Fast solvers for Eulerian convection schemes

Andreas Hiltebrand

Curriculum Computational Science and Engineering

Semester Thesis FS 2010

Seminar for Applied Mathematics
ETH Zürich

Supervisor: Holger Heumann

Professor: Ralf Hiptmair

Contents

1 Introduction

1.1 Subject and motivation

Advection-diffusion problems with a dominating diffusion can be handled well by standard finite elements. But in case of small or even vanishing diffusion special schemes are needed. We use here discontinuous Galerkin methods with penalty terms and upwinding. The solution of the corresponding linear system can then be speeded up by methods tailored to these problems.

The goal of this work is to implement and test a certain idea: Advection dominated problems introduce a flow of information. By ordering the degrees of freedom accordingly one can utilize this and construct a fast solver. More exactly the sorted system is (almost) lower block triangular. This implies that a block Gauss-Seidel method works well.

1.2 Outline of the thesis

Chapter 2 introduces the theoretical background for this work. First the main idea of ordering the degrees of freedom and then solving the easier block triangular system is illustrated. The rest of the chapter deals with the construction of orderings, especially how to reduce the size of blocks in the sorted linear system.

The goal of chapter 3 is to show the main structure of the code. In particular the classes and their most important functions are discussed. First the construction of orderings is in focus and then the preconditioners that approximatively solve the system. The last point is a discussion of the complexity of the algorithms.

In chapter 4 the test cases are defined and the results are discussed. The advection-diffusion equation and a generalized convection-diffusion problem are studied (in 3D). The performance of different preconditioners are compared and their dependency on the diffusion coefficient is investigated.

The last chapter 5 summarizes the work and draws the conclusions.

2 Ordering of the degrees of freedom

In order to solve a linear differential equation numerically a discretization has to be applied. It leads to a linear system

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}\,, \tag{2.1}$$

where $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is the system matrix, $\boldsymbol{b} \in \mathbb{R}^n$ is the right-hand side and $\boldsymbol{u} \in \mathbb{R}^n$ is the solution vector with n degrees of freedom.

We investigate pure advection and advection dominated problems. The discretization is done using discontinuous Galerkin methods with penalty terms and upwinding (see [1]). The goal of this work is to solve the resulting linear systems efficiently. We therefore introduce an ordering of the degrees of freedom that corresponds to the flow of information in the problem and solve the sorted system. The numerical solvers often depends on the ordering of the degrees of freedom; using a good ordering can significantly improve their performance.

An ordering of the degrees of freedom π is a permutation of the indices of the degrees of freedom $\{1, \ldots, n\}$. It maps the index in the sorted system to the index in the original system.

The corresponding permutation matrix $\boldsymbol{P} \in \mathbb{R}^{n\times n}$ has the entries

$$P_{i,j} = \begin{cases} 1, & i = \pi(j) \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

The sorted matrix is then $\widetilde{\boldsymbol{A}} = \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}$, the sorted right-hand side is $\widetilde{\boldsymbol{b}} = \boldsymbol{P}\boldsymbol{b}$ and the sorted solution vector is $\widetilde{\boldsymbol{u}} = \boldsymbol{P}\boldsymbol{u}$. Out of the original linear system we get

$$\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{\mathrm{T}}\boldsymbol{P}\boldsymbol{u} = \boldsymbol{P}\boldsymbol{b} \tag{2.3}$$

by multiplication of $\boldsymbol{P}$ from the left side and using the property of a permutation matrix $\boldsymbol{P}^{\mathrm{T}}\boldsymbol{P} = \boldsymbol{I}$, where $\boldsymbol{I}$ is the identity matrix. Then we use these definitions to obtain the equivalent system

$$\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{u}} = \widetilde{\boldsymbol{b}}\,. \tag{2.4}$$

The various steps for solving the original system are summarized in algorithm 1.

Note that the permutation matrix $\boldsymbol{P}$ does not need to be constructed explicitly. It would be more efficient to sort the rows and columns directly according to π. We also do not really want to sort the matrix $\boldsymbol{A}$ for efficiency reasons.[1] Therefore this algorithm is avoided and our preconditioners work directly on the original matrix $\boldsymbol{A}$. But it is much easier to develop algorithms based on the

[1] One to could try to construct the ordering without assembling the original matrix and then directly assemble the sorted matrix.

Algorithm 1: Solution of linear system by sorting it

input : matrix $\boldsymbol{A}$, right-hand side $\boldsymbol{b}$
output: $\boldsymbol{u}$ solution to $\boldsymbol{Au} = \boldsymbol{b}$
construct ordering π, the corresponding permutation matrix is $\boldsymbol{P}$
sort matrix $\widetilde{\boldsymbol{A}} = \boldsymbol{PAP}^{\mathrm{T}}$
sort right-hand side $\widetilde{\boldsymbol{b}} = \boldsymbol{P}b$
solve $\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{u}} = \widetilde{\boldsymbol{b}}$
invert sorting of solution $\boldsymbol{u} = \boldsymbol{P}^{\mathrm{T}}\widetilde{\boldsymbol{u}}$

sorted matrix, which is the reason why this approach is presented here. Working on the original matrix then just adds some index transformations on top. However, from the cache efficiency point of view, this is not optimal too, because it leads to many jumps within the memory.

2.1 Solution of the sorted system

To justify the additional effort in sorting the matrix, the sorted system should be much easier to invert. For pure advection problems the sorted matrix $\widetilde{\boldsymbol{A}}$ will typically be lower block triangular, for a suitable definition of the ordering of the degrees of freedom. Hence, it is easy to invert the system, i.e. a quick numerical solver can be constructed. The original system could be solved in the same way, but it is not as apparent as for the lower block triangular system that this is possible.

