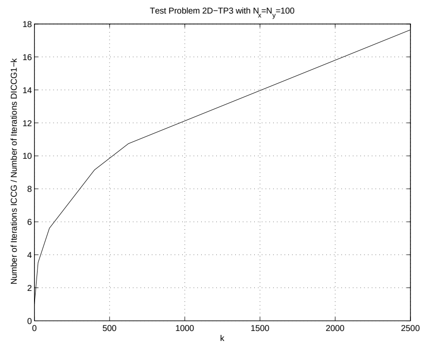
- Subfigure 3(a): When k is increased, the number of iterations of DICCG1$-k$ decreases significantly in the beginning of the curve. For large k, the benefit is smaller.
- Subfigure 3(b): The required CPU time for DICCG1$-k$ decreases until $k = 25^3$. Thereafter, the CPU time increases and the DICCG1$-k$ is less efficient.



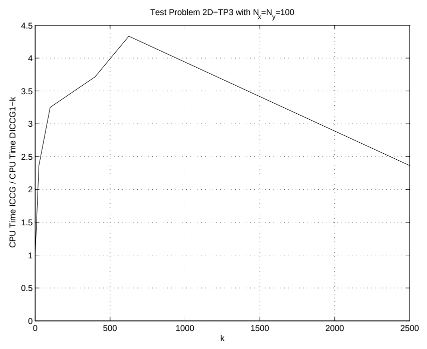
(a) The number of iterations versus the number of deflation vectors .



(b) The CPU time versus the number of deflation vectors.



(c) The ratio of the number of ICCG and DICCG1$-k$ iterations versus the number of deflation vectors versus .



(d) The ratio of the CPU time required in ICCG and DICCG1$-k$ versus the number of deflation vectors.

FIGURE 3. Visualization of the results of Test Problem 2D-TP3.

- Subfigure 3(c): The benefit factor considering the number of iterations is depicted for each k. Obviously, the larger k the larger the profit of DICCG1$-k$. For example, in the case of $k = 50^2$, DICCG1$-k$ requires almost 18 times less iterations compared to ICCG, which is a relatively huge gain.
- Subfigure 3(d): The benefit factor considering the CPU time is depicted for each k. The maximum benefit factor is around 4.3, which means that in that case DICCG1$-k$ is 4.3 times faster than ICCG.

7.2.3. *Update Residual Plots.* To see the development of the update residuals during the iterates of DICCG1$-k$ for all k, the corresponding plots of ICCG and DICCG1$-k$ can be found in Figure 4.

From Figure 4, we see that the residuals of ICCG show a lot of wiggles, possibly caused by the relatively high number of bubbles in the domain. The wiggles disappeared completely in DICCG1$-k$. Apparently, the piecewise constant deflation vectors represent the eigenvectors corresponding to the small eigenvalues of A very well.

7.2.4. *Solution Plots.* We have seen above that ICCG and DICCG1$-k$ have comparable small relative exact residuals ϕ at the end of the iterates of each test case. However, it is not always guaranteed that the relative exact errors are sufficiently small if ϕ is small. Hence, we consider the solution plots of both ICCG and DICCG1$-k$ for Test Problem 2D-TP3 to check those exact errors, see Figure 5.

From Figure 5 we see obviously the nine bubbles, since in these regions the solution is constant. Moreover, a vertical shift can be observed by comparing the

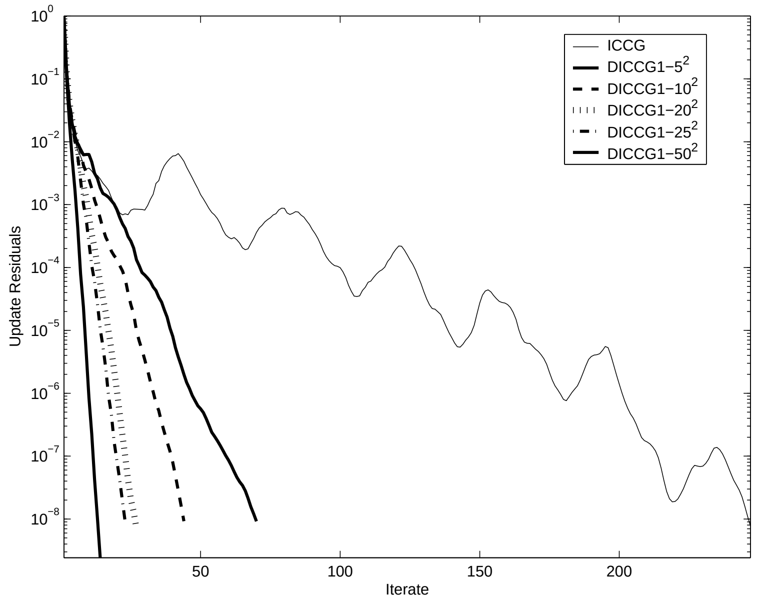


FIGURE 4. Update Residual plots of ICCG and DICCG1$-k$ for Test Problem 2D-TP3 with $k = 5^2, 10^2, 20^2, 25^2, 50^2$.

solutions of ICCG and DICCG1. This has been caused by the fact that we have solved a singular system in ICCG, whereas an invertible system has been solved in DICCG. Therefore, in the case of ICCG, x is not unique, since $x + c \cdot \mathbf{1}_n$ with $c \in \mathbb{R}$ is also a solution. The vertical shift between Subfigures 5(a) and 5(b) is approximately $c = 0.6$.

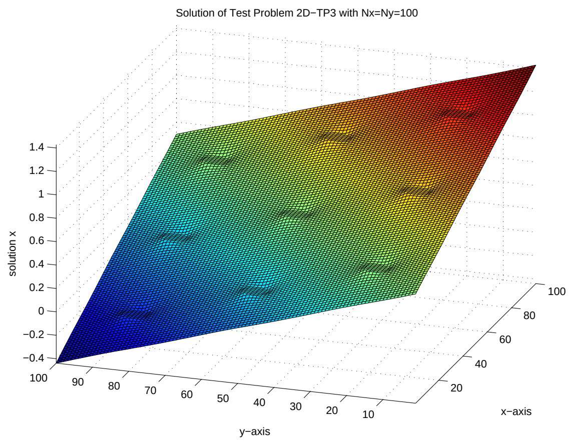
7.3. **Results for Test Problem 2D-TP3 with varying grid sizes.** The results for Test Problem 2D-TP3 with varying grid sizes are given in Table 2.

	$N_x = N_y = 100$		$N_x = N_y = 250$		$N_x = N_y = 500$	
Method	# Iter.	CPU	# Iter.	CPU	# Iter.	CPU
ICCG	247	0.26	466	3.56	1027	34.4
DICCG1-5^2	70	0.11	143	1.39	237	9.32
DICCG1-10^2	44	0.08	88	0.89	150	6.09
DICCG1-25^2	23	0.06	45	0.52	82	3.55
DICCG1-50^2	14	0.11	26	0.41	43	2.14
DICCG1-100^2	–	–	–	–	27	3.12

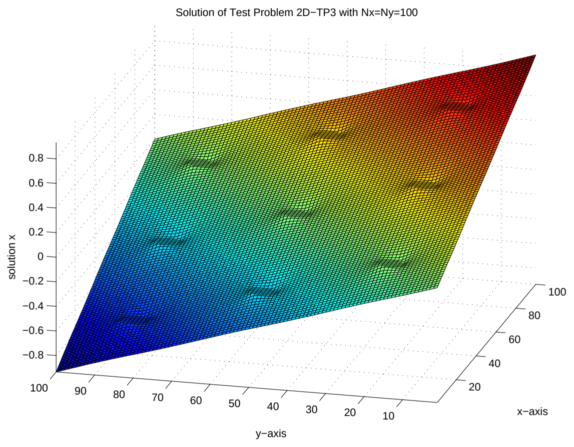
TABLE 2. Results of 2D-TP3 (nine bubbles) for varying grid sizes N_x and N_y.

From the table, one observes immediately that for larger grid sizes, the differences in performance between ICCG and DICCG1$-k$ becomes significantly large. For instance, in the case of $N_x = N_y = 500$, ICCG does not converge within 1000 iterations and 30 seconds, while DICCG1-50^2 finds the solution within 43 iterations in just 2.14 seconds.

The residual plots corresponding to these results are omitted here, but they have the same behavior as in Subsection 7.2. Moreover, observe that if we take k more or less proportional to the grid sizes then the number of iterations remains approximately the same. Equivalently, if we compare DICCG1$-k$ for varying grid



(a) Solution x using ICCG.



(b) Solution x using DICCG1-10^2.

FIGURE 5. Solution plot of ICCG and DICCG1$-k$ for Test Problem 2D-TP3.

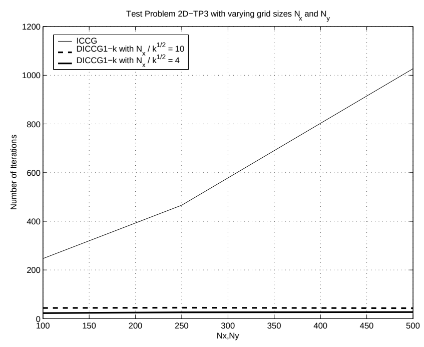
sizes satisfying

$$\frac{N_x}{k_x} = \psi, \quad \psi \in \mathbb{N},$$

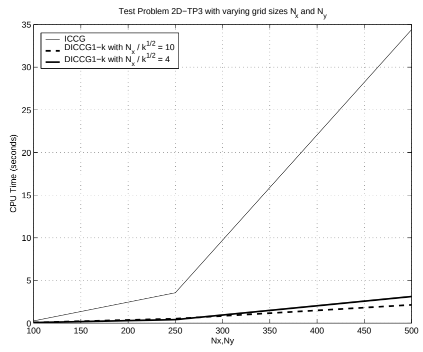
where ψ is fixed, then the number of iterations are approximately equal for all DICCG$-k$. This can also be observed in Figure 6, which is a visualization of Table 2 by taking $\psi = 10$ and $\psi = 4$.

In Subfigure 6(a) we see that the number of ICCG iterates grows, while the number of iterations for DICCG$-k$ for both $\psi = 4$ and $\psi = 10$ remains constant. With respect to the CPU time, we conclude from Subfigure 6(b) that this grows more or less exponentially for ICCG, whereas the CPU time grows approximately linearly for DICCG$-k$. Finally, note that although DICCG$-k$ for $\psi = 4$ requires less iterations compared to DICCG$-k$ for $\psi = 10$, it is not more efficient considering the CPU time.

7.4. **Results for Test Problem 2D-TP3 with varying contrasts.** After varying the grid sizes, the grid sizes are fixed to $N_x = N_y = 100$ and now we vary the contrast θ between the phases. In practice this means that we consider other phases instead of only the combination of water and air. The results considering the CPU time and the number of iterations of Test Problem 2D-TP3 are given in Table 3.



(a) The number of iterations versus grid size per direction.



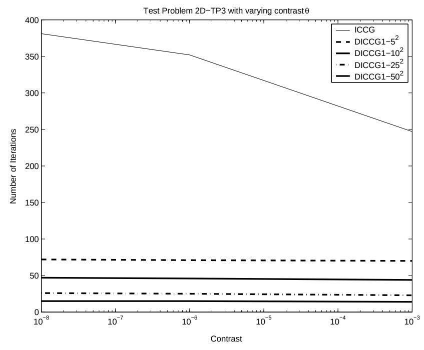
(b) The CPU time versus grid size per direction.

FIGURE 6. Visualization of the results of Test Problem 2D-TP3 with varying grid sizes. ICCG and DICCG$-k$ with both $\psi = 10$ and $\psi = 4$ are used.

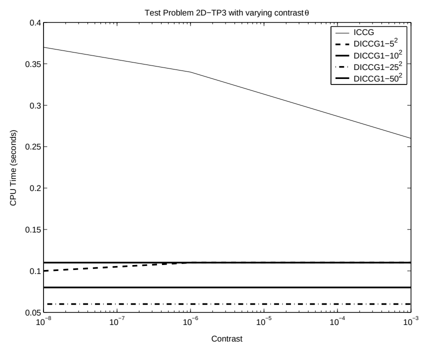
	$\theta = 10^{-3}$		$\theta = 10^{-6}$		$\theta = 10^{-8}$	
Method	# Iter.	CPU	# Iter.	CPU	# Iter.	CPU
ICCG	247	0.26	352	0.34	381	0.37
DICCG1-5^2	70	0.11	71	0.10	72	0.10
DICCG1-10^2	44	0.08	46	0.08	47	0.08
DICCG1-25^2	23	0.06	25	0.06	26	0.06
DICCG1-50^2	14	0.11	15	0.11	15	0.11

TABLE 3. Results of 2D-TP3 (nine bubbles) for varying contrast θ.

From the table we see that DICCG1$-k$ depends hardly on the contrast θ, while obviously ICCG becomes worse by decreasing θ. This observation can also be seen in Figure 7. DICCG$-k$ is insensitive for the contrast θ considering both the number of iterations and the CPU time. This is an important advantage in favor of DICCG1$-k$.



(a) The number of iterations versus the contrast between the phases.



(b) The CPU time versus the contrast between the phases.

FIGURE 7. Visualization of the results of Test Problem 2D-TP3 with varying contrast θ. Comparison of ICCG and DICCG$-k$ are given for all k.

Again, the residual plots corresponding to these results are omitted, but they have the same behavior as in Subsection 7.2.

8. 3-D Numerical Experiments

Similar to the previous section, we present the results of some 3-D numerical experiments in this section. In fact, this is a generalization of the previous section to the 3-D case.

8.1. **Test Problems.** We apply the 3-D variant of the problem setting as given in Section 2. The following test problems will be considered:

- (3D-TP1) no bubbles;
- (3D-TP2) one bubble;
- (3D-TP3) eight bubbles;
- (3D-TP4) 27 bubbles,

in Ω. Moreover, we will vary again θ, n and k. We apply both ICCG and DICCG1$-k$ to solve the resulting linear systems. Finally, in the last subsection we will compare DICCG1$-k$ and DICCG2$-k$.

8.2. **Results with $N_x = N_y = N_z = 100$.** We present the results of the test problems with grid sizes $N_x = N_y = N_z = 100$.

8.2.1. *Table of the Results.* The results considering the CPU time, the number of iterations and the relative exact residuals ϕ of the test problems with $\theta = 10^{-3}$ are given in Table 4.

Test Problem	Method	# Iter.	CPU (sec)	ϕ ($\times 10^{-9}$)
	ICCG	170	25.2	2.9
3D-TP1	DICCG1-2^3	109	20.2	2.9
(No Bubbles)	DICCG1-5^3	56	11.3	2.3
	DICCG1-10^3	35	8.0	1.8
	DICCG1-20^3	22	26.5	1.1
	ICCG	211	31.1	1.4
3D-TP2	DICCG1-2^3	206	37.5	1.3
(One Bubble)	DICCG1-5^3	58	11.5	1.5
	DICCG1-10^3	36	8.5	1.2
	DICCG1-20^3	25	27.6	1.5
	ICCG	291	43.0	1.1
3D-TP3	DICCG1-2^3	160	29.1	1.1
(8 Bubbles)	DICCG1-5^3	72	14.2	1.2
	DICCG1-10^3	36	8.2	0.7
	DICCG1-20^3	22	27.2	0.9
	ICCG	310	46.0	1.3
3D-TP3	DICCG1-2^3	275	50.4	1.3
(27 Bubbles)	DICCG1-5^3	97	19.0	1.2
	DICCG1-10^3	60	13.0	1.2
	DICCG1-20^3	31	29.3	1.2

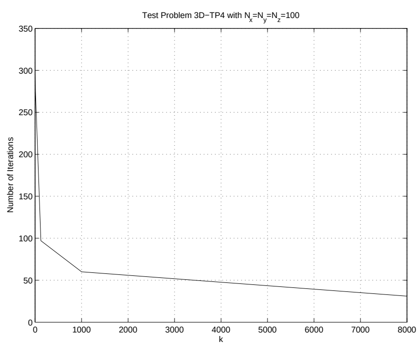
Table 4. Convergence Results of ICCG and DICCG$-k$ for all test problems with $N_x = N_y = N_z = 100$.