If $\widetilde{\boldsymbol{A}}$ is lower block triangular we can write

$$\widetilde{\boldsymbol{A}} = \begin{pmatrix} \boldsymbol{D}_1^B & \boldsymbol{0} & \cdots & \cdots & \boldsymbol{0} \\ \boldsymbol{L}_{2,1}^B & \boldsymbol{D}_2^B & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \boldsymbol{D}_{n_B-1}^B & \boldsymbol{0} \\ \boldsymbol{L}_{n_B,1}^B & \cdots & \cdots & \boldsymbol{L}_{n_B,n_B-1}^B & \boldsymbol{D}_{n_B}^B \end{pmatrix}, \tag{2.5}$$

where $\boldsymbol{D}_i^B \in \mathbb{R}^{n_i \times n_i}$ for $i = 1, \ldots, n_B$ are the blocks on the diagonal with size n_i and $\boldsymbol{L}_{i,j}^B \in \mathbb{R}^{n_i \times n_j}$ for $1 \leq j < i \leq n_B$ are the blocks on the lower part.

The right-hand side and the solution vector are partitioned accordingly

$$\widetilde{\boldsymbol{b}} = \begin{pmatrix} \boldsymbol{b}_1^B \\ \vdots \\ \boldsymbol{b}_{n_B}^B \end{pmatrix}, \quad \widetilde{\boldsymbol{u}} = \begin{pmatrix} \boldsymbol{u}_1^B \\ \vdots \\ \boldsymbol{u}_{n_B}^B \end{pmatrix} \tag{2.6}$$

with $\boldsymbol{b}_i^B, \boldsymbol{u}_i^B \in \mathbb{R}^{n_i}$.

The linear system is then equivalent to

$$\sum_{j=1}^{i-1} \boldsymbol{L}_{i,j}^B \boldsymbol{u}_j^B + \boldsymbol{D}_i^B \boldsymbol{u}_i^B = \boldsymbol{b}_i^B, \quad i = 1, \ldots, n_B \tag{2.7}$$

Note that each $\boldsymbol{u}_i^B$ only depends on $\boldsymbol{u}_j^B$ with $j < i$. This implies that we can solve for $\boldsymbol{u}_1^B$, then for $\boldsymbol{u}_2^B$ and so on. We summarize this in algorithm 2.

Algorithm 2: Solution of block triangular system

input : block triangular matrix $\widetilde{\boldsymbol{A}}$, right-hand side $\widetilde{\boldsymbol{b}}$
output: $\widetilde{\boldsymbol{u}}$ solution to $\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{u}} = \widetilde{\boldsymbol{b}}$
$\widetilde{\boldsymbol{A}}$, $\widetilde{\boldsymbol{b}}$, $\widetilde{\boldsymbol{u}}$ *are partitioned as described above*
for $i = 1, \ldots, n_B$ **do**
    $\boldsymbol{u}_i^B = \left(\boldsymbol{D}_i^B\right)^{-1} \left(\boldsymbol{b}_i^B - \sum_{j=1}^{i-1} \boldsymbol{L}_{i,j}^B \boldsymbol{u}_j^B\right)$
end

The solution of the whole system splits into the solution of systems corresponding to each diagonal block $\boldsymbol{D}_i^B$. The smaller the blocks the easier it is to solve each block and also to solve the whole system. So the first goal of our ordering is to generate blocks that are as small as possible.

In order to compute the solution of the block-wise systems one can use direct or iterative methods. In section 3.5 an implementation is presented that uses both methodologies. If iterative methods are used, the solution will typically be less exact and therefore errors will accumulate. But since the system is in general not block triangular, there will be a few iterations over the whole system anyway, so the error will decrease more and more.

That the small systems are invertible whenever $\boldsymbol{A}$ is invertible can be seen from

$$\det \boldsymbol{A} = \det(\boldsymbol{P}^{\mathrm{T}} \widetilde{\boldsymbol{A}} \boldsymbol{P}) = \det(\boldsymbol{P}^{\mathrm{T}}) \det(\widetilde{\boldsymbol{A}}) \det(\boldsymbol{P}) = \underbrace{\det(\boldsymbol{P}^{\mathrm{T}} \boldsymbol{P})}_{\det(\boldsymbol{I})=1} \cdot \prod_{i=1}^{n_B} \det(\boldsymbol{D}_i^B)\,. \tag{2.8}$$

The systems for advection dominated problems will not really be block triangular (we consider just one big block as not block triangular). But it is typically possible to find an ordering such that the part above the diagonal is small.

For advection dominated problem this can be done in the following way: We split the matrix

$$\boldsymbol{A} = \boldsymbol{A}_{\mathrm{adv}} + \boldsymbol{A}_{\mathrm{diff}} \tag{2.9}$$

into $\boldsymbol{A}_{\mathrm{adv}}$ corresponding to the advection operator and $\boldsymbol{A}_{\mathrm{diff}}$ corresponding to the diffusion operator. $\boldsymbol{A}_{\mathrm{diff}}$ is assumed to be small. (The inflow boundary conditions are included in the advection part; the remaining boundary conditions are accounted for in the diffusion part.) To construct the ordering we use only the advection part, which can be brought in lower block

triangular form. The diffusion part is typically symmetric and hence no matter how we order the degrees of freedom the corresponding matrix will not be block triangular; thus it leads to some non-zero entries in the upper part, but they are small if the assumption is true.

We partition the matrix $\widetilde{\boldsymbol{A}}$ as follows

$$\widetilde{\boldsymbol{A}} = \begin{pmatrix} \boldsymbol{D}_1^B & \boldsymbol{U}_{1,2}^B & \cdots & \cdots & \boldsymbol{U}_{1,n_B}^B \\ \boldsymbol{L}_{2,1}^B & \boldsymbol{D}_2^B & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \boldsymbol{D}_{n_B-1}^B & \boldsymbol{U}_{n_B-1,n_B}^B \\ \boldsymbol{L}_{n_B,1}^B & \cdots & \cdots & \boldsymbol{L}_{n_B,n_B-1}^B & \boldsymbol{D}_{n_B}^B \end{pmatrix}, \tag{2.10}$$

where $\boldsymbol{D}_i^B \in \mathbb{R}^{n_i \times n_i}$ for $i = 1, \ldots, n_B$ are the blocks on the diagonal with size n_i and $\boldsymbol{L}_{i,j}^B \in \mathbb{R}^{n_i \times n_j}$ for $1 \leq j < i \leq n_B$ are the blocks on the lower part as before. The new parts are the blocks on the upper part: $\boldsymbol{U}_{i,j}^B \in \mathbb{R}^{n_i \times n_j}$ for $1 \leq i < j \leq n_B$. Note that for a given matrix $\widetilde{\boldsymbol{A}}$ there is in general no unique partitioning, unless the sizes of the blocks n_i are specified. The blocks $\boldsymbol{U}_{i,j}^B$ should contain only small elements and as few non-zero elements as possible. The construction of the ordering should take care of this.

We partition the right-hand side and the solution accordingly (as before). The linear system is equivalent to

$$\sum_{j=1}^{i-1} \boldsymbol{L}_{i,j}^B \boldsymbol{u}_j^B + \boldsymbol{D}_i^B \boldsymbol{u}_i^B + \sum_{j=i+1}^{n_B} \boldsymbol{U}_{i,j}^B \boldsymbol{u}_j^B = \boldsymbol{b}_i^B, \quad i = 1, \ldots, n_B \,. \tag{2.11}$$

Assuming the entries in $\boldsymbol{U}_{i,j}^B$ to be small, it is natural to propose an algorithm similar to algorithm 2.

Algorithm 3: Solution of almost block triangular system

input : almost block triangular matrix $\widetilde{\boldsymbol{A}}$, right-hand side $\widetilde{\boldsymbol{b}}$, initial guess $\widetilde{\boldsymbol{u}}$
output: $\widetilde{\boldsymbol{u}}$ improved solution to $\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{u}} = \widetilde{\boldsymbol{b}}$
$\widetilde{\boldsymbol{A}}$, $\widetilde{\boldsymbol{b}}$, $\widetilde{\boldsymbol{u}}$ *are partitioned as described above*
for $i = 1, \ldots, n_B$ **do**
    $\boldsymbol{u}_i^B = \left(\boldsymbol{D}_i^B\right)^{-1} \left(\boldsymbol{b}_i^B - \sum_{j=1}^{i-1} \boldsymbol{L}_{i,j}^B \boldsymbol{u}_j^B - \sum_{j=i+1}^{n_B} \boldsymbol{U}_{i,j}^B \boldsymbol{u}_j^B\right)$
end

The algorithm can also be iterated a few times to improve the solution. Note the following: If we define $\boldsymbol{L}$ as the strictly lower block triangular part, $\boldsymbol{D}$ as the block diagonal part and $\boldsymbol{U}$ as the strictly upper block triangular part, then algorithm 3 is equivalent to

$$\widetilde{\boldsymbol{u}} = (\boldsymbol{L} + \boldsymbol{D})^{-1} (\boldsymbol{b} - \boldsymbol{U}\widetilde{\boldsymbol{u}}) \,, \tag{2.12}$$

i.e. a block Gauss-Seidel method.

The necessary and sufficient condition that the error always vanishes if the number of iterations is increased, is, as usual, that the spectral radius ρ of the iteration matrix $-(\boldsymbol{L}+\boldsymbol{D})^{-1}\boldsymbol{U}$ satisfies

$$\rho\Big(-(\boldsymbol{L}+\boldsymbol{D})^{-1}\boldsymbol{U}\Big) < 1\,. \tag{2.13}$$

Because it can be estimated by

$$\rho\Big(-(\boldsymbol{L}+\boldsymbol{D})^{-1}\boldsymbol{U}\Big) \leq \Big\|(\boldsymbol{L}+\boldsymbol{D})^{-1}\Big\|\,\|\boldsymbol{U}\| \tag{2.14}$$

using a sub-multiplicative matrix norm $\|\cdot\|$, the solution of the algorithm converges at least if the upper part $\boldsymbol{U}$ is small enough.

2.2 Construction of orderings

In order to construct a suitable ordering of the degrees of freedom let us have a look at the matrix graph.

To motivate the matrix graph consider the i-th equation of a linear system with the matrix $\boldsymbol{A}$:

$$\sum_{k=1}^{n} A_{i,k}u_k = b_i\,. \tag{2.15}$$

If the diagonal element is non-zero $A_{i,i} \neq 0$, we can solve for u_i and get

$$u_i = \frac{1}{A_{i,i}}\left(b_i - \sum_{\substack{k=1\\k\neq i}}^{n} A_{i,k}u_k\right) = \frac{1}{A_{i,i}}\left(b_i - \sum_{\substack{k=1\\k\neq i\\A_{i,k}\neq 0}}^{n} A_{i,k}u_k\right)\,. \tag{2.16}$$

u_i depends on all u_k with k such that $A_{i,k} \neq 0$ and $k \neq i$. This dependency is captured in the matrix graph: The matrix graph $G = (V, E)$ of a matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is a directed graph and composed of the following: The nodes $V = \{1, \ldots, n\}$ are the indices of the degrees of freedom and the edges are

$$E = \{(i,j) \in V \times V : i \neq j, |A_{j,i}| > tol\}\,. \tag{2.17}$$

In principle we would like to use $tol = 0$, but because of rounding errors in the matrix a small tolerance is useful.