Considering Table 4 we can note the following.

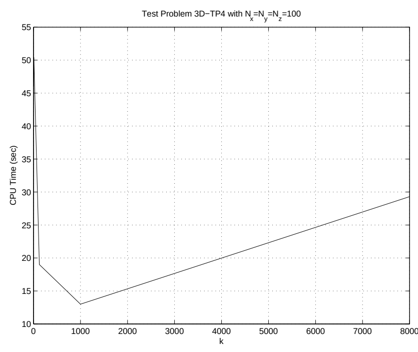
- DICCG1$-k$ requires always less iterations compared to ICCG. It can be observed that for larger k, DICCG1$-k$ requires less iterations.

- Considering the CPU time we have found that the optimal choice is $k = 10^3$, i.e., in all test cases DICCG1-10^3 converges the fastest. The benefit of CPU time is relatively large compared to ICCG.
- The relative exact residuals of both ICCG and DICCG1$-k$ are comparable in all test cases and they are of the same order as the relative update residuals.
- The more bubbles in the test problem, the more iterations and therefore the more CPU time both ICCG and DICCG$-k$ require to converge.
- For large k, DICCG1$-k$ shows difficulties with respect to the computations with E. Therefore, for $k > 10^3$ DICCG1$-k$ converges in a low number of iterations, but it requires a lot of CPU time in each iterate.

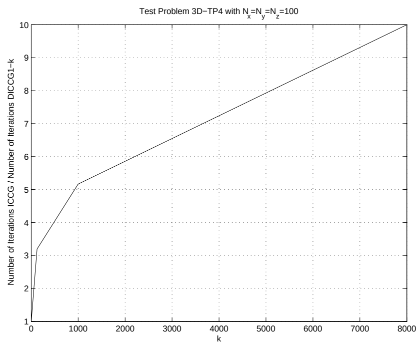
8.2.2. *Visualization of the Results.* To visualize the results in Table 4, we present those of Test Problem 3D-TP4 in Figure 8.



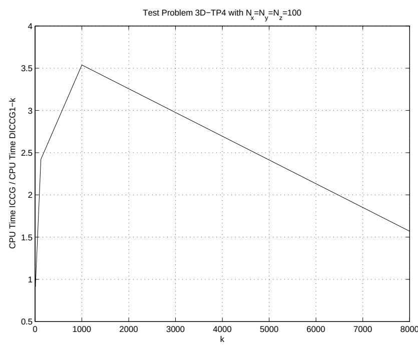
(a) The number of iterations versus the number of deflation vectors.



(b) The CPU time versus the number of deflation vectors.



(c) The ratio of the number of ICCG and DICCG1$-k$ iterations versus the number of deflation vectors.



(d) The ratio of the CPU time required in ICCG and DICCG1$-k$ versus the number of deflation vectors.

FIGURE 8. Visualization of the results of Test Problem 3D-TP4.

Figure 8 can be interpreted as follows.

- Subfigure 8(a): Similar to the 2D case, if k is increased, the number of iterations of DICCG1$-k$ decreases significantly in the beginning of the curve. For large k, the benefit is smaller.
- Subfigure 8(b): The required CPU time for DICCG1$-k$ decreases until $k = 10^3$. Thereafter, the CPU time increases and DICCG1$-k$ is less efficient.
- Subfigure 8(c): The benefit factor considering the number of iterations is depicted for each k. Obviously, the larger k the larger the profit of DICCG1$-k$.

For example, in the case of $k = 20^3$, DICCG1$-k$ requires almost 10 times less iterations compared to ICCG.

- Subfigure 8(d): The benefit factor considering the CPU time is depicted for each k. For $k = 10^3$, the maximum benefit factor has been achieved. In this case, DICCG1-10^3 is 3.5 times faster than ICCG.

8.2.3. *Update Residual Plots.* For all cases of Test Problem 4, the plots of the residuals of both ICCG and DICCG can be found in Figure 9.

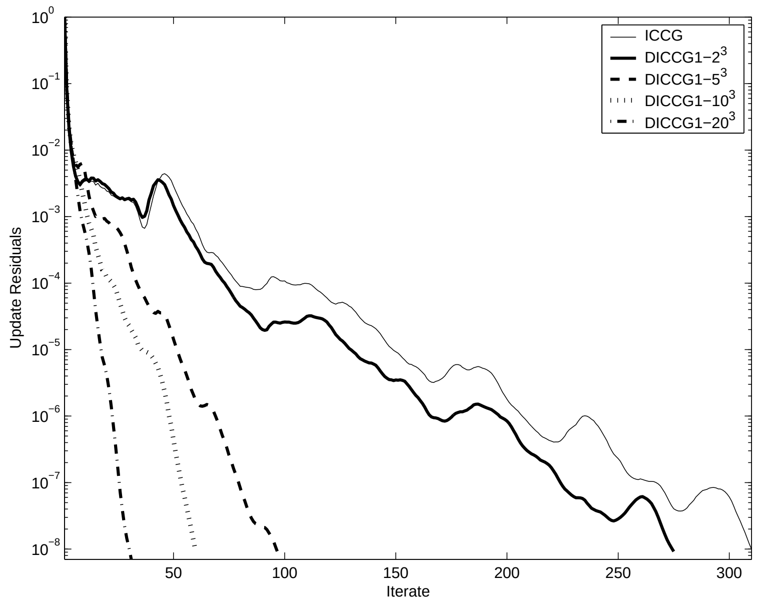


FIGURE 9. Update residual plots of ICCG and DICCG1$-k$ for Test Problem 3D-TP4 with $k = 2^3, 5^3, 10^3, 20^3$.

Note that the behavior of the residuals of ICCG are somewhat irregular due to the presence of the bubbles. For DICCG1$-k$ we can conclude that the larger k the more linear the residual plot is. Apparently, the larger k the better the deflation vectors resemble the eigenvectors corresponding to the smallest eigenvalues of the linear system.

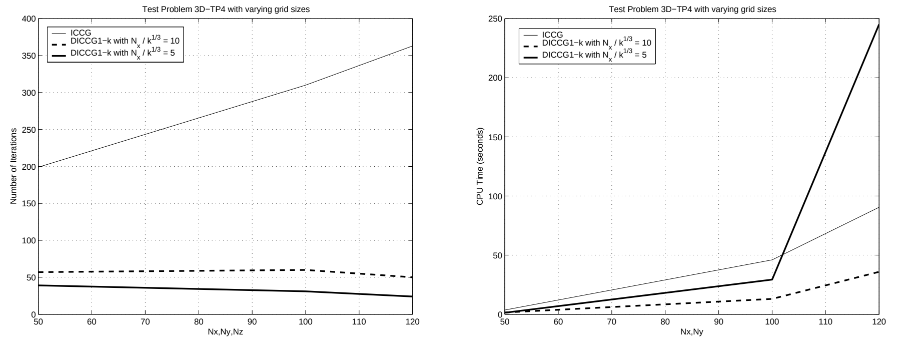
8.3. **Results for Test Problem 3D-TP4 with varying grid sizes.** The results for Test Problem 3D-TP4 with varying grid sizes are given in Table 5.

	$n = 50^3$		$n = 100^3$		$n = 120^3$	
Method	# It.	CPU	# It.	CPU	# It.	CPU
ICCG	199	3.6	310	46.0	363	90.5
DICCG1-2^3	179	4.1	275	50.4	291	90.2
DICCG1-5^3	57	1.5	97	19.0	111	35.9
DICCG1-10^3	39	1.3	60	13.0	68	24.1
DICCG1-12^3	–	–	–	–	50	18.9
DICCG1-20^3	–	–	31	29.3	33	35.9
DICCG1-24^3	–	–	–	–	24	245.2

TABLE 5. Results of 3D-TP4 (27 bubbles) for varying grid sizes N_x, N_y and N_z.

From the table, one observes immediately that for larger grid sizes, the differences in performance between ICCG and DICCG1$-k$ becomes significantly larger. For instance, in the case of $N_x = N_y = N_z = 120$, ICCG does not converge within 350 iterations and 90 seconds, while DICCG1-10^2 finds the solution within 68 iterations in just 35.9 seconds.

Similar to the previous section we use $N_x/k_x = \psi$. In the 2-D case, we have concluded that the number of iterations are approximately equal for all DICCG$-k$ with fixed ψ. In the 3D case, this observation holds to some extent. This can also be observed in Figure 10, which is a visualization of Table 5 by taking $\psi = 10$ and $\psi = 5$.



(a) The number of iterations versus the grid size per direction.

(b) The CPU time versus the grid size per direction.

FIGURE 10. Visualization of the results of Test Problem 3D-TP4 with varying grid sizes. DICCG$-k$ with both $\psi = 10$ and $\psi = 5$ are given.

In Subfigure 10(a) we see that the number of ICCG iterates grows, while the number of iterations for DICCG$-k$ for both $\psi = 4$ and $\psi = 10$ remains more or less constant. With respect to the CPU time, we have seen in the previous section that this grows more or less exponentially for ICCG, whereas the CPU time increases approximately linear for DICCG$-k$. In the 3D case, DICCG1$-k$ grows also exponentially, due to the expensive computations with E. In Subsection 8.5, we will remedy this by using DICCG2 instead of DICCG1.

Finally, note that although DICCG$-k$ for $\psi = 5$ requires less iterations compared to DICCG$-k$ for $\psi = 10$, due to the increased costs per iteration it is not more efficient considering the CPU time.

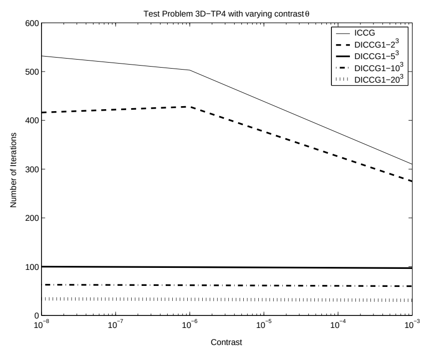
8.4. **Results for Test Problem 3D-TP4 with varying contrasts.** Now, the grid sizes are fixed to $n = 100^3$ and we vary the contrast θ between the phases. The results considering the CPU time and the number of iterations of Test Problem 3D-TP4 are given in Table 6.

From the table we see that DICCG1$-k$ with $k > 2^3$ depends hardly on the contrast θ, while obviously ICCG becomes worse by enlarging θ. This observation can also be seen in Figure 11. DICCG$-k$ is insensitive for the contrast θ by considering both the number of iterations and the CPU time.

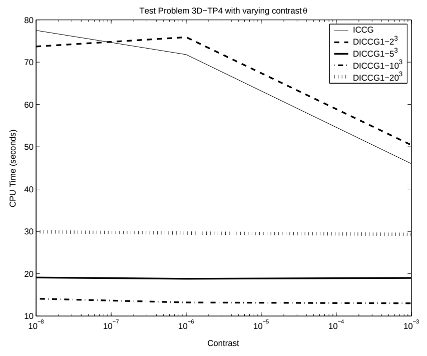
8.5. **Comparison of DICCG1 and DICCG2.** In the previous sections we have seen that DICCG1$-k$ is very efficient as long as $k < 20^3$. From $k = 20^3$ DICCG1$-k$ requires too many computational costs per iterate, although only a relatively low number of iterations is needed. The bottleneck for $k > 10^3$ is the expensive construction of the banded Cholesky decomposition of E. We have seen that the bandwidth of E is k_x^2 and for the band Cholesky matrix all $k_x^2 + 1$ bands are filled resulting in

	$\theta = 10^{-3}$		$\theta = 10^{-6}$		$\theta = 10^{-8}$	
Method	# Iter.	CPU	# Iter.	CPU	# Iter.	CPU
ICCG	310	46.0	503	71.8	532	77.5
DICCG1-2^3	275	50.4	428	75.9	416	73.7
DICCG1-5^3	97	19.0	99	18.8	100	19.1
DICCG1-10^3	60	13.0	62	13.2	63	14.1
DICCG1-20^3	31	29.3	33	29.6	34	29.9

TABLE 6. Results of 3D-TP4 (27 bubbles) for varying contrast θ.



(a) The number of iterations versus the contrast between the phases.



(b) The CPU time versus the contrast between the phases.

FIGURE 11. Visualization of the results of Test Problem 3D-TP4 with varying contrast θ. Comparison of ICCG and DICCG$-k$ is made for all k.

an inefficient method for large k. Direct computations considering E can be avoided by using DICCG2$-k$, as mentioned in Section 5. In this subsection, a comparison between DICCG1$-k$ and DICCG2$-k$ will be made.

Since we have applied $\epsilon_{out} = 10^{-8}$ in all test cases, we need the stopping tolerance $\epsilon_{in} = 10^{-10}$ for the inner iterations, also mentioned in Section 5.

Some results of Test Problem 3D-TP4 for varying grid sizes can be found in Table 7. Similar results have been found for the other test problems. It appears that the number of iterations of both DICCG1 and DICCG2 are equal in almost all test cases. Therefore, the common number of iterations are given for both DICCG1 and DICCG2, whereas rarely an asterisk (*) is put behind the number when there are slight differences. Moreover, since it appears that the relative exact residuals are comparable for the two methods, those are omitted in the table.

From Table 7, we conclude that for sufficiently small problems or for a small number of deflation vectors, DICCG1$-k$ is the fastest method, but for large problems or for problems with relatively large k, DICCG2$-k$ is clearly more efficient. The differences between the two variants of DICCG becomes significantly large from $k = 20^3$ in all test cases.

The visualization of the last test case with $n = 120^3$ can be found in Figure 9. From this figure we observe immediately that in all cases DICCG1$-k$ achieves its optimum if $k = 10^3$. For $k > 10^3$ DICCG1$-k$ is not efficient anymore. On the other hand, it can be noticed that in all cases the optimum of DICCG2$-k$ is achieved if $k > 10^3$ and these optima are somewhat lower than the optima of DICCG1$-k$ for both $n = 100^3$ and $n = 120^3$. In these cases, DICCG2$-k$ has been proven to be the most efficient method.

	$n = 60^3$			$n = 100^3$			$n = 120^3$		
k	# It.	CPU-1	CPU-2	# It.	CPU-1	CPU-2	# It.	CPU-1	CPU-2
2^3	166*	6.5	6.5	275	50.4	48.8	291*	90.5	101.0
5^3	63	2.8	2.8	97	19.0	18.7	111	35.9	36.8
10^3	42*	2.3	2.5	60	13.0	13.0	68	24.1	24.8
20^3	22	22.3	3.8	31	29.3	11.0	33	35.9	19.3
25^3	–	–	–	26	300	14.2	–	–	–
30^3	12	> 300	8.1	–	–	–	19	> 300	19.9
40^3	–	–	–	–	–	–	21	> 300	52.1

Table 7. Results of Test Problem 3D-TP4 (27 bubbles) for varying grid sizes $n = N_x \cdot N_y \cdot N_z$. The CPU time (in seconds) with DICCG1 is denoted by CPU-1, while the CPU time with DICCG2 is denoted by CPU-2.