An ordering of the degrees of freedom can be seen as a numbering of the vertices of the matrix graph. This is true because the nodes correspond the degrees of freedom. We would like to find an ordering π which is consistent with the edges in the following way:

$$(i,j) \in E \Rightarrow \pi(i) < \pi(j) \quad \forall i,j \in V\,. \tag{2.18}$$

Then we could solve successively for $u_{\pi(1)}$, $u_{\pi(2)}$,...,$u_{\pi(n)}$ and nothing would (strongly) depend on anything that is not already available. Put into the framework of the last section: The sorted

matrix $\widetilde{\boldsymbol{A}}$ will then be lower triangular (up to some small entries, if $tol>0$) and we see that the linear system will be (approximately) solvable by a forward substitution.

If the graph G is acyclic, this is indeed possible. In section 2.3 we give an algorithm to construct a suitable ordering for this setting.

Unfortunately, the graphs G for our problems are not acyclic (for a small choice of tol, which we assume in the following). But with some modifications we can get to a related graph that is acyclic at least for simple cases.

We notice that for discontinuous finite elements each degree of freedom is assigned to a certain element and that the degrees of freedom of a certain element typically all depend on each other, i.e. they form cycles. Therefore, we condensate the graph G into an element-wise graph $G_{elem} = (V_{elem}, E_{elem})$. The degrees of freedom of one element form a new vertex, i.e. each element index is a node: $V_{elem} = \{1, ..., n_{elem}\}$, where n_{elem} is the number of elements. Two vertices of the condensation graph are connected with an edge if there is any edge between degrees of freedom of the elements represented by these vertices. We denote by m_{dof}^{elem} the mapping

$$m_{dof}^{elem} : \{1, ..., n\} \to \{1, ..., n_{elem}\} \tag{2.19}$$

that assigns each index of a degree of freedom the index of the element the degree of freedom is assigned to. Then the edge set of the condensation graph is

$$E_{elem} = \left\{ (i,j) \in V_{elem} \times V_{elem} : i \neq j, \exists (k,l) \in E : i = m_{dof}^{elem}(k), j = m_{dof}^{elem}(l) \right\}. \tag{2.20}$$

For simple pure advection problems this graph will be acyclic. We can then construct a consistent ordering for the elements π_{elem} as above. This implies that we can solve for all the degrees of freedom of the element $\pi_{elem}(1)$, then for those of the element $\pi_{elem}(2)$ and so on. Their will be no (strong) dependencies between degrees of freedom of different elements that are broken. In terms of the last section: the sorted matrix will be (almost) lower block triangular and each block corresponds to one element.

2.3 Topological sorting

The construction of consistent orderings for acyclic graphs is typically called topological sorting or ordering. There are different algorithms, we refer here to [2] (Hackbusch calls it downwind numbering, but it is essentially the same). Note that except for simple cases the ordering is not unique. Different algorithms will produce different orderings; the outcome of the algorithms might even depend on the primal ordering of the vertices.

According to Hackbusch, the basic procedure to construct a consistent ordering works as follows: Each node v gets an attribute $attr(v) \in \{F, C, L\}$. F stand for first and indicates that all predecessors of this node are already included in the ordering with a low index and that they also have the flag F. Similarly, if a node is assigned L (last) then all successors of this node have been included in the ordering with a high index and they have been assigned L. In the beginning

each node gets C. If it is not in a cycle or between two cycles, it will get either an F or an L in the running of the algorithm. Those that keep the C-flag, are finally inserted in the middle of the ordering π in the original order.

Algorithm 4: Simple ordering

input : graph $G = (V, E)$
output: ordering π
for $v \in V$ **do** $attr(v) = C$
for $v \in V$ **do** SetAttr(v)
for $v \in V$ **do**
  if $attr(v) = C$ **then** $\pi(first) = v$
end

$first$ stands for the first entry in π that is not yet assigned.

Procedure SetAttr(v)

if $attr(v) = C$ **then** SetF(v);
if $attr(v) = C$ **then** SetL(v);

SetAttr tries to assign the flag F to the node v and its direct and indirect successors unless v has already got an attribute F or L. Then it tries to assign L to it.

Procedure SetF(v)

if $\forall w \in pred(v) : attr(w) = F$ **then**
  $attr(v) = F$;
  $\pi(first) = v$;
  for $w \in succ(v)$ **do if** $attr(w) = C$ **then** SetF(w)
end

Only if all the predecessors of v $pred(v)$ already have the attribute F, also v gets F by SetF and it is included in the ordering π at the first still undefined place. The search continues with v's successors $succ(v)$.

SetL works analogously but it uses the flag L and the role of successors and predecessors is swapped. $last$ stands for the last entry in π that is yet undefined.

If there are no cycles in the graph G then all nodes get either F or L and the returned ordering π is consistent (compare eq. 2.18). If the graph contains cycles the ordering is only consistent for the nodes that do not have the C-flag. That is not surprising, because a consistent ordering does not exist in this case. But most of the time we can still construct a better ordering; this is treated in the next section.

Procedure SetL(v)

if $\forall w \in succ(v) : attr(w) = L$ **then**
    $attr(v) = L$;
    $\pi(last) = v$;
    for $w \in pred(v)$ **do if** $attr(w) = C$ **then** SetL(w)
end

SetAttr is called once for each node. SetF is called at most once for each node by SetAttr. Additionally for a node v SetF(v) might be called by each of v's predecessors. Similarly SetL(v) may be called once by SetAttr and at most once by each of v's successors. This leads to a total complexity of this algorithm of $\Theta(\,|V| + |E|\,)$.