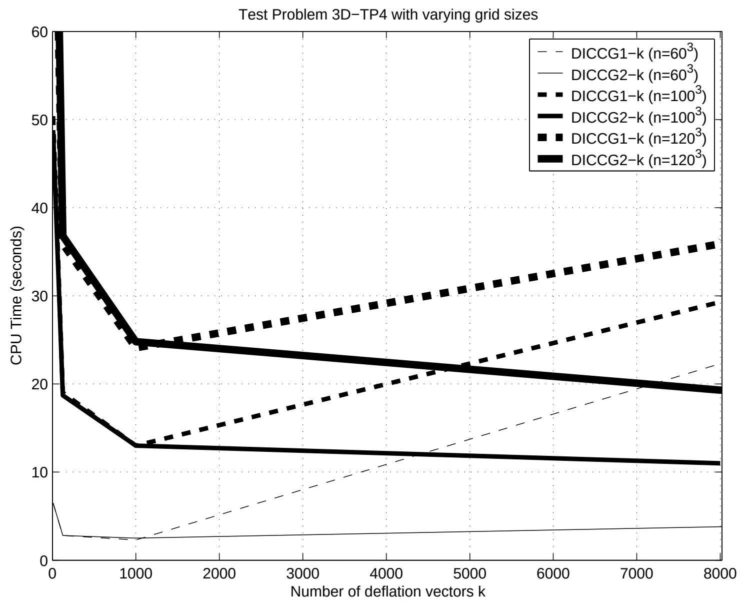


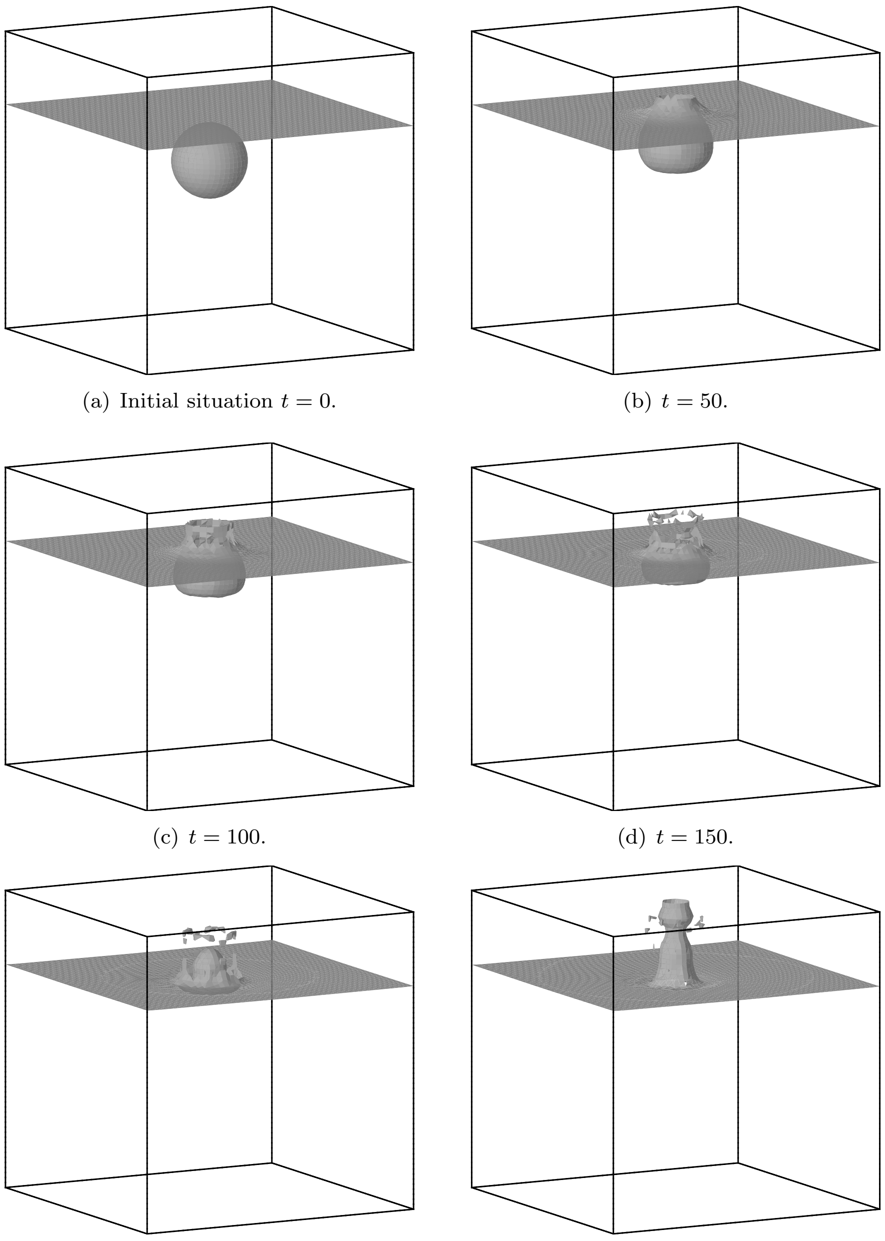
Figure 12. CPU Time of DICCG1$-k$ and DICCG2$-k$ for Test Problem 3D-TP4 with various grid sizes.

9. 3-D Realistic Simulations

In the previous two sections we have consider test problems with fixed geometries, i.e., the bubbles are chosen to be fixed in the computational domain and they do not evolve in time. In this section, we apply a 3D realistic simulations of 250 time steps. We adopt the mass-conserving level-set method [8] but it could be replaced by any operator-splitting method in general. At each time step a Poisson equation has to be solved which is the most time-consuming part of the whole simulation.

9.1. **Simulation 1: Rising Air Bubble in Water.**

We consider a test problem with a rising air bubble in water without surface tension. The exact material constants and other relevant conditions can be found in [8, Sect. 8.3.2]. The starting position of the bubble in the domain and the evolution of the movement during the 250 time steps can be seen in Figure 13.



(a) Initial situation $t = 0$.

(b) $t = 50$.

(c) $t = 100$.

(d) $t = 150$.

(e) $t = 200$.

(f) $t = 250$.

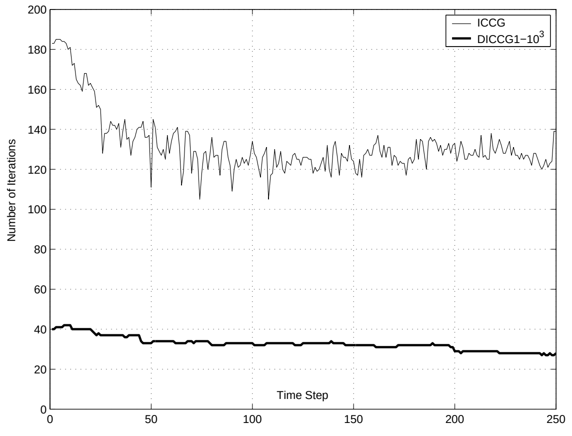
FIGURE 13. Evolution of the rising bubble in water without surface tension in the first 250 time steps.

In [8] the Poisson solver is based on ICCG. In this section we will compare this method with DICCG1-10^3 for both $n = 60^3$ and $n = 100^3$. In future, k can be adapted or DICCG2$-k$ can be applied to obtain a more efficient method.

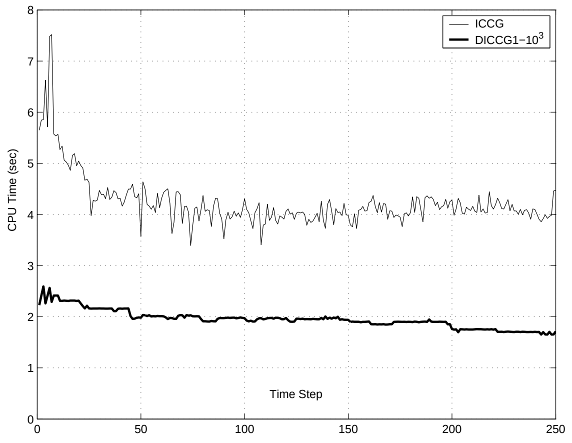
9.1.1. *Results with $N_x = N_y = N_z = 60$.* We present the results with grid sizes $N_x = N_y = N_z = 60$ in Figure 14.

From Figure 14 we can make the following observations.

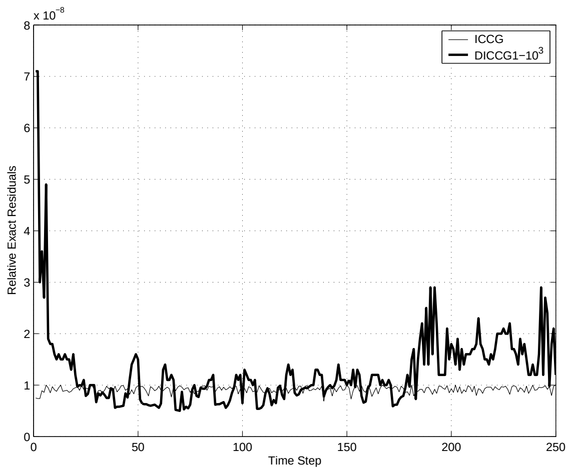
- Subfigure 14(a): Similar to previous sections, we notice that the number of iterations is strongly reduced by the deflation method. DICCG1-10^3



(a) Results considering the number of iterations.



(b) Results considering the CPU time.



(c) Results considering the relative exact residuals.

FIGURE 14. Results with $N_x = N_y = N_z = 60$ of Simulation 1.

requires more or less 30 iterations, while ICCG converges between 120 and 140 iterations in the most time steps. Moreover, observe the erratic behavior of ICCG, whereas DICCG-10^3 seems to be less sensitive of the geometries during the evolution of the simulation.

- Subfigure 14(b): Also considering the CPU time, DICCG1-10^3 shows very good performance. In most time steps ICCG requires 4 seconds to converge, whereas DICCG1-10^3 needs only around 2 seconds which is a benefit of approximately 50%.
- Subfigure 14(c): Obviously, the relative exact residuals are comparable and are of the same order for both ICCG and DICCG-10^3.

The main conclusion is however that it appears that DICCG-10^3 shows good performance for all time steps. This is rather unexpected, since the deflation vectors are fixed beforehand, i.e., they are chosen independent of the geometry of the problem.

In Figure 15 one can find the gain factors considering both the ratio's of the iterations and the CPU time between ICCG and DICCG1-10^3. From the figure, we conclude that DICCG1-10^3 needs approximately 3.5–5 times less iterations. More important, at all time steps DICCG1-10^3 converges more or less 2–2.5 times faster to the solution compared to ICCG.

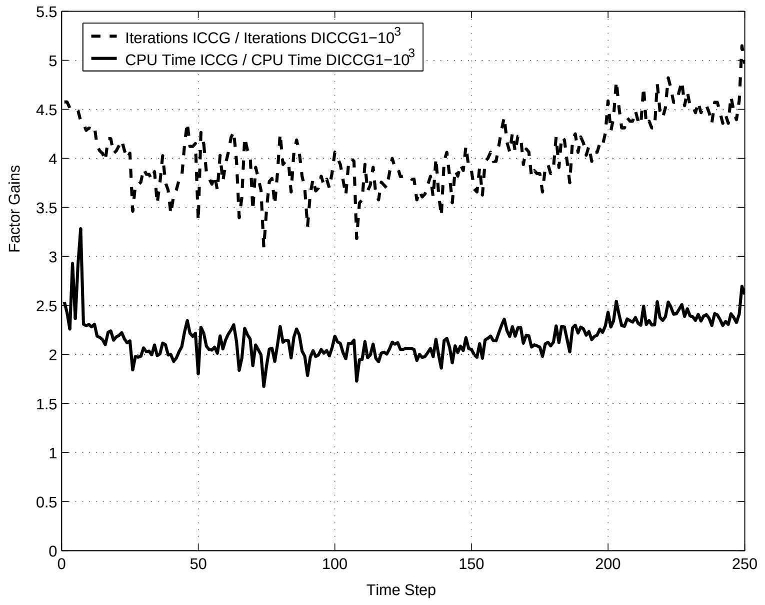
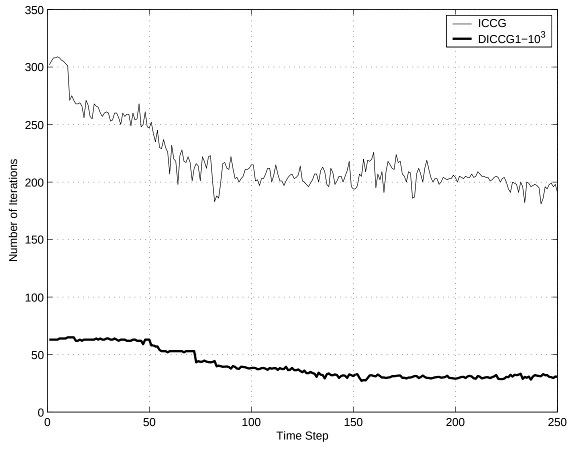


FIGURE 15. Visualization of the results of the realistic test problem with $N_x = N_y = N_z = 60$ of Simulation 1.

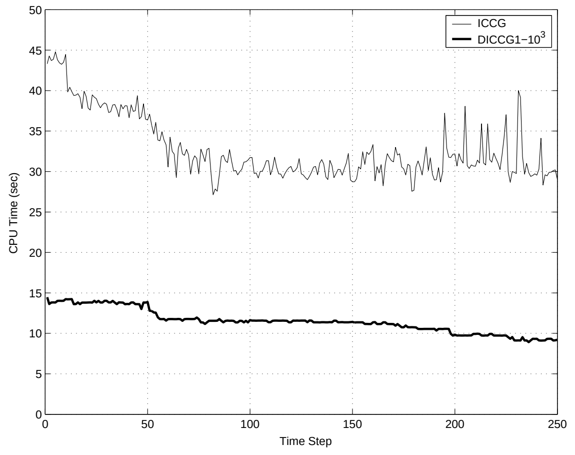
9.1.2. *Results with $N_x = N_y = N_z = 100$.* Next, we present the results with grid sizes $n = 100^3$ in Figure 16.

From Figure 16 we can make the following observSations.

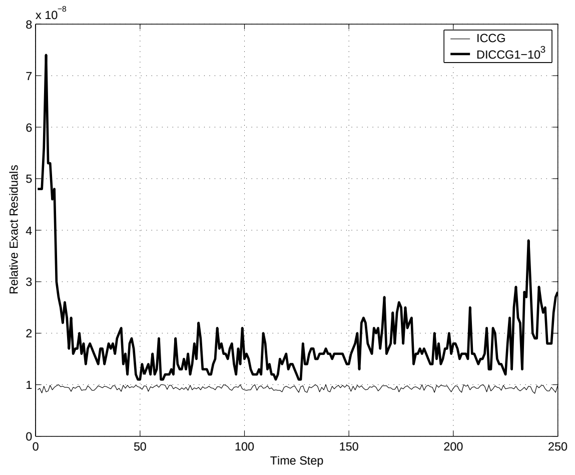
- Subfigure 16(a): Similar to the case of $n = 60^3$, we notice that the number of iterations is strongly reduced by the deflation method. DICCG1-10^3 requires more or less 60 iterations, while ICCG converges between 200 and 300 iterations in the most time steps. Moreover, observe again the erratic behavior of ICCG, whereas DICCG-10^3 seems to be less sensitive of the geometries during the evolution of the simulation.
- Subfigure 16(b): Also considering the CPU time, DICCG1-10^3 shows very good performance. In most time steps ICCG requires 25–45 seconds to



(a) Results considering the number of iterations.



(b) Results considering the CPU time.



(c) Results considering the relative exact residuals.

FIGURE 16. Results with $N_x = N_y = N_z = 100$ of Simulation 1.

converge, whereas DICCG1-10^3 needs only around 11–14 seconds which is a relatively large benefit.

- Subfigure 16(c): Obviously, the relative exact residuals are comparable for both ICCG and DICCG-10^3, where we can note that in the case of DICCG1-10^3 those residuals are almost constant while the residuals for ICCG shows a rather erratic behavior.