2.4 Cyclic case

If the graph G is not acyclic we will not get a consistent ordering. But the goal is to find an ordering such that the resulting blocks in the lower triangular system are as small as possible.

Once we note that there is a link between blocks and strongly connected components in the graph it is rather easy and fast to find an optimal ordering that leads to the smallest possible blocks. Therefore we reintroduce strongly connected components: A graph is strongly connected if and only if there exists a directed path from each vertex to each vertex. A subgraph G_S of G is a strongly connected component of graph G if and only if it is strongly connected and there is no larger subgraph containing G_S that is strongly connected. The condensation of G is the graph that is obtained if each strongly connected components of G is condensed into one vertex.

Whenever there is a nontrivial strongly connected component in our matrix graph we will get a block in the sorted matrix that contains at least the degrees of freedom in this component, no matter how we order them. This is a consequence of the fact that each vertex in a strongly connected component is reachable by each vertex in the same component and therefore each degree of freedom depends on all the degrees of freedom in the same component. So the smallest blocks that we might achieve are those corresponding to the strongly connected components.

We will see that this is indeed possible: A simple but useful fact for this purpose is that the condensation of a graph is always acyclic. (Assume that there was a cycle in the condensed graph. Then obviously there would have been a cycle in the original graph that contains vertices from different strongly connected components, which leads to a contradiction with the definition of strongly connected components.) This implies that we can order the strongly connected components consistently. The degrees of freedom can then be ordered only according to this component ordering (the ordering in the component is not important here) and the ordering will be consistent except for some relations inside the strongly connected components.

An efficient algorithm for computing the strongly connected components is Tarjan's Algorithm (see [4] or [3]):

Algorithm 5: Tarjan's Algorithm

input : graph $G = (V, E)$
output: strongly connected components *components*
$index = 1$
$S = \{\}$
$components = \{\}$
for $v \in V$ **do**
  if $index(v)$ *is undefined* **then** tarjan(v)
end

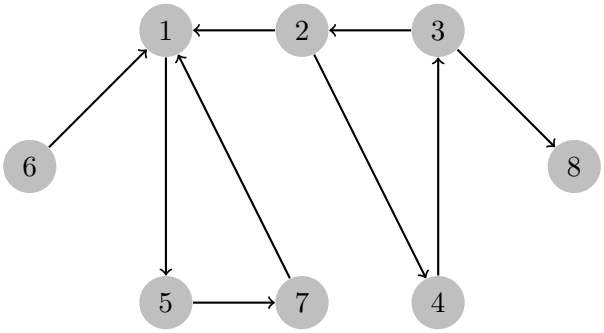
Procedure tarjan(v)

$index(v) = index$
$lowlink(v) = index$
$index = index + 1$
$S.push(v)$
for $(v, v') \in E$ **do**
  if $index(v')$ *is undefined* **then**
    tarjan(v')
    $lowlink(v) = min(lowlink(v), lowlink(v'))$
  end
  else if $v' \in S$ **then**
    $lowlink(v) = min(lowlink(v), index(v'))$
  end
end
if $lowlink(v) = index(v)$ **then**
  $c = \{\}$
  repeat
    $v' = S.pop()$
    $c = c \cup \{v'\}$
  until $v' = v$
  $components = components \cup \{c\}$
end

The algorithm performs a depth-first search. Each node v gets an index $index(v)$ that is increasing in the order of appearance. Additionally each node is kept in a stack S until we have identified its components. In order to determine the components we have an additional property, called *lowlink*, for each node. $lowlink(v)$ contains the smallest index of a node discovered so far that is reachable from v (and that is not contained in an already identified component). $lowlink(v)$ is initialised by the *index* and then we update it using the values of the successors. If after all this updating $lowlink(v) = index(v)$ holds, this means that v is the node with the smallest index of a strongly connected component. All nodes in this component are on the stack S and we only have to pop elements from it until we get v. We can store all those elements in

a list and add this list to the list of components. Fig. 2.1 illustrates the algorithm applied to a small graph.

The algorithm is efficient: the procedure tarjan is called once for each node. For the update of the lowlinks each edges is considered once. If the test whether v' is in S is done in constant time (by using a flag for each node), this leads to a complexity of $\Theta(\,|V| + |E|\,)$.



v	$index(v)$	$lowlink(v)$	S	c
1	1	1	$\{1\}$	
5	2	2	$\{1,5\}$	
7	3	3	$\{1,5,7\}$	
7	3	1	$\{1,5,7\}$	
5	2	1	$\{1,5,7\}$	
1	1	1	$\{1,5,7\}$	
			$\{\}$	$\{7,5,1\}$
2	4	4	$\{2\}$	
4	5	5	$\{2,4\}$	
3	6	6	$\{2,4,3\}$	
3	6	4	$\{2,4,3\}$	
8	7	7	$\{2,4,3,8\}$	
			$\{2,4,3\}$	$\{8\}$
4	5	4	$\{2,4,3\}$	
2	4	4	$\{2,4,3\}$	
			$\{\}$	$\{3,4,2\}$
6	8	8	$\{6\}$	
			$\{\}$	$\{6\}$

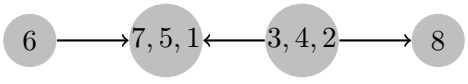


Figure 2.1: Illustration of Tarjan's algorithm. Top: Example graph. Centre: Summary of the changes of $index(v)$ and $lowlink(v)$ of the node v in the running of the algorithm, together with the stack S and the identified strongly connected components c. Bottom: Component-wise condensation.

3 Implementation

The Implementation is done in the Fenics/Dolfin framework.[1] The code language is C++, but the classes can also be used through a python interface. This chapter should give a brief introduction to the code. There are two general notes to be made.

We use PETSc as the linear algebra backend.[2] Mostly we do not explicitly use the PETSc data types, but sometimes it is not avoidable or we need access to the underlying data structure to efficiently implement the algorithms.

Although, we have used 1-based indices in the last chapter (e.g. for the degrees of freedom), we will switch here to the 0-based convention of C++.

3.1 Ordering

The class `Ordering` is the base class for all orderings. The ordering should be created in the constructer of the sub class and then be stored in the field `ordering`. The size of the graph for which the ordering is constructed is stored in the variable `size`. `blocks` contains the blocks that the ordering algorithm has identified. It contains the start and the end indices of the blocks as pairs and they should be sorted. (When working with blocks one must not assume that the sorted matrix is lower block triangular with those blocks, but rather assume that this is a good block partitioning.) Blocks of size one are usually omitted.

The member variables are:

```
dolfin::uint size;
std::vector<dolfin::uint> ordering;
std::vector<std::pair<dolfin::uint,dolfin::uint> > blocks;
```

they are declared as protected, to be accessible by the sub classes.

The constructor

```
Ordering(dolfin::uint size)
```

just gets the size of the graph and does some initialisation.

The methods

```
void get_ordering(std::vector<dolfin::uint> & o)
void get_inverse_ordering(std::vector<dolfin::uint> & io)
```

[1] The FEniCS Project: `http://www.fenics.org/`
[2] Portable, Extensible Toolkit for Scientific Computation (PETSc): `http://www.mcs.anl.gov/petsc/`

can be used to get copies of the ordering (permutation) or its inversion.

The methods

```
void sort_matrix(const dolfin::Matrix& A, dolfin::Matrix& As )
void sort_rhs(const dolfin::GenericVector& b, dolfin::GenericVector& bs)
void unsort_unknowns(const dolfin::GenericVector& xs, dolfin::GenericVector& x)
```

can be used to sort the system and then to get the solution back in the original order. The two arguments should never be the same. For efficiency reason the sorting of the matrix is done as follows: First the sparsity pattern of the sorted matrix is constructed and only then the entries in the sorted matrix are stored.

3.2 SimpleOrdering

The class `SimpleOrdering` is derived from `Ordering` and it implements the ordering as described in algorithm 4 (topological sorting).

The constructors

```
SimpleOrdering(const dolfin::Matrix& A,bool verbose=false);
SimpleOrdering(const double A[],int size,bool verbose=false);
```

take a matrix `A` and construct an ordering based on its matrix graph. If `verbose` is true more output is generated.

```
SimpleOrdering(std::vector<std::vector<dolfin::uint> >& pred,
               std::vector<std::vector<dolfin::uint> >& succ,bool verbose=false);
```

The last constructor gets the graph directly via a list of the predecessors and successors of each node (`pred` and `succ`).

The predecessors and successors are stored in

```
std::vector<std::vector<dolfin::uint> >* ppred;
std::vector<std::vector<dolfin::uint> >* psucc;
```

The variables are lists of the predecessors and successors for each degree of freedom.

The methods

```
void get_predecessors(std::vector<dolfin::uint>& pred, const dolfin::Matrix& A,
                      dolfin::uint idx);
void get_successors(std::vector<dolfin::uint>& succ, const dolfin::Matrix& At,
                      dolfin::uint idx);
void get_predecessors(std::vector<dolfin::uint>& pred, const double A[],
                      dolfin::uint idx);
void get_successors(std::vector<dolfin::uint>& succ, const double A[],
                      dolfin::uint idx);
```

are used to construct those. `idx` is the index of the node for which the predecessors or successors are extracted.

Once the information of the graph is construced, all constructors call

```
void create_ordering();
```

This procedure directly implements the algorithm 4 (simple ordering) using the functions from section 2.3

```
void set_F(dolfin::uint i);
void set_L(dolfin::uint i);
void set_attr(dolfin::uint i);
```

The attribute of each node is stored in an array

```
std::vector<flag> flags;
```

where the type `flag` is defined as follows:

```
enum {Fflag,Cflag,Lflag};
typedef dolfin::uint flag;
```

During the construction of the ordering one has to know which index the next node that gets an F-flag would get ($first$ in the algorithm) and similar the new index of the next node that gets the attribute L ($last$ in the algorithm). This is stored in

```
dolfin::uint next_F;
dolfin::uint next_L;
```

and increased or decreased whenever a new node with the corresponding flag is inserted in the ordering.

3.3 CycleOrdering

`CycleOrdering` is based on `SimpleOrdering` (but it does not inherit from it). Everything is the same except that the routines for the dense matrices are left away and that `create_ordering` also orders the nodes with the attribute C.

It determines the strongly connected components of the subgraph of the C-nodes using algorithm 5 (Tarjan's algorithm). For efficiency reasons and to avoid a stack overflow the recursion is replaced by explicitly using a stack of the nodes that have to be considered.

Then the condensation of the graph is constructed by looping over all edges and testing whether they link different components. This part has to be implemented carefully in order not to destroy the order of complexity. In principle one could get lists of predecessors and successors of the components of order $\mathcal{O}(n)$ in some cases. Doing a sort on those would then lead to $\mathcal{O}(n \log(n))$. Using a flag whether a certain edge is already included one can avoid the overhead. The resulting lists are no longer sorted, but this is not required in the following part.

`SimpleOrdering` is then used to determine the ordering of the components. The component-wise predecessors and successors (condensation graph) are passed.

In the end, the ordering is constructed. Component by component in the computed ordering, all their nodes are inserted. Inside the components (that correspond to the blocks in the sorted matrix) the nodes are not further ordered.