We can note that for larger problems the DICCG1-10^3 becomes more favorable compared to ICCG. Also in these larger problems, DICCG1-10^3 depends hardly on the geometry of the problem in each time step.

In Figure 17 one can find the gain factors considering both the ratio's of the iterations and the CPU time between ICCG and DICCG1-10^3. From the figure, we conclude that DICCG1-10^3 needs approximately 4–8 times less iterations, depending on the time step. More important, at all time steps DICCG1-10^3 converges more or less 2–4 times faster to the solution compared to ICCG. Compared to the case of $n = 60^3$ the gain is larger. Hence, the larger the problem the more favorable DICCG1-10^3 becomes.

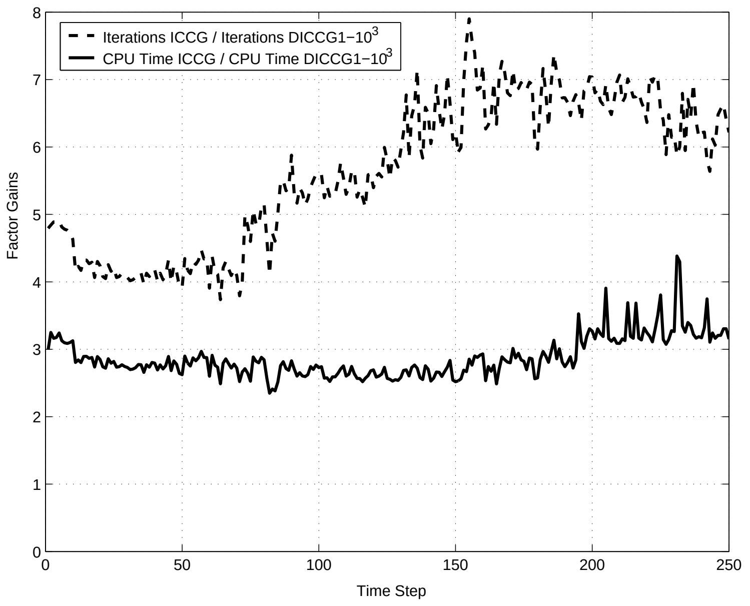


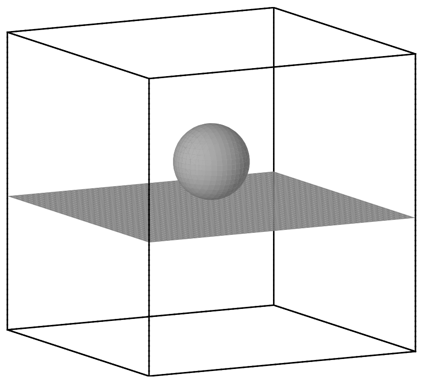
FIGURE 17. Visualization of the results of the realistic test problem with $N_x = N_y = N_z = 100$ of Simulation 1.

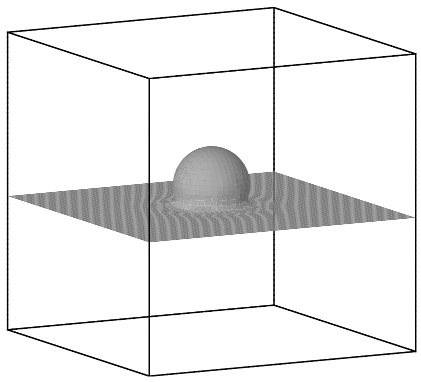
9.2. **Simulation 2: Falling Water Droplet in Air.** Similar to the previous subsection, we consider a test problem with a falling water droplet in air without surface tension. The exact material constants and other relevant conditions can be found in [8, Sect. 8.3.4]. The starting position of the bubble in the domain and the evolution of the movement during the 250 time steps can be observed in Figure 18.

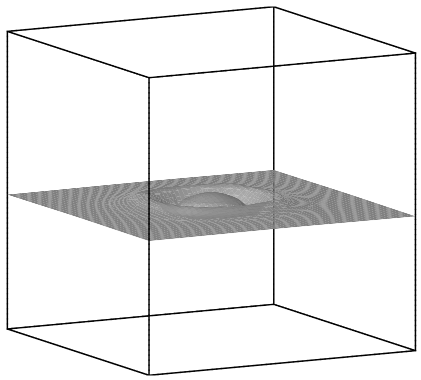
9.2.1. *Results with* $N_x = N_y = N_z = 60$. We present the results with grid sizes $N_x = N_y = N_z = 60$ in Figure 19.

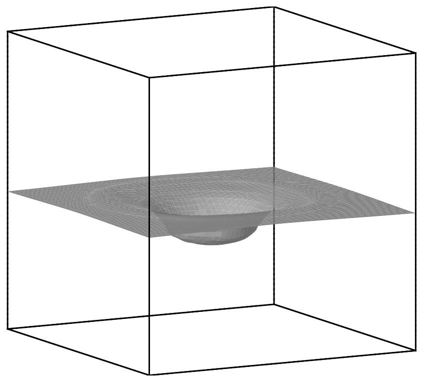
From Figure 19 we can make the following observations.

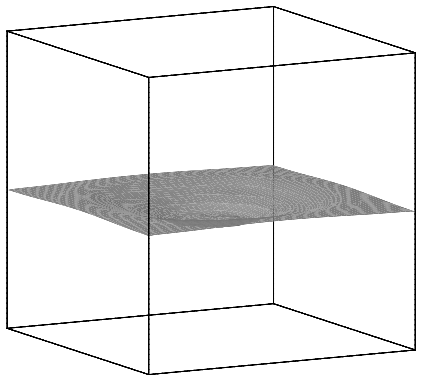
- Subfigure 19(a): Similar to previous sections, we notice that the number of iterations is strongly reduced by the deflation method.
- Subfigure 19(b): Also considering the CPU time, DICCG1-10^3 shows very good performance. Observe the two high peaks around time steps 30 and


(a) Initial situation $t = 0$.


(b) $t = 50$.


(c) $t = 100$.


(d) $t = 150$.


(e) $t = 200$.

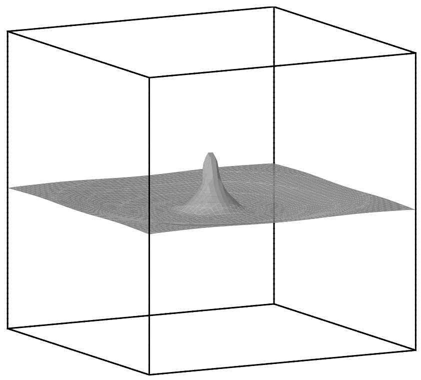

(f) $t = 250$.

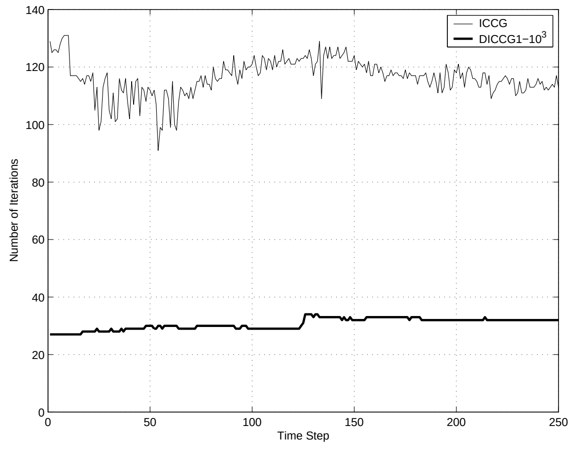
FIGURE 18. Evolution of the falling droplet in air without surface tension in the first 250 time steps.

245 which are caused by external factors. They should be omitted in the results.

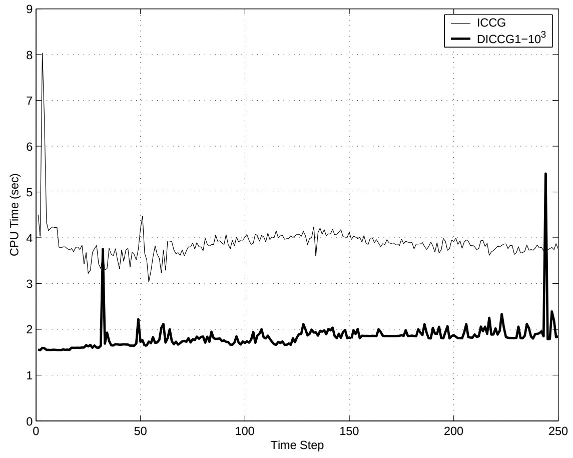
- Subfigure 19(c): Obviously, the relative exact residuals are comparable for both ICCG and DICCG-10^3.

When we compare the results of Simulation 1 and 2 we see that they are more or less comparable. DICCG1-10^3 is very efficient in both simulations.

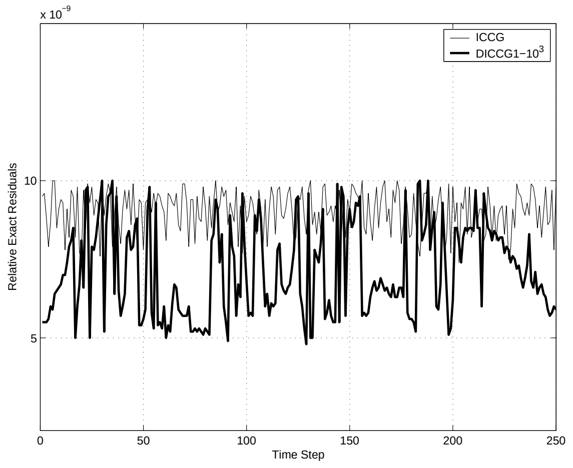
In Figure 20 one can find the factor gains considering both the ratio's of the iterations and the CPU time between ICCG and DICCG1-10^3. From the figure, we conclude that DICCG1-10^3 needs approximately 3.5–4.5 times less iterations.



(a) Results considering the number of iterations.



(b) Results considering the CPU time.



(c) Results considering the relative exact residuals.

FIGURE 19. Results with $N_x = N_y = N_z = 60$ of Simulation 2.

More important, at all time steps DICCG1-10^3 converges more or less 1.5–2.5 times faster to the solution compared to ICCG.

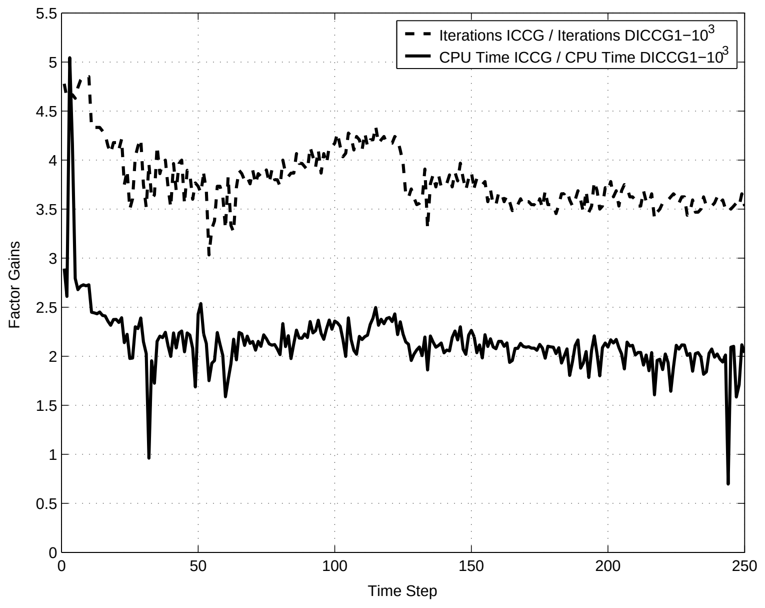


FIGURE 20. Visualization of the results of the realistic test problem with $N_x = N_y = N_z = 60$ of Simulation 2.

9.2.2. *Results with* $N_x = N_y = N_z = 100$. Next, we present the results with grid sizes $n = 100^3$ in Figure 21.

From Figure 21 we can make the following observations.

- Subfigure 21(a): Similar to the case of $n = 60^3$, we notice that the number of iterations is strongly reduced by the deflation method.
- Subfigure 21(b): Also considering the CPU time, DICCG1-10^3 shows very good performance.
- Subfigure 21(c): Obviously, the relative exact residuals are comparable for both ICCG and DICCG-10^3.

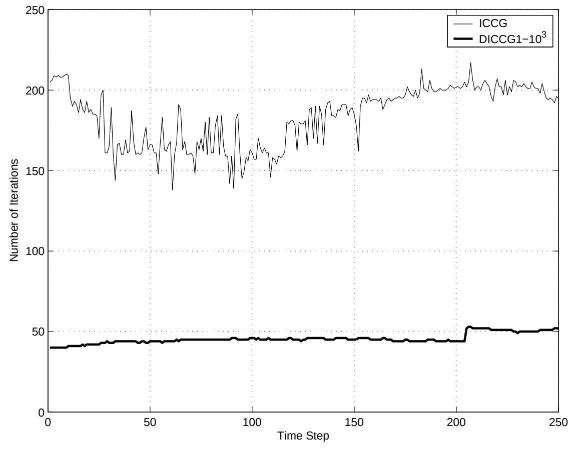
We can note that for larger problems the DICCG1-10^3 becomes more favorable compared to ICCG. Also in these larger problems, DICCG1-10^3 depends hardly on the geometry of the problem in each time step.

In Figure 22 one can find the factor gains considering both the ratio's of the iterations and the CPU time between ICCG and DICCG1-10^3. From the figure, we conclude that DICCG1-10^3 needs approximately 3–5 times less iterations, depending on the time step. More important, at all time steps DICCG1-10^3 converges more or less 2–3 times faster to the solution compared to ICCG. Compared to the case of $n = 60^3$ the gain is larger. Hence, the larger the problem the more favorable DICCG1-10^3 becomes.

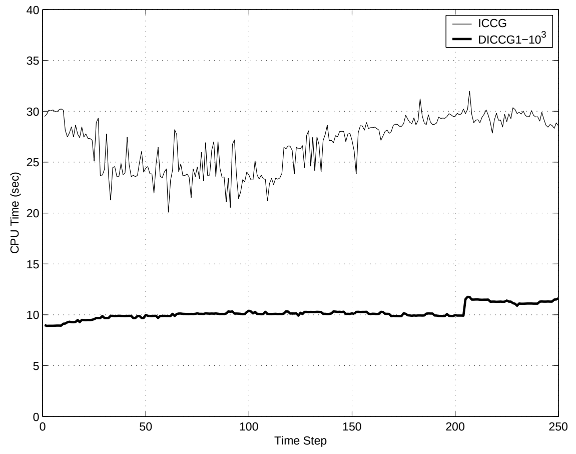
10. CONCLUSIONS

In [14] it has been concluded that the deflation method DICCG is very efficient in bubbly flow problems by considering the number of iterations. In this paper, we have shown that DICCG is even efficient by considering the CPU time.

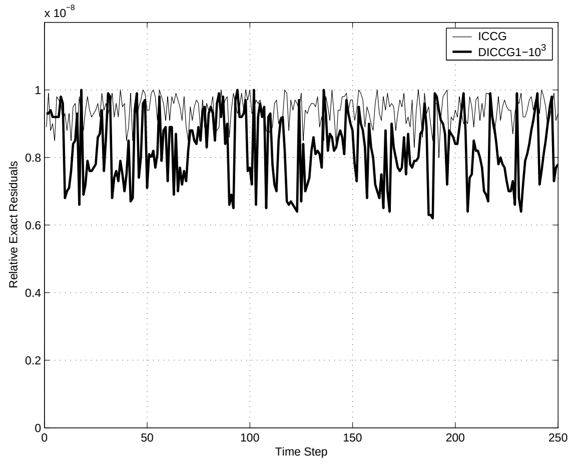
We have considered two variants of DICCG: one with a direct solver (DICCG1) and one with an iterative solver (DICCG2) for the coarse systems within the



(a) Results considering the number of iterations.