3.4 ElementOrdering

`ElementOrdering` implements an ordering based on the element-wise matrix graph as defined in last part of section 2.2.

```
ElementOrdering(const dolfin::Matrix& A, const dolfin::DofMap& dofm,
                const dolfin::Mesh& m,bool verbose=true);
```

The constructor takes a matrix `A` and computes its matrix graph (we need here only the predecessors).

Next the condensation is computed. We need to know which degree of freedom corresponds to which element, that is why we need the degree of freedom map `dofm` and the mesh `m`.

Once we have the predecessor and the successors in this element-wise graph, `CycleOrdering` is used to construct an ordering of the elements.

The last step is then to construct a suitable ordering of the degrees of freedom. Element by element in the just computed ordering, all indices to the corresponding degrees of freedom are inserted into the ordering.

Apart from a few procedures to compute the predecessors and successors (not all of them are used), there is nothing more in this class.

3.5 Preconditioners

There are two new preconditioners. Both of them are derived from `dolfin::PETScPreconditioner` that they can be used in the corresponding Krylov solver. The first is

```
class MySORPreconditioner : public dolfin::PETScPreconditioner
```

It performs Successive Over-Relaxation (SOR) on the sorted matrix implicitly. It works directly on the original matrix and uses the ordering via additional index transformations.

The constructor

```
MySORPreconditioner(const dolfin::Matrix& A,const Ordering& o,
                    unsigned int itermax, double omega,
                    bool backwardToo, bool zeroInitialGuess);
```

needs to get the original matrix `A` and the ordering `o` that should be used. Additionally it gets the number of SOR-iterations `itermax` that should be done in each step and the relaxation parameter `omega`. If `backwardToo` is `true` not only forward sweeps are done. Instead a forward and backward sweep is performed in each iteration (symmetric SOR). `zeroInitialGuess` indicates that in the beginning the solution vector should filled with zero values. For both flags the default values is `true`.

The only method that needs to be implemented is

```
virtual void solve(dolfin::PETScVector& x, const dolfin::PETScVector& b);
```

which returns an approximate solution to the system `A*x=b`. This is a straightforward implementation of the SOR iteration.

The second preconditioner is `MyBlockGSPreconditioner`. It performs a block Gauss-Seidel iteration as described in algorithm 3. Since the smaller blocks are dense anyway, a dense direct solver based on LU-decomposition is used for those. For the bigger blocks SOR is used. Convergence is not guaranteed then, but it has basically worked well in our cases. Blocks are considered to be small if they are not larger than the constant `MAXSIZELU` defined in the code. This should be at least the number of degrees of freedom per element.

As the first preconditioner it works implicitly on the sorted matrix, but it uses the original matrix and additional index transformations corresponding to the ordering.

The constructor

```
MyBlockGSPreconditioner(const dolfin::Matrix& A,const Ordering& o,
                        unsigned int itermax, double omega,
                        bool backwardToo, bool zeroInitialGuess);
```

has the same form as the constructor for `MyBlockGSPreconditioner`. But note that the number of iterations `itermax` and the relaxation parameter `omega` only correspond to the SOR-iterations for the systems due to the bigger blocks. Globally, the function

```
virtual void solve(dolfin::PETScVector& x, const dolfin::PETScVector& b);
```

performs only one iteration. If `backwardToo` is set, in deviation from the algorithm not only forward sweeps but also backward sweep are done. This holds true for the global iteration as well as for the SOR-iterations of the bigger blocks systems.

As the most costly part in the solution of a linear system is the LU-decomposition, it is only computed when `solve` is called the first time and is afterwards reused.

Because of stability we do column pivoting. The column pivoting has to be stored too. Hence, we defined the following two variables

```
std::vector<std::vector<double> > LUblocks;
std::vector<std::vector<dolfin::uint> > pivots;
```

and use them to store the computed LU-decomposition together with the pivoting information.

To solve the small systems we then permute the right hand side and do a forward and backward substitution.

3.6 Remaining code

The remaining code consists mainly of the definitions of the different problems that are defined in the next chapter. Because it is a direct implementation of these problems, we do no concentrate longer on this code.

But to get an overview, the base structure is typically as follows: The solution and the velocity field are defined. The mesh is generated. The boundary conditions are defined and later on imposed. The bilinear and the linear forms are defined and the corresponding matrices and vectors are assembled. The ordering is constructed, based on the advection part. The linear system is solved and finally errors and error rates are computed.

3.7 Complexity

We discuss here the complexity of some of the algorithms applied to the sparse linear system in the finite element context.

For the construction of orderings with `ElementOrdering` we have get the (element-wise) matrix graph. When we assume that shape regular meshes are used, there exists a constant, such that each element has less neighbouring elements. The number of matrix entries due to one element is at most its number of degrees of freedom times the number of degrees of freedom of its neighbours. Unless we use p-adaptivity this is then also bounded by a constant. All in all, the number of matrix entries and therefore the number of edges in the matrix graph is $\mathcal{O}(n)$.

The element-wise condensation graph has therefore also $\mathcal{O}(n)$ nodes and $\mathcal{O}(n)$ edges. It can be computed efficiently in $\mathcal{O}(n)$.

Using the topological sorting and then Tarjan's algorithm needs therefore also only $\mathcal{O}(n)$ (because both of them are linear in the number of edges and nodes). The condensation into the component-wise graph can also be done in linear time; the number of nodes and edges is not bigger than in the original graph. Hence, also the component-wise ordering can be constructed efficiently in $\mathcal{O}(n)$.

This results in a complexity of $\mathcal{O}(n)$ for the complete algorithm, which is the optimal order that one can hope for.

Sorting the matrix and the right hand side is $\mathcal{O}(n)$. It is not $\mathcal{O}(n \log(n))$ because we do not sort the entries, but we do only rearrange them in the ordering already computed. (It depends on the framework, whether this is really achieved, but in principle it is doable.)