(b) Results considering the CPU time.



(c) Results considering the relative exact residuals.

FIGURE 21. Results with $N_x = N_y = N_z = 100$ of Simulation 2.

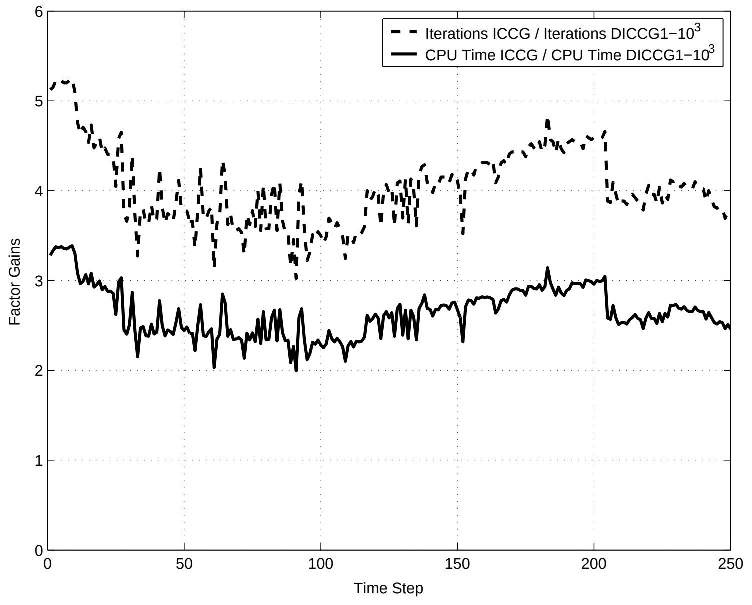


FIGURE 22. Visualization of the results of the realistic test problem with $N_x = N_y = N_z = 100$ of Simulation 2.

DICCG. Some theoretical properties considering these coarse systems have been derived which are of importance for DICCG2. Moreover, we have proven that the two variants DICCG1 and DICCG2 give the same convergence results in exact arithmetics.

Several 2-D and 3-D numerical experiments have been performed to check the efficiency of both variants of DICCG. For relatively small number of deflation vectors, DICCG1 performs very well but for a larger number DICCG2 is more efficient. With respect to ICCG, both variants reduce the computational costs significantly for all test cases of multi-phase flows, especially in relatively large problems. Moreover, DICCG is insensitive for the contrasts between the phases, while ICCG gives difficulties for large contrasts. Finally, by considering the number of iterations, DICCG seems also to be insensitive for the grid sizes, where the number of deflation vectors have to be taken proportional to these grid sizes.

The success of the deflation method has also been emphasized in 3-D realistic simulations, where 250 time steps have been carried out in the cases of a falling droplet in air and a rising bubble in water. The larger (i.e., the more grid points) the problems, the larger the benefit of the deflation method DICCG1 compared to ICCG.

References

[1] J. Frank and C. Vuik, *On the construction of deflation-based preconditioners*, SIAM J. Sci. Comp., **23**, pp. 442–462, 2001.

[2] E.F. Kaasschieter, *Preconditioned conjugate gradients for solving singular systems*, J. Comp. Appl. Maths., **24**, pp. 265–275, 1988.

[3] G.H. Golub, C.F. van Loan, *Matrix Computations*, Third Edition, The John Hopkins University Press, Baltimore, Maryland 21218, 1996.

[4] R. Horn and C. Johnson, *Matrix Analysis*, Cambridge University Press, USA Edition, 1990.

[5] J.A. Meijerink and H. A. Van der Vorst, *An iterative solution for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp., **31**(137), pp. 148–162, 1977.

[6] R. Nabben and C. Vuik, *A comparison of Deflation and Coarse Grid Correction applied to porous media flow*, SIAM J. Numer. Anal., **42**, pp. 1631–1647, 2004.

[7] R.A. Nicolaides, *Deflation of Conjugate Gradients with applications to boundary value problems*, SIAM J. Matrix Anal. Appl., **24**, pp. 355–365, 1987.

[8] S.P. van der Pijl, *Computation of bubbly flows with a mass-conserving level-set method*, PHD thesis, Delft University of Technology, 2005.

[9] S.P. van der Pijl, A. Segal, C. Vuik, and P. Wesseling, *A mass-conserving Level-Set method for modelling of multi-phase flows*, Int. J. Numer. Meth. Fluids , **47**, pp. 339–361, 2005.

[10] V. Simoncini and D.B. Szyld, *Theory of Inexact Krylov Subspace Methods and Applications to Scientific Computing*, SIAM J. Sc. Comp., **25**, pp. 454–477, 2003.

[11] V. Simoncini and D.B. Szyld, *On the Occurrence of Superlinear Convergence of Exact and Inexact Krylov Subspace Methods*, SIAM Review, **25**, pp. 247–272, 2005.

[12] F.S. Sousa, N. Mangiavacchi, L.G. Nonato, A. Castelo, M.F. Tome, V.G. Ferreira, J.A. Cuminato and S. McKee, *A Front-Tracking / Front-Capturing Method for the Simulation of 3D Multi-Fluid Flows with Free Surfaces*, J. Comp. Physics, **198**, pp. 469–499, 2004.

[13] J.M. Tang, *Parallel Deflated CG Methods applied to Linear Systems from Moving Boundary Problems*, DUT Report 05-02, Delft University of Technology, 2005.

[14] J.M. Tang and C. Vuik, *On Deflation and Singular Symmetric Positive Semi-Definite Matrices*, *submitted to* J. Comp. Appl. Math., 2005.

[15] C. Vuik, A. Segal, J.A. Meijerink and G.T. Wijma, *The construction of projection vectors for a Deflated ICCG method applied to problems with extreme contrasts in the coefficients*, J. Comp. Physics, **172**, pp. 426–450, 2001.

Appendix A. Efficient Construction of S_{AZ} and S_E in 2-D

A.1. **Geometry and Terminology.** Sparse matrix S_{AZ} consists of three columns, while the number of rows is equal to the number of non-zeros, denoted by γ, in the full matrix AZ. The first column consists of the row number of AZ. The second column is the corresponding subdomain, which is the column number of AZ, and the third column is the corresponding value of AZ.

Each deflation vector in Z corresponds to one subdomain in Ω. If we assume Ω to be a square, then these subdomains can be divided into nine different groups, as depicted in Figure 23. Note that all groups (except for the corner groups 1, 3, 7, 9) may consist of more subdomains. For instance, for $r = 25$ Group 5 consists of exactly 16 subdomains, while Group 2, 4, 6 and 8 consist of 4 subdomains each.

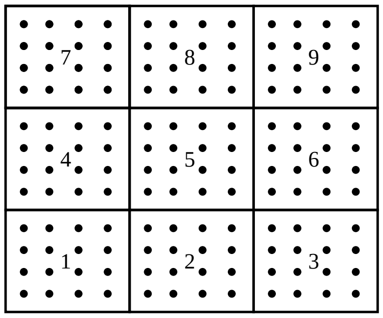


Figure 23. Square domain Ω divided in nine subdomains ($k = 9$). In this situation each subdomain correspond to exactly one group.

In Figure 24, we can see the grid points which are concerned in the computation of S_{AZ}. Moreover, one can find the different cases concerning these grid points.

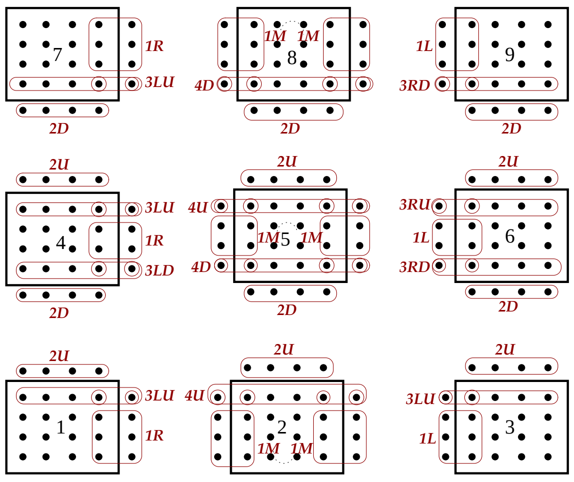


Figure 24. Cases of grid points concerned in AZ.

Next, the variables used in this appendix are explained in the Table 8. Note that in the case of Figure 24, subdomains 1, 3, 7, 9 represent all corner s_d, subdomains 2, 4, 6, 8 represent all boundary s_d and, finally, subdomain 5 is the only interior s_d.

A.2. **Number of Elements γ.** The number of elements γ can be computed easily.

Variable	Meaning
s_d	Subdomain
k	Number of s_d
k_x	Number of s_d in each direction ($= \sqrt{k}$)
n_b	Number of grid points in each direction of each s_d
γ	Total number of non-zeros of AZ
γ_c	Number of non-zeros of AZ coming from all corner s_d
γ_b	Number of non-zeros of AZ coming from all boundary s_d
γ_i	Number of non-zeros of AZ coming from all interior s_d

TABLE 8. Explanations of the variables.

- *Corner Subdomains.* For the corner subdomains, which are Group 1, 3, 7, 9 as can be observed in Figure 23, we have $4n_b - 1$ elements of fill-in for each subdomain. In total, there are four corner subdomains. Hence,
$$\gamma_c = 4(4n_b - 1)$$
elements belongs to all corner subdomains.
- *Boundary Subdomains.* For the boundary subdomains (Group 2, 4, 6, 8) we have $6n_b - 2$ elements of non-zeros for each subdomain. In total, there are $4(k_x - 2)$ boundary subdomains. Therefore,
$$\gamma_b = 8(3n_b - 1)(k_x - 2).$$
- *Interior Subdomains.* For the interior subdomains which is Group 5 we have $8n_b - 4$ non-zero elements for each subdomain. Since there are $(k_x - 2)^2$ interior subdomains, this implies
$$\gamma_i = 4(2n_b - 1)(k_x - 2)^2.$$

We sum up all elements of the corner, boundary and interior subdomains to obtain the total number of non-zeros γ:

$$\gamma = \gamma_c + \gamma_b + \gamma_i = 4(4n_b - 1) + 8(3n_b - 1)(k_x - 2) + 4(2n_b - 1)(k_x - 2)^2. \quad (26)$$

Obviously, if k is large then γ_i is the dominant term in Eq. (26).

A.3. **Treatment of the Cases.** Each case, as can be observed in Figure 24, will be considered separately.

A.3.1. *Case 1 (1R, 1L, 1M).* We distinguish the cases ‘left’ (l), ‘right’ (r) or ‘middle’ (m) variant of Case 1. Then for each row of the domain we add the elements corresponding to the concerned grid points to S_{AZ}, where we use the fact that

$$-a_{i,j} = \sum_{k \neq j} a_{i,k}, \quad \forall i, \quad (27)$$

since from Assumption 1 we know that $A\mathbf{1}_n = \mathbf{0}_n$. For instance, for the case 'left' we add two elements to S_{AZ} in each row: for the first element x, we add the corresponding right element of A with a negative sign in front and for the second element $x + 1$ we add the corresponding left element of A to S_{AZ}.

Note further that for variant ‘middle’ the required work is twice as much compared to ‘left’ or ‘right’.

A.3.2. *Case 2 (2U, 2D).* Two variants ‘upper’ (u) and ‘down’ (d) are distinguished in this case. The corresponding elements of S_{AZ} can be easily computed: for ‘upper’ we add the corresponding under element of A and for ‘down’ we add the corresponding upper element of A to S_{AZ}.

A.3.3. *Case 3 (3LU, 3LD, 3RU, 3RD, 3MU, 3MD).* This case requires six different variants. Each variant takes a few steps, since the 'corner' points have to be treated differently compared by the 'boundary' points. For instance, in variant 3LU we first compute the elements for the boundary points using Eq. (27). Then, the two corner points are treated separately, using again (27).

A.3.4. *Case 4 (4D, 4U).* We distinguish the variants 4D and 4U. The procedure of each variant resembles strongly the procedure of the variants in Case 3. Instead of two corner points, now we have 4 corner points which has to be dealt with.

A.4. **Constructing S_{AZ}.** Since all cases are considered, computing S_{AZ} is straightforward. Each subdomain will be treated during the process by computing the components belonging to each case. For the case of $k = 4$, it consists of four subdomains with three cases each. In the case of $k > 4$, we note that Group 2 (or 4, 6, 8) in Figure 2 appears $k_x - 2$ times in the general case. Similarly, we deduce that Group 5 appear $(k_x - 2)^2$ times in these computations.

Note further that in the case that A is forced to be invertible, we have to correct S_{AZ} by noting that

$$(AZ)_{n,n} = \sigma a_{n,n}, \quad \tilde{a}_{n,n} = \sigma a_{n,n},$$

and hence,

$$(AZ)_{n,n} = \frac{\sigma}{1+\sigma} \tilde{a}_{n,n}.$$

A.5. **Constructing S_E.** After constructing S_{AZ}, we continue with efficiently constructing of $E := Z^T AZ \in \mathbb{R}^{k \times k}$. E is a small and sparse SPD matrix with the same nonzero pattern as A. Earlier we have seen the different cases in the computations of S_{AZ}. Now, these cases will again be the basis of our method. Obviously, to compute S_E we need all non-zero non-zeros of AZ exactly one time. The geometry of the method is given in Figure 25.

Some remarks can be made about this figure.

- E is symmetric and therefore not all non-zero elements in AZ are needed in the computation of E.
- S_E will be stored efficiently in a matrix with three columns (k elements each): $S_E := [E1 \ E2 \ E3]$, where $E1$ is the main diagonal, $E2$ the first subdiagonal and $E3$ the second subdiagonal of matrix E. All indicated interior points contribute to E1, while all right and top elements besides the interior are contributions to E2 and E3, respectively. Later on, zero columns can be put between E2 and E3 which can be filled during the Cholesky decomposition.
- The construction of S_E can be implemented in the existing code for computing S_{AZ}, which is rather straightforward.

Appendix B. Efficient Construction of S_{AZ} and S_E in 3-D

In 2-D we have used Figure 24, where the possible groups and cases have been distinguished. We can also generalize this idea to the 3-D case, where each subdomain is a block now. First we start with the case of eight blocks, thereafter we consider the case of 28 blocks and finally we consider S_{AZ} with various number of blocks.

B.1. **Matrix S_{AZ} with Eight Blocks ($k = 2^3 = 8$).** We first start with considering eight blocks in 3-D. This means that from Figure 24 we need only Blocks 1, 3, 7 and 9. The other blocks can be omitted for the time being. The geometry of the eight blocks in 3-D can be found in Figure 26. We will treat Block 1 extensively. The remaining blocks can be treated similarly and this will be done shortly in this appendix.

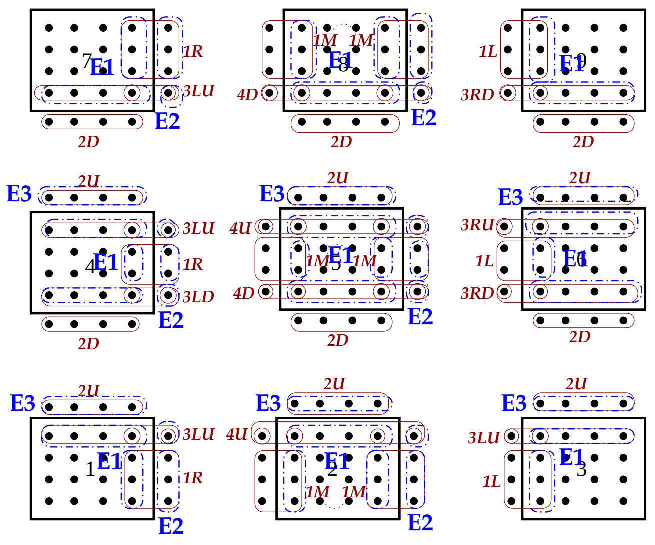


FIGURE 25. Cases of grid points concerned in $E := Z^T AZ$ which are indicated with E1, E2 and E3.

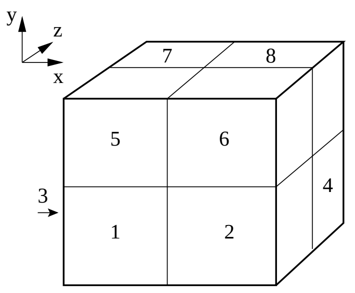


FIGURE 26. Geometry of the blocks in 3-D (eight blocks).

B.1.1. *Treatment of Block 1.* Block 1 is virtually divided into layers. Each layer corresponds to one $z-$position. It appears that the layers of $z = 1, \ldots, n_b - 1$ are all the same, see also Figure 27. In other words, for the layers $z = 1, \ldots, n_b - 1$,

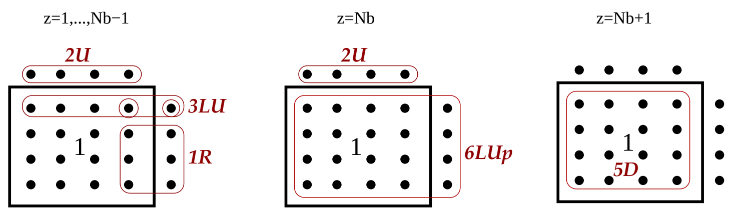


FIGURE 27. Treatment of Block 1 in 3D.

they are the same as Block 1 in the 2-D case. For layer $z = n_b$ each element of the block counts an extra element (at $z = n_b + 1$). Therefore, we introduce Case 6 which encounters these elements. Cases 1 and 3 are not required anymore. For layer $z = n_b + 1$, each interior point of the block has a contribution which is treated in Case 5.

B.1.2. *Blocks 1–8.* The treatment of the remaining blocks is similar to the above analysis of Block 1. In Table 9 it has been summarized which cases are involved in these blocks.

Subdomain	z	Cases
1	$1, \ldots, n_b - 1$	1R, 3LU, 2U
	n_b	6LUp, 2U
	$n_b + 1$	5D
2	$1, \ldots, n_b - 1$	1L, 3RU, 2U
	n_b	6RUp, 2U
	$n_b + 1$	5D
3	$1, \ldots, n_b - 1$	2U, 3LD, 1R
	n_b	2U,6LDp
	$n_b + 1$	5D
4	$1, \ldots, n_b - 1$	2D, 3RD, 1L
	n_b	2D, 6RDp
	$n_b + 1$	5D
5	n_b	5U
	$n_b + 1$	6RUn, 2U
	$n_b + 2, \ldots, 2n_b$	1L, 3RU, 2U
6	n_b	5U
	$n_b + 1$	6RUn, 2U
	$n_b + 2, \ldots, 2n_b$	1L, 3RU, 2U
7	n_b	5U
	$n_b + 1$	2D
	$n_b + 2, \ldots, 2n_b$	6LDn
8	n_b	5U
	$n_b + 1$	2D, 6RDn
	$n_b + 2, \ldots, 2n_b$	2D, 3RD, 1L

TABLE 9. Treatment of the Blocks for $k = 8$.

B.2. **Matrix S_{AZ} with 27 Blocks ($k = 3^3 = 27$).** In the case of $r = 27$, the constructed eight blocks in the previous section are the eight corner blocks. The remaining 19 blocks can be constructed in a similar way. To do so, we start with the lower nine blocks.

B.2.1. *Blocks 1–9.* The treatment of the blocks are given in Table 10.

B.2.2. *Blocks 10–18.* The implementation of blocks 10–18 follows from the first nine blocks. Now, there are five groups for z instead of three, namely, $z = n_b, z = n_b + 1, z = n_b + 2, \ldots, 2n_b - 1, z = 2n_b, z = 2n_b + 1$. The last three groups are the same as the block below, and the first two follows immediately from the last two. For example, Block 10 consists of

- $z = n_b : 5U$;
- $z = n_b + 1 : 6LUn$;
- $z = n_b + 2, \ldots, 2n_b - 1 : 1R, 3LU, 2U$;
- $z = 2n_b : 6LUp, 2U$;
- $z = 2n_b + 1 : 5D$.

In this case, $z = n_b + 2, \ldots, 2n_b - 1 : 1R, 3LU, 2U, z = 2n_b : 6LUp, 2U, z = 2n_b + 1 : 5D$ are exactly the same as Block 1. Moreover, $z = n_b : 5U, z = n_b + 1 : 6LUn$ are almost the same as $z = 2n_b : 6LUp, 2U, z = 2n_b + 1 : 5D$, where only 'p' is replaced by 'n' in Case 6. This same pattern holds for all Blocks 10–18.

Block	z	Cases
1	$1,\ldots,n_b-1$	1R, 3LU, 2U
	n_b	6LUp, 2U
	n_b+1	5D
2	$1,\ldots,n_b-1$	1M, 4U, 2U
	n_b	6MUp, 2U
	n_b+1	5D
3	$1,\ldots,n_b-1$	2U, 3LD, 1R
	n_b	2U,6LDp
	n_b+1	5D
4	$1,\ldots,n_b-1$	2D, 3LD, 1R, 3LU, 2U
	n_b	2D, 6LMp, 2U
	n_b+1	5D
5	$1,\ldots,n_b-1$	2D, 4D, 1M, 4U,2U
	n_b	2D, 6MMp, 2U
	n_b+1	5D
6	$1,\ldots,n_b-1$	2D, 3RD, 1L, 3RU, 2U
	n_b	2D, 6MMp, 2U
	n_b+1	5D
7	$1,\ldots,n_b-1$	2U, 3LD, 1R
	n_b	2U,6LDp
	n_b+1	5D
8	$1,\ldots,n_b-1$	2D, 4D, 1M
	n_b	2D, 6MDp
	n_b+1	5D
9	$1,\ldots,n_b-1$	2D, 3RD, 1L
	n_b	2D, 6RDP
	n_b+1	5D

TABLE 10. Treatment of Blocks 1–9 for $k = 27$.

B.2.3. *Blocks 19–27.* Also Blocks 19–27 follow from Blocks 1–9. The different groups of Blocks 1–9 have to be reversed and moreover, 'p' has to be replaced by 'n' in Case 6 and 'D' has to be replaced by 'n' in Case 5. For instance, Block 20 consists of

- $z = 2n_b : 5U;$
- $z = 2n_b + 1 : 6MUn; 2U$
- $z = 2n_b + 2, \ldots, 3n_b : 1M, 4U, 2U$

This is exactly the reverse treatment of Block 2, where now 5D and 6LUp are 5U and 6LUn, respectively. This same pattern holds for all Blocks 19–27.

B.3. **Matrix S_{AZ} with Variable Number of Blocks.** Implementing matrix S_{AZ} with variable number of blocks is a straightforward generalization of the case with 27 blocks. Each of the 27 blocks should be considered to be a class in this case, where all new blocks will be covered.

B.4. **Constructing S_E.** In similar way as for the 2-D case, S_E can be constructed. Instead of $S_E = [E1\ E2\ E3]$, now we have $S_E = [E1\ E2\ E3\ E4]$.

B.5. **Number of Elements γ.** Similar to the 2-D case, we compute γ which is required to construct S_{AZ}.

- *Corner Blocks.* We need $(n_b - 1)$ times $4n_b - 1$ and one time $2n_b^2 + 2n_b$ for each corner block. So, $6n_b^2 - 3n_b + 1$ elements per corner block are required.

In total we have 8 corner blocks. Hence,

$$\gamma_c = 8 \cdot (6n_b^2 - 3n_b + 1).$$

- *Interior Blocks.* $(n_b - 2)$ times $8n_b - 4$ and two times $2n_b^2 + 4n_b$ for each interior block are required. Therefore, $12n_b^2 - 12n_b + 8$ elements are required for each interior block. There are $(k_x - 2)^3$ interior blocks. Hence,

$$\gamma_i = (k_x - 2)^3 \cdot (12n_b^2 - 12n_b + 8).$$

- *Boundary Blocks.* We divide the boundary blocks into two parts: (i) the 'real' boundary blocks and the 'boundary-interior' blocks.

  (i) We need $(n_b - 1)$ times $6n_b - 2$ and one time $2n_b^2 + 3n_b$ for each real boundary block. So in total: $8n_b^2 - 5n_b + 2$ elements are required for each real boundary block. There are exactly $12(k_x - 2)$ real boundary blocks.

  (ii) We need $(n_b - 1)$ times $8n_b - 4$ and one time $2n_b^2 + 4n_b$ for each boundary-interior block. Hence, $10n_b^2 - 8n_b + 4$ elements are required for each boundary-interior block. In total, we have $6(k_x - 2)^2$ boundary-interior blocks.

  Hence,

$$\gamma_b = 12(k_x - 2) \cdot 8n_b^2 - 5n_b + 2 + 6(k_x - 2)^2 \cdot 10n_b^2 - 8n_b + 4.$$

Subsequently, the total number γ can be computed using

$$\gamma = \gamma_c + \gamma_i + \gamma_b.$$

Note that if A is invertible, one extra element is required, similar to the 2-D case.

Appendix C. Flop Counting of ICCG and DICCG1 in 2-D

A comparison of flops between ICCG and DICCG1 has been performed in [13, Appendix B]. However, AZ and E have not been computed efficiently in that paper. We will revise it based on S_{AZ} and S_E in this and the next appendix. In this appendix, the 2-D case will be considered shortly where we compare ICCG and DICCG1. For more details we refer to [13, Appendix B].

C.1. **Assumptions.** To compare the number of flops between DICCG1 and ICCG in the 2-D case, we assume that

- A has size $n \times n$ and consists of 5 non-zero diagonals;
- AZ is computed and stored efficiently using S_{AZ};
- E has size $k \times k$ and has bandwidth $\sqrt{k}$. E is computed and stored efficiently using S_E.

Compared to [13, Appendix B], Z and AZ will not explicitly be formed, which restrict the number of flops.

We make some assumptions with respect to the (approximated) number of flops in Table 11, where x, y are vectors of length n and v, w are vector of length k. In the analysis, we shall neglect $\mathcal{O}(1)$ terms.

Notation	**Operation**	**# Flops**
$F_{(x,y)}$	(x, y)	$2n$
F_{x+y}	$x + y$	n
F_{Ax}	Ax	$9n$
$F_{chol(A)}$	Making C from A	$8n$
$F_{Ax=y}$	Solving x from $CC^T x = y$	$11n$

Table 11. Assumptions of flop counts in standard operations of both ICCG and DICCG1.

C.2. **Flop Counts of Main Operations of DICCG1.** In this subsection, we compute the number of flops required to construct S_{AZ}, S_E, $(AZ)x_2$, $Z^T x_2$, $(AZ)^T x_1$, Zx_1, solving $Ev = w$ and computing Px and PAx.

C.2.1. *Computations S_{AZ} and S_E.* From Appendix A, the number of rows of S_{AZ} is given by

$$\begin{aligned}\gamma &= 4(4n_b-1)+8(3n_b-1)(k_x-2)+4(2n_b-1)(k_x-2)^2\\ &= 16n_b-4+(24n_b-8)(k_x-2)+(8n_b-4)(k_x^2-4k_x+4)\\ &= 8n_bk_x^2-8n_bk_x-4k_x^2+8k_x-4,\end{aligned}$$

where

$$n_b=\sqrt{n/k},\quad k_x=\sqrt{k}.$$

Subsituting the latter expression into γ yields

$$\begin{aligned}\gamma &= 8k\sqrt{n/k}-8\sqrt{n/k}\sqrt{k}-4k+8\sqrt{k}-4\\ &= 8\sqrt{nk}-8\sqrt{n}-4k+8\sqrt{k}-4.\end{aligned}$$

Almost each element of S_{AZ} has exactly one contribution from A. Only the corner points of each subdomain have more contributions from A. The flops of these extra contributions can be neglected since $n \gg k$. The number of flops F_{AZ} to create AZ efficiently is (neglecting $\mathcal{O}(1)$ terms):

$$F_{AZ}=8\sqrt{nk}-8\sqrt{n}-4k+8\sqrt{k}=\mathcal{O}(\sqrt{nk}).$$

Next, F_E can be computed easily. This is equal to γ, because each non-zero element of AZ has a contribution once to E. This leads to

$$F_E\approx F_{AZ}=8\sqrt{nk}-8\sqrt{n}-4k+8\sqrt{k}=\mathcal{O}(\sqrt{nk}).$$

C.2.2. *Computations $(AZ)x_2$, Z^Tx_2, $(AZ)^Tx_1$ and Zx_1.* The number of flops for the matrix-vector computation $(AZ)x_2$ can be computed easily, since AZ is stored as S_{AZ}. A loop is required to explore S_{AZ}. In each iterate of this loop, we need two flops: the value of the element multiplied by the corresponding element from x_2 and the addition to the previous value of the element of the new vector. Based on this idea, we find the same number of flops for $(AZ)^Tx_1$. In other words,

$$F_{AZx_2}=F_{(AZ)^Tx_1}=2\gamma\approx 16\sqrt{nk}-16\sqrt{n}-8k+16\sqrt{k}=\mathcal{O}(\sqrt{nk}).$$

Moreover, matrix Z is not formed explicitly, since Z^Tx_2 en Zx_1 can be constructed directly from x_2 en x_1. Here, x_1 and x_2 are vectors of length k and n, respectively. Each element of the vectors has exactly one contribution to the new vectors. A loop is again required, where in each iterate one flop are needed: element from x_2 added to the corresponding element in the new vector. In a similar way we obtain that both Zx_1 and Z^Tx_2 require the same number of flops. Therefore,

$$F_{Zx_2}=F_{Z^Tx_1}=n=\mathcal{O}(n).$$

Note that constructing Zx_2 is more expensive than constructing AZx_2 by assuming that S_{AZ} is already formed.

C.2.3. *Computations for Solving $Ev = w$ directly.* Solving $Ev = w$ will be performed by first constructing the band-Cholesky decomposition L from E and thereafter solving v from $LL^Tv = w$. As described in [13, Appendix B],

$$F_{chol(E)}=k^2+8k\sqrt{k}+k=\mathcal{O}(k^2)$$

is needed to compute the band-Cholesky decomposition of E. Then the number of flops to solve $Ev = w$ with the backward- and forward substitution is exactly

$$F_{Ev=w}=2k\sqrt{k}+k=\mathcal{O}(k\sqrt{k}).$$

C.2.4. *Computations Px and PAx.* To compute Px we assume that $chol(E)$ and S_{AZ} are known. Note that $F_{x-x_3} = n$ and $F_{Ax} = 9n$ from Table 11. Then we have

$$\begin{aligned} F_{Px} &= F_{Z^Tx_1} + F_{Ev=w} + F_{AZx_2} + F_{x-x_3} \\ &= n + 2k\sqrt{k} + k + 16\sqrt{nk} - 16\sqrt{n} - 8k + 16\sqrt{k} + n \\ &= 2n + 16\sqrt{nk} - 16\sqrt{n} + 2k\sqrt{k} - 7k + 16\sqrt{k} \\ &= \mathcal{O}(n), \end{aligned}$$

and moreover, we obtain immediately

$$\begin{aligned} F_{PAx} &= F_{Px} + F_{Ax} \\ &= 11n + 16\sqrt{nk} - 16\sqrt{n} + 2k\sqrt{k} - 7k + 16\sqrt{k} \\ &= \mathcal{O}(n). \end{aligned}$$

C.2.5. *Summary.* For convenience, we summarize the flop counting results of this subsection in Table 12.

Notation	Operation	# Flops
F_{AZ}	S_{AZ}	$8\sqrt{nk} - 8\sqrt{n} - 4k + 8\sqrt{k}$
F_E	S_E	$8\sqrt{nk} - 8\sqrt{n} - 4k + 8\sqrt{k}$
F_{AZx_2}	AZx_2	$16\sqrt{nk} - 16\sqrt{n} - 8k + 16\sqrt{k}$
$F_{(AZ)^Tx_1}$	$(AZ)^Tx_1$	$16\sqrt{nk} - 16\sqrt{n} - 8k + 16\sqrt{k}$
F_{Zx_2}	Zx_2	n
$F_{Z^Tx_1}$	Z^Tx_1	n
$F_{chol(E)}$	$chol(E)$	$k^2 + 8k\sqrt{k} + k$
$F_{Ev=w}$	$Ev = w$	$2k\sqrt{k} + k$
F_{Px}	Px	$2n + 16\sqrt{nk} - 16\sqrt{n} + 2k\sqrt{k} - 7k + 16\sqrt{k}$
F_{PAx}	PAx	$11n + 16\sqrt{nk} - 16\sqrt{n} + 2k\sqrt{k} - 7k + 16\sqrt{k}$

TABLE 12. Flops required to compute the main steps in DICCG1.

C.3. **Flop Counts in ICCG and DICCG1.** We divide this subsection into two parts. The first part computes the number of flops required prior to and after the ICCG and DICCG1 loops. The second part determines the number of flops required in both loops.

C.3.1. *Computations before and after the ICCG and DICCG1 loops.* DICCG1 requires more prior and post computations compared to ICCG. These extra computations are determined below and they are denoted by F_{prior} and F_{after}. The results are based on computations done in [13, Appendix B].

For F_{prior}, we obtain

$$\begin{aligned} F_{prior} &= F_{AZ} + F_E + F_{chol(E)} \\ &= 16\sqrt{nk} - 16\sqrt{n} - 8k + 16\sqrt{k} + k^2 + 8k\sqrt{k} + k \\ &= 16\sqrt{nk} - 16\sqrt{n} + k^2 + 8k\sqrt{k} - 7k + 16\sqrt{k} \\ &= \mathcal{O}(\sqrt{nk}). \end{aligned}$$

To compute F_{after} we have to determine the number of flops F_u and F_x, needed for $u = ZE^{-1}Z^Tb$ and $x = u + P^T\tilde{x}_j$:

$$\begin{aligned} F_u &= F_{Z^Tx_1} + F_{Ev=w} + F_{Zx_2} \\ &= 3n + 2k\sqrt{k} + k \\ &= \mathcal{O}(n), \end{aligned}$$

and

$$\begin{aligned} F_x &= n + F_{Px} = \\ &= 3n + 16\sqrt{nk} - 16\sqrt{n} + 2k\sqrt{k} - 7k + 16\sqrt{k} \\ &= \mathcal{O}(n). \end{aligned}$$

Hence,

$$\begin{aligned} F_{after} &= F_u + F_x \\ &= 4n + 2k\sqrt{k} + k + 4n + 16\sqrt{nk} - 16\sqrt{n} + 2k\sqrt{k} - 7k + 16\sqrt{k} \\ &= 6n + 4k\sqrt{k} - 6k + 16\sqrt{nk} - 16\sqrt{n} + 16\sqrt{k} \\ &= \mathcal{O}(n). \end{aligned}$$

Next, ICCG and DICCG1 have several common computations which leads to

$$F_{prior-com} = 31n.$$

C.3.2. *Computations during the ICCG and DICCG1 loops.* In the iterates, ICCG and DICCG1 have also common operations which results in (see [13, Appendix B])

$$F_{com} = 27n.$$

Moreover, the difference between ICCG and DICCG1 in the iterates is computing w_j. In DICCG1, we compute $w_j = PAp_j$, while in ICCG we have $w_j = Ap_j$. This gives us

$$F_{PAp} = F_{PAx} = 11n + 16\sqrt{nk} - 16\sqrt{n} + 2k\sqrt{k} - 7k + 16\sqrt{k}$$

and

$$F_{Ap} = F_{Ax} = 9n.$$

C.3.3. *Summary.* The results of this subsection are summarized in Table 13.

Notation	**Operation**	**# Flops**
F_{prior}	Steps before the DICCG1 loop	$16\sqrt{nk} - 16\sqrt{n} +$ $k^2 + 8k\sqrt{k} - 7k + 16\sqrt{k}$
F_{after}	Steps after the DICCG1 loop	$6n + 4k\sqrt{k} - 6k +$ $16\sqrt{nk} - 16\sqrt{n} + 16\sqrt{k}$
$F_{prior-com}$	Common steps prior and after the ICCG and DICCG1 loops	$31n$
F_{com}	Common steps during the ICCG and DICCG1 loops	$27n$
F_{PAp}	Computing PAp in DICCG1	$11n + 16\sqrt{nk} - 16\sqrt{n} +$ $2k\sqrt{k} - 7k + 16\sqrt{k}$
F_{Ap}	Computing Ap in ICCG	$9n$

TABLE 13. Flops required to compute the steps in ICCG and DICCG1.

C.4. **Total Number of Flops for ICCG and DICCG1.** Now we can compute the number of flops required in ICCG and DICCG. We denote the number of required iterations of ICCG and DICCG1 with I_{ICCG} and I_{DICCG}. Then,

$$\begin{aligned} F_{ICCG} &= F_{prior-com} + I_{ICCG} \cdot (F_{com} + F_{Ap}) \\ &= 31n + I_{ICCG} \cdot (27n + 9n) \\ &= 31n + I_{ICCG} \cdot (36n) \end{aligned}$$

and

$$\begin{aligned} F_{DICCG} &= F_{prior} + F_{prior-com} + F_{after} + I_{DICCG} \cdot (F_{com} + F_{PAp}) \\ &= 16\sqrt{nk} - 16\sqrt{n} + k^2 + 8k\sqrt{k} - 7k + 16\sqrt{k} + 31n + \\ &\quad 6n + 4k\sqrt{k} - 6k + 16\sqrt{nk} - 16\sqrt{n} + 16\sqrt{k} + \\ &\quad I_{DICCG} \cdot (27n + 11n + 16\sqrt{nk} - 16\sqrt{n} + 2k\sqrt{k} - 7k + 16\sqrt{k}) \\ &= 37n + 32\sqrt{nk} - 32\sqrt{n} + k^2 + 12k\sqrt{k} - 13k + 32\sqrt{k} \\ &\quad I_{DICCG} \cdot (38n + 16\sqrt{nk} - 16\sqrt{n} + 2k\sqrt{k} - 7k + 16\sqrt{k}). \end{aligned}$$

To compare F_{ICCG} and F_{DICCG} in the numerical experiments, it is convenient to define

$$\xi_F := \frac{F_{DICCG}}{F_{ICCG}}.$$

If $\xi_F < 1$, then DICCG1 is more efficient than ICCG. For instance, if $\xi_F = 0.25$, then ICCG requires 4 times as much flops as DICCG. In the numerical experiments we will also compare ξ_F with ξ_{CPU} which is defined by

$$\xi_{CPU} := \frac{\text{CPU Time of DICCG}}{\text{CPU Time of ICCG}}.$$

The parameters ξ_F and ξ_{CPU} should resemble each other.

C.5. **Example: Flop Counts for Test Problem 2D-TP3 with $n = 100^2$.** We consider Test Problem 2D-TP3 which is a 2-D test problem with nine bubbles and grid sizes $n = 100^2$, see also Section 7. The results considering this test problem can be found in Table 14.

Method	# Iter.	CPU (sec)	ξ_{CPU}	# Flops ($\times 10^6$)	ξ_F
ICCG	247	0.26	–	89.2	–
DICCG1-5^2	70	0.11	0.42	28.2	0.32
DICCG1-10^2	44	0.08	0.31	18.2	0.21
DICCG1-20^2	27	0.07	0.27	12.1	0.14
DICCG1-25^2	23	0.06	0.23	11.5	0.13
DICCG1-50^2	14	0.11	0.42	18.1	0.20

TABLE 14. CPU time and flop counting results of ICCG and DICCG$-k$ for Test Problem 2D-TP3 (nine bubbles) with $N_x = N_y = 100$.

Considering both the CPU time and the ratio's ξ_F and ξ_{CPU}, we see that $k = 25^2$ is optimal and leads to the most efficient method in this test problem. Note that there are clear differences between ξ_{CPU} and ξ_F in absolute sense, since the CPU time does not only depend on the number of required flops, but also on for instance memory and other implementation aspects which are not considered in this paper.

Appendix D. Flop Counting of ICCG and DICCG2 in 3-D

The flop counts performed for the 2-D case can be generalized for the 3-D case, where we distinguish the two variants DICCG1 and DICCG2.

D.1. **Assumptions.** To compare the number of flops between DICCG1/DICCG2 and ICCG in the 3-D case, we assume that

- A has size $n \times n$ and consists of 7 non-zero diagonals;
- AZ is computed and stored efficiently using S_{AZ};
- E has size $k \times k$ and has bandwidth $k_x^2 + k_x = k^{2/3} + k^{1/3}$. E is computed and stored efficiently using S_E.

This leads to the assumptions as given in Table 15.

Notation	Operation	# Flops
$F_{(x,y)}$	(x,y)	$2n$
F_{x+y}	$x+y$	n
F_{Ax}	Ax	$13n$
$F_{chol(A)}$	Making C from A	$12n$
$F_{Ax=y}$	Solving x from $CC^Tx=y$	$15n$

Table 15. Assumptions of flop counts in standard operations of both ICCG and DICCG1/DICCG2 in 3-D.

D.1.1. *Computations S_{AZ} and S_E.* From Appendix B, the number of rows of S_{AZ} in the 3-D case is given by

$$\gamma = \gamma_c + \gamma_i + \gamma_b,$$

where

$$\left\{\begin{array}{lcl} \gamma_c & = & 8 \cdot (6n_b^2 - 3n_b + 1), \\ \gamma_i & = & (k_x - 2)^3 \cdot (12n_b^2 - 12n_b + 8), \\ \gamma_b & = & 12(k_x - 2) \cdot 8n_b^2 - 5n_b + 2 + 6(k_x - 2)^2 \cdot 10n_b^2 - 8n_b + 4 \end{array}\right.$$

and

$$n_b = \sqrt[3]{n/k}, \quad k_x = \sqrt[3]{k}.$$

This can be rewritten into

$$\left\{\begin{array}{lcl} \gamma_c & = & 8 \cdot (6n_b^2 - 3n_b + 1) \\ & = & 48n_b^2 - 24n_b + 8 \\ & = & 48(\sqrt[3]{n/k})^2 - 24(\sqrt[3]{n/k}) + 8 \\ & = & 48(n/k)^{2/3} - 24(n/k)^{1/3} + 8 \\ & \approx & 48(n/k)^{2/3}; \\ & & \\ \gamma_i & = & (k_x - 2)^3 \cdot (12n_b^2 - 12n_b + 8) \\ & = & ((\sqrt[3]{k}) - 2)^3 \cdot (12(\sqrt[3]{n/k})^2 - 12(\sqrt[3]{n/k}) + 8) \\ & = & (k^{1/3} - 2)(k^{2/3} - 4k^{1/3} + 4) \cdot (12(n/k)^{2/3} - 12(n/k)^{1/3} + 8) \\ & = & (k - 6k^{2/3} + 12k^{1/3} - 8) \cdot (12(n/k)^{2/3} - 12(n/k)^{1/3} + 8) \\ & = & 12n^{2/3}k^{1/3} - 12n^{1/3}k^{2/3} + 8k - 72n^{2/3} + 72(nk)^{1/3} - 48k^{2/3} \\ & & +144n^{2/3}k^{-1/3} - 144n^{1/3} + 96k^{1/3} - 96(n/k)^{2/3} + 96(n/k)^{1/3} - 64 \\ & \approx & 12n^{2/3}k^{1/3} + 8k; \\ & & \\ \gamma_b & = & 12(k_x - 2) \cdot 8n_b^2 - 5n_b + 2 + 6(k_x - 2)^2 \cdot 10n_b^2 - 8n_b + 4 \\ & = & 12((\sqrt[3]{k}) - 2) \cdot 8(\sqrt[3]{n/k})^2 - 5(\sqrt[3]{n/k}) + 2+ \\ & & 6((\sqrt[3]{k}) - 2)^2 \cdot 10(\sqrt[3]{n/k})^2 - 8(\sqrt[3]{n/k}) + 4; \\ & = & (12k^{1/3} - 24) \cdot 8(n/k)^{2/3} - 5(n/k)^{1/3} + 6+ \\ & & 60(k^{2/3} - 4k^{1/3} + 4)((n/k)^{2/3} - 8(n/k)^{1/3}) \\ & = & 96n^{2/3}k^{-1/3} - 192(n/k)^{2/3} - 5(n/k)^{1/3} + 6 + 60n^{2/3} - 480(nk)^{1/3} \\ & & -240n^{2/3}k^{-1/3} + 960n^{1/3}240(n/k)^{2/3} - 960(n/k)^{1/3} \\ & = & 60n^{2/3} + 48(n/k)^{2/3} - 144n^{2/3}k^{-1/3} - 480(nk)^{1/3}+ \\ & & 960n^{1/3} - 965(n/k)^{1/3} + 6 \\ & \approx & 60n^{2/3}. \end{array}\right.$$

Subsituting these approximations into γ yields

$$\gamma \approx 12n^{2/3}k^{1/3} + 60n^{2/3} + 48(n/k)^{2/3} + 8k.$$

Since again almost each element of S_{AZ} has exactly one contribution from A, the number of flops F_{AZ} and F_E to create AZ and E efficiently is approximately

$$\begin{aligned} F_E \approx F_{AZ} &\approx 12n^{2/3}k^{1/3} + 60n^{2/3} + 48(n/k)^{2/3} + 8k \\ &\approx 12n^{2/3}k^{1/3} = \mathcal{O}(n^{2/3}k^{1/3}). \end{aligned}$$

Note that the latter expression can also easily be obtained by observing that $\gamma_i \gg \gamma_b + \gamma_c$ for sufficiently large k and therefore the contributions of γ_b and γ_c could be omitted.

Obviously, in the 3-D case, the construction of S_{AZ} and S_E is more expensive ($\mathcal{O}(n^{2/3}k^{1/3})$) than in the 2-D case ($\mathcal{O}(n^{1/3})$).

D.1.2. *Computations $(AZ)x_2$, $Z^T x_2$, $(AZ)^T x_1$ en Zx_1,* We compute easily

$$\begin{aligned} F_{AZx_2} = F_{(AZ)^T x_1} &= 2\gamma \\ &\approx 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + 16k \\ &\approx 24n^{2/3}k^{1/3} = \mathcal{O}(n^{2/3}k^{1/3}). \end{aligned}$$

Moreover, the costs for F_{Zx_2} and $F_{Z^T x_1}$ are the same in 2-D and 3-D, i.e.,

$$F_{Zx_2} = F_{Z^T x_1} = n = \mathcal{O}(n).$$

Note that in 2-D constructing Zx_2 is more expensive than constructing AZx_2 ($\mathcal{O}(n) \gg \mathcal{O}(\sqrt{n})$), but in 3-D these costs are almost the same ($\mathcal{O}(n^{2/3}k^{1/3}) \approx \mathcal{O}(n)$).

D.1.3. *Computations for Solving $Ev = w$.* Solving $Ev = w$ is done differently in DICCG1 and DICGG2. We consider both cases below.

- *DICCG1: Solving $Ev = w$ directly.* First the band-Cholesky decomposition L is constructed from E and thereafter v is solved from $LL^T v = w$. Since in 3-D the bandwidth of E is $k^{2/3} + k^{1/3}$ instead of $k^{1/2}$, we obtain (cf. [13, Appendix B])

$$\begin{aligned} F_{chol(E)} &= k((k^{2/3} + k^{1/3})^2 + 3(k^{2/3} + k^{1/3})) + k \\ &= k(k^{4/3} + 2k + 4k^{2/3} + 3k^{1/3}) + k \\ &= k^{7/3} + 2k^2 + 4k^{5/3} + 3k^{4/3} + k \\ &\approx \mathcal{O}(k^{7/3}) \end{aligned}$$

  and

$$\begin{aligned} F_{Ev=w,DICCG1} &= k(2(k^{2/3} + k^{1/3}) + 1) \\ &= 2k^{5/3} + 2k^{4/3} + k \\ &\approx \mathcal{O}(k^{5/3}). \end{aligned}$$

  Obviously, solving $Ev = w$ directly in 3-D is less attractive than in the 2-D case.
- *DICCG2: Solving $Ev = w$ iteratively.* The system $Ev = w$ is solved using ICCG. These inner iterations are denoted by ICCG2. In the previous appendix we have seen that

$$\begin{aligned} F_{Ev=w,DICCG2} &= F_{prior-com} + I_{ICCG2} \cdot (F_{com} + F_{Ap}) \\ &= 31k + I_{ICCG2} \cdot (36k) \end{aligned}$$

  by replacing n by k.

Clearly, it depends on k and on I_{ICCG2} whether DICCG1 or DICCG2 is the most efficient method.

D.1.4. *Computations Px and PAx.* To compute Px we assume again that S_{AZ} is known and in DICCG1 also $chol(E)$ should be known. Obviously, F_{Px} and F_{PAx} depend on the choice of DICCG1 and DICCG2. We obtain

$$\begin{aligned} F_{Px,DICCG1} &= F_{Z^Tx_1} + F_{Ev=w,DICCG1} + F_{AZx_2} + F_{x-x_3} \\ &= n + 2k^{5/3} + 2k^{4/3} + k + 24n^{2/3}k^{1/3} + \\ &\quad 120n^{2/3} + 96(n/k)^{2/3} + 16k + n \\ &= 2n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} \\ &\quad +2k^{5/3} + 2k^{4/3} + 17k \\ &= \mathcal{O}(n), \end{aligned}$$

and

$$\begin{aligned} F_{Px,DICCG2} &= F_{Z^Tx_1} + F_{Ev=w,DICCG2} + F_{AZx_2} + F_{x-x_3} \\ &= n + 31k + I_{ICCG2}\cdot(36k) + 24n^{2/3}k^{1/3} + \\ &\quad 120n^{2/3} + 96(n/k)^{2/3} + 16k + n \\ &= 2n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 47k + \\ &\quad I_{ICCG2}\cdot(36k) + 96(n/k)^{2/3} \\ &= \mathcal{O}(n). \end{aligned}$$

Moreover, we obtain immediately

$$\begin{aligned} F_{PAx,DICCG1} &= F_{Px,DICCG1} + F_{Ax} \\ &= 15n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} \\ &\quad +2k^{5/3} + 2k^{4/3} + 17k \\ &= \mathcal{O}(n), \end{aligned}$$

and

$$\begin{aligned} F_{PAx,DICCG2} &= F_{Px,DICCG2} + F_{Ax} \\ &= 15n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 47k \\ &\quad +I_{ICCG2}\cdot(36k) + 96(n/k)^{2/3} \\ &= \mathcal{O}(n). \end{aligned}$$

D.1.5. *Summary.* In Table 16, we summarize the flop counting results of this subsection.

Notation	**Operation**	**# Flops**
F_{AZ}	S_{AZ}	$12n^{2/3}k^{1/3} + 60n^{2/3} + 48(n/k)^{2/3} + 8k$
F_E	S_E	$12n^{2/3}k^{1/3} + 60n^{2/3} + 48(n/k)^{2/3} + 8k$
F_{AZx_2}	AZx_2	$24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + 16k$
$F_{(AZ)^Tx_1}$	$(AZ)^Tx_1$	$24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + 16k$
F_{Zx_2}	Zx_2	$2n$
$F_{Z^Tx_1}$	Z^Tx_1	$2n$
$F_{chol(E)}$	$chol(E)$	$k^{7/3} + 2k^2 + 4k^{5/3} + 3k^{4/3} + k$
$F_{Ev=w,DICCG1}$	$Ev = w$	$2k^{5/3} + 2k^{4/3} + k$
$F_{Ev=w,DICCG2}$	$Ev = w$	$31k + I_{ICCG}\cdot(36k)$
$F_{Px,DICCG1}$	Px	$2n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + 2k^{5/3} + 2k^{4/3} + 17k$
$F_{Px,DICCG2}$	Px	$2n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 47k + I_{ICCG2}\cdot(36k) + 96(n/k)^{2/3}$
$F_{PAx,DICCG1}$	PAx	$15n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + 2k^{5/3} + 2k^{4/3} + 17k$
$F_{PAx,DICCG2}$	PAx	$15n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 47k + I_{ICCG2}\cdot(36k) + 96(n/k)^{2/3}$

TABLE 16. Flops required to compute the main steps in DICCG1 and DICCG2 in the 3-D case.

D.2. **Flop Counts in ICCG and DICCG1/DICCG2.** We divide this subsection into two parts. The first part computes the number of flops required prior to and after the ICCG and DICCG1/DICCG2 loops. The second part determines the number of flops required in both loops.

D.2.1. *Computations before and after the ICCG and DICCG1/DICCG2 loops.* The extra computations required in DICCG1 and DICCG2 are determined below.

For $F_{prior,DICCG1}$ and $F_{prior,DICCG2}$, we obtain

$$\begin{array}{rcl} F_{prior,DICCG1} & = & F_{AZ} + F_E + F_{chol(E)} \\ & = & 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + 16k \\ & & +k^{7/3} + 2k^2 + 4k^{5/3} + 3k^{4/3} + k \\ & = & 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + k^{7/3} + \\ & & 2k^2 + 4k^{5/3} + 3k^{4/3} + 17k \\ & = & \mathcal{O}(n^{2/3}k^{1/3}), \end{array}$$

and

$$\begin{array}{rcl} F_{prior,DICCG2} & = & F_{AZ} + F_E \\ & = & 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + 16k \\ & = & \mathcal{O}(n^{2/3}k^{1/3}). \end{array}$$

To compute $F_{after,DICCG1}$ and $F_{after,DICCG2}$, we have to determine the number of flops $F_{u,DICCG1}$ and $F_{u,DICCG2}$ and also $F_{x,DICCG1}$ and $F_{x,DICCG2}$:

$$\begin{array}{rcl} F_{u,DICCG1} & = & F_{Z^T x_1} + F_{Ev=w,DICCG1} + F_{Zx_2} \\ & = & 2n + 2k^{5/3} + 2k^{4/3} + k \\ & = & \mathcal{O}(n), \end{array}$$

$$\begin{array}{rcl} F_{u,DICCG2} & = & F_{Z^T x_1} + F_{Ev=w,DICCG2} + F_{Zx_2} \\ & = & 2n + 31k + I_{ICCG2} \cdot (36k) \\ & = & \mathcal{O}(n), \end{array}$$

and

$$\begin{array}{rcl} F_{x,DICCG1} & = & n + F_{Px,DICCG1} = \\ & = & 3n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} \\ & & +2k^{5/3} + 2k^{4/3} + 17k \\ & = & \mathcal{O}(n), \end{array}$$

$$\begin{array}{rcl} F_{x,DICCG2} & = & n + F_{Px,DICCG2} = \\ & = & 16n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 47k \\ & & +I_{ICCG2} \cdot (36k) + 96(n/k)^{2/3} \\ & = & \mathcal{O}(n). \end{array}$$

Hence, this implies

$$\begin{array}{rcl} F_{after,DICCG1} & = & F_{u,DICCG1} + F_{x,DICCG1} \\ & = & 2n + 2k^{5/3} + 2k^{4/3} + k + 2n + 24n^{2/3}k^{1/3} + 120n^{2/3} \\ & & +96(n/k)^{2/3} + 2k^{5/3} + 2k^{4/3} + 17k \\ & = & 4n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + \\ & & 4k^{5/3} + 4k^{4/3} + 18k \\ & = & \mathcal{O}(n), \end{array}$$

$$\begin{array}{rcl} F_{after,DICCG2} & = & F_{u,DICCG2} + F_{x,DICCG2} \\ & = & 2n + 31k + I_{ICCG2} \cdot (36k) + 16n + 24n^{2/3}k^{1/3} \\ & & +120n^{2/3} + 47k + I_{ICCG2} \cdot (36k) + 96(n/k)^{2/3} \\ & = & 18n + 58k + 24n^{2/3}k^{1/3} + 96(n/k)^{2/3} + \\ & & 120n^{2/3} + I_{ICCG2} \cdot (72k) \\ & = & \mathcal{O}(n). \end{array}$$

Next, similar to 2-D, ICCG and DICCG1/DICCG2 have several common computations which gives

$$F_{prior-com} = 39n.$$

D.2.2. *Computations during the ICCG and DICCG1/DICCG2 loops.* In the iterates, ICCG and DICCG1/DICCG2 have also common operations which results in

$$F_{com} = 31n.$$

Moreover, the difference between ICCG and DICCG1/DICCG2 in the iterates is computing w_j. In DICCG1 and DICCG2, we compute $w_j = PAp_j$ while in ICCG we have $w_j = Ap_j$. This gives

$$\begin{aligned} F_{PAp,DICCG1} &= F_{PAx,DICCG1} \\ &= 15n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + \\ &\quad 2k^{5/3} + 2k^{4/3} + 17k \\ &= \mathcal{O}(n), \end{aligned}$$

$$\begin{aligned} F_{PAp,DICCG2} &= F_{PAx,DICCG2} \\ &= 15n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 47k \\ &\quad + I_{ICCG2} \cdot (36k) + 96(n/k)^{2/3} \\ &= \mathcal{O}(n) \end{aligned}$$

and

$$F_{Ap,DICCG1} = F_{Ax,DICCG1} = 13n = \mathcal{O}(n).$$

D.2.3. *Summary.* The results of this subsection are summarized in Table 17.

Notation	**Operation**	**# Flops**
$F_{prior,DICCG1}$	Steps before DICCG1 loop	$24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + k^{7/3} + 2k^2 + 4k^{5/3} + 3k^{4/3} + 17k$
$F_{prior,DICCG2}$	Steps before DICCG2 loop	$24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + 16k$
$F_{after,DICCG1}$	Steps after DICCG1 loop	$4n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + 4k^{5/3} + 4k^{4/3} + 18k$
$F_{after,DICCG2}$	Steps after DICCG2 loop	$18n + 58k + 24n^{2/3}k^{1/3} + 96(n/k)^{2/3} + 120n^{2/3} + I_{ICCG2} \cdot (72k)$
$F_{prior-com}$	Common steps prior and after ICCG and DICCG1/2 loops	$39n$
F_{com}	Common steps during ICCG and DICCG1/2 loops	$31n$
$F_{PAp,DICCG1}$	Computing PAp in DICCG1	$15n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} + 2k^{5/3} + 2k^{4/3} + 17k$
$F_{PAp,DICCG2}$	Computing PAp in DICCG2	$15n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 47k + I_{ICCG2} \cdot (36k) + 96(n/k)^{2/3}$
F_{Ap}	Computing Ap in ICCG	$13n$

TABLE 17. Flops required to compute the steps in ICCG and DICCG1/DICCG2.

D.3. **Flops for ICCG and DICCG1/DICCG2.** Now we can easily compute the number of flops required in ICCG and DICCG1 / DICCG2, namely

$$\begin{aligned} F_{ICCG} &= F_{prior-com} + I_{ICCG} \cdot (F_{com} + F_{Ap}) \\ &= 39n + I_{ICCG} \cdot (31n + 13n) \\ &= 39n + I_{ICCG} \cdot (44n), \end{aligned}$$

and

$$\begin{aligned}
F_{DICCG1} &= F_{prior,DICCG1} + F_{prior-com} + F_{after,DICCG1} + \\
&\quad I_{DICCG1} \cdot (F_{com} + F_{PAp,DICCG1}) \\
&= 43n + 47n^{2/3}k^{1/3} + 240n^{2/3} + 192(n/k)^{2/3} + \\
&\quad k^{7/3} + 2k^2 + 8k^{5/3} + 7k^{4/3} + 25k + \\
&\quad I_{DICCG1} \cdot (46n + 24n^{2/3}k^{1/3} + 120n^{2/3} + 96(n/k)^{2/3} \\
&\quad + 2k^{5/3} + 2k^{4/3} + 17k),
\end{aligned}$$

$$\begin{aligned}
F_{DICCG2} &= F_{prior,DICCG2} + F_{prior-com} + F_{after,DICCG2} + \\
&\quad I_{DICCG2} \cdot (F_{com} + F_{PAp,DICCG2}) \\
&= 57n + 58k + 48n^{2/3}k^{1/3} + 240n^{2/3} + 192(n/k)^{2/3} + 16k + \\
&\quad I_{ICCG2} \cdot (72k) + I_{DICCG2} \cdot (46n + 24n^{2/3}k^{1/3} \\
&\quad + 120n^{2/3} + 47k + I_{ICCG2} \cdot (36k) + 96(n/k)^{2/3}).
\end{aligned}$$

Next, to compare F_{ICCG} and F_{DICCG1} or F_{DICCG2} in the numerical experiments, we consider

$$\xi_{F1} := \frac{F_{DICCG1}}{F_{ICCG}} \quad \xi_{CPU1} := \frac{\text{CPU Time of DICCG1}}{\text{CPU Time of ICCG}},$$

and

$$\xi_{F2} := \frac{F_{DICCG2}}{F_{ICCG}} \quad \xi_{CPU2} := \frac{\text{CPU Time of DICCG2}}{\text{CPU Time of ICCG}}.$$

D.4. **Example: Flop Counts for Test Problem 3D-TP4 with $n = 100^3$.** We consider Test Problem 3D-TP4, which is a 3-D test problem with 27 bubbles and grid sizes $n = 100^3$, see also Section 8. The results considering this test problem can be found in Table 18.

		DICCG1		DICCG2		DICCG1		DICCG2	
Method	# It.	CPU	ξ_{CPU1}	CPU	ξ_{CPU2}	# Fl.	ξ_{F1}	# Fl.	ξ_{F2}
ICCG	310	46.0	–	46.0	–	13.7	–	13.7	–
DICCG1/2-2^3	275 (5)	50.4	1.1	48.8	1.1	13.5	1.0	13.5	1.0
DICCG1/2-5^3	97 (16)	19.0	0.4	18.7	0.4	4.8	0.4	4.8	0.4
DICCG1/2-10^3	60 (43)	13.0	0.3	13.0	0.3	3.1	0.2	3.2	0.2
DICCG1/2-20^3	31 (115)	29.3	0.6	11.0	0.2	3.5	0.3	2.8	0.2
DICCG1/2-25^3	26 (146)	>300	>6.0	14.2	0.3	9.2	0.7	3.8	0.3

TABLE 18. Iterations (in brackets the required inner iterations in DICCG2), CPU time (in seconds) and flop counting ($\times 10^9$) results of ICCG, DICCG1$-k$ and DICCG2$-k$ for Test Problem 3D-TP4 (27 bubbles) with $N_x = N_y = N_z = 100$.

Although there are differences between ξ_{F1} (ξ_{F2}) and ξ_{CPU1} (ξ_{CPU2}), the tendencies of these two parameters resemble each other. Considering these parameters, DICCG1 is optimal around $k = 10^3$ and DICCG2 is optimal around $k = 10^3 - 25^3$.