Concerning the preconditioners, each iteration of `MySORPreconditioner` has a complexity in the order of the number of the matrix elements, which is $\mathcal{O}(n)$. The block Gauss-Seidel iteration

of `MyBlockGSPreconditioner` needs $\mathcal{O}(n)$ to apply all the non-block-diagonal elements of the matrix. The solution of the block-wise systems costs as follows: The size of the blocks for which the LU-decomposition is used is fixed. Therefore the complexity of the solution of the small systems is linear in the number of degrees of freedom contained in small blocks. For the big blocks a fixed number of SOR-iterations is done, this results in costs linear to the number of degrees of freedom in big blocks. Together, this implies a complexity of $\mathcal{O}(n)$ for each iteration of this preconditioner.

4 Problems and Results

We have used different problem cases to test and validate the code.

4.1 Advection-diffusion equation in 3D

First we consider the scalar advection diffusion equation

$$-\epsilon \Delta u + \boldsymbol{b} \cdot \boldsymbol{\nabla} u = f \tag{4.1}$$

on the unit cube $[0,1]^3$. u is the unknown scalar function, $\boldsymbol{b}$ is the velocity field and f is the source term. $\epsilon \geq 0$ is the diffusivity coefficient. The smaller it gets relative to $|\boldsymbol{b}|$ the more dominating is the advection term.

We use a source term f and Dirichlet boundary conditions such that solution is given by

$$u(x,y,z) = y(1-y)(1-x)(1-z) \tag{4.2}$$

Three different flow fields are used:

1. a constant flow (we will refer to it as CONST)

$$\boldsymbol{b}(x,y,z) = (0.6, 0.8, -0.3)^{\mathrm{T}}, \tag{4.3}$$

2. a smooth perturbation of the constant flow (SIN)

$$\boldsymbol{b}(x,y,z) = (0.6, 0.8 + 2\sin(4\pi x), -0.3 + 0.2\sin(4\pi y))^{\mathrm{T}}, \tag{4.4}$$

3. a U-turning flow (UTURN)

$$\boldsymbol{b}(x,y,z) = \begin{cases} (-r\sin(\phi), r\cos(\phi), -0.1)^{\mathrm{T}}, & x > 1/2 \\ (-(y-1/2), 0, -0.1)^{\mathrm{T}}, & \text{otherwise} \end{cases} \tag{4.5}$$

   where $\phi = \mathrm{atan2}(y-1/2, x-1/2)$ and $r = \sqrt{(x-1/2)^2 + (y-1/2)^2}$.

We use the BiCGSTAB Krylov solver. For the preconditioners we distinguish three different methods: SOR corresponds to using the default PETSc-SOR preconditioner on the original system. For SORTSOR we sort the system according to the element-wise ordering and use then Krylov solver with again the default PETSc-SOR preconditioner. For the last method BLOCKGS we construct an element-wise ordering too, but we do not explicitly sort the system. Instead

the block Gauss-Seidel preconditioner described in section 3.5 that directly works on the original system is used.

The default PETSc-SOR preconditioner does a symmetric SOR (SSOR): a forward and a backward substitution. The value used for ω is 1.0, which means that it does two Gauss-Seidel sweeps. In principle, the default PETSc-SOR preconditioner would do a block SOR by combining successive lines with the same non-zero structure. But to do a meaningful comparison we have bypassed this behaviour. For the block Gauss-Seidel preconditioner `omega` is 1.0 and `itermax` is 10, but for almost all systems all blocks are small and these parameters are not important, because only LU-decomposition is then used.

The total time we consider here is the time that is needed to solve the system once it is assembled. For all three methods this implies that the time that the Krylov solver needs is included. Additionally for SORTSOR and BLOCKGS the time that is spent to construct the ordering is also included. As we need to sort the matrix in SORTSOR this is also part of the time measurement there. The timing are done on a AMD Opteron Quad-Core processor with 2300 MHz, but only one core is used.

The meshes are uniform and regular, linear elements are used (but a higher polynomial degree would be possible). The stopping criterions for the Krylov solver are (among others) a relative tolerance of 10^{-8} and a maximum number of iterations of 1000.

We study the effect of different values for the diffusivity constant ϵ. We therefore use four different values: 0, 10^{-6}, 10^{-3}, 0.1.

For the case where there is no diffusion ($\epsilon = 0$), we expect that the block Gauss-Seidel preconditioner is quite fast. Indeed it is the fastest method in all three different flows (top left of fig. 4.1, 4.2 and 4.3). As illustrated in fig. 4.4 only one iteration is needed in the CONST case. The reason is that there are no cycles in the element-wise graph and the resulting matrix is lower block triangular with blocks of size 4 (the number of degrees of freedom per element) as it can be seen in table 4.1. For the other two flows there are some cycles in the element-wise graph but as soon as the flow is well resolved the blocks get small (compare table 4.2 and 4.3). Because we use LU-decomposition up to a block size of `MAXSIZELU`= 12, also there the solver converges in one step.

block size	n=3000	n=24000	n=192000	n=648000	n=1536000
4	750	6000	48000	162000	384000

Table 4.1: Number of blocks of a certain size for the CONST case.

For a small ϵ of 10^{-6} the block Gauss-Seidel preconditioner performs still quite well (top right of fig. 4.1, 4.2 and 4.3). Due to the diffusion the upper part of the matrix is no longer zero and the Krylov solver needs more than just one iteration until it converges. But the effect is small and BLOCKGS is still the fastest solver.

For higher values of ϵ the block Gauss-Seidel preconditioner gets slower and slower (bottom of fig. 4.1, 4.2 4.3). For $\epsilon = 10^{-3}$ BLOCKGS is approximatively as fast as SOR, but for $\epsilon = 0.1$ SOR is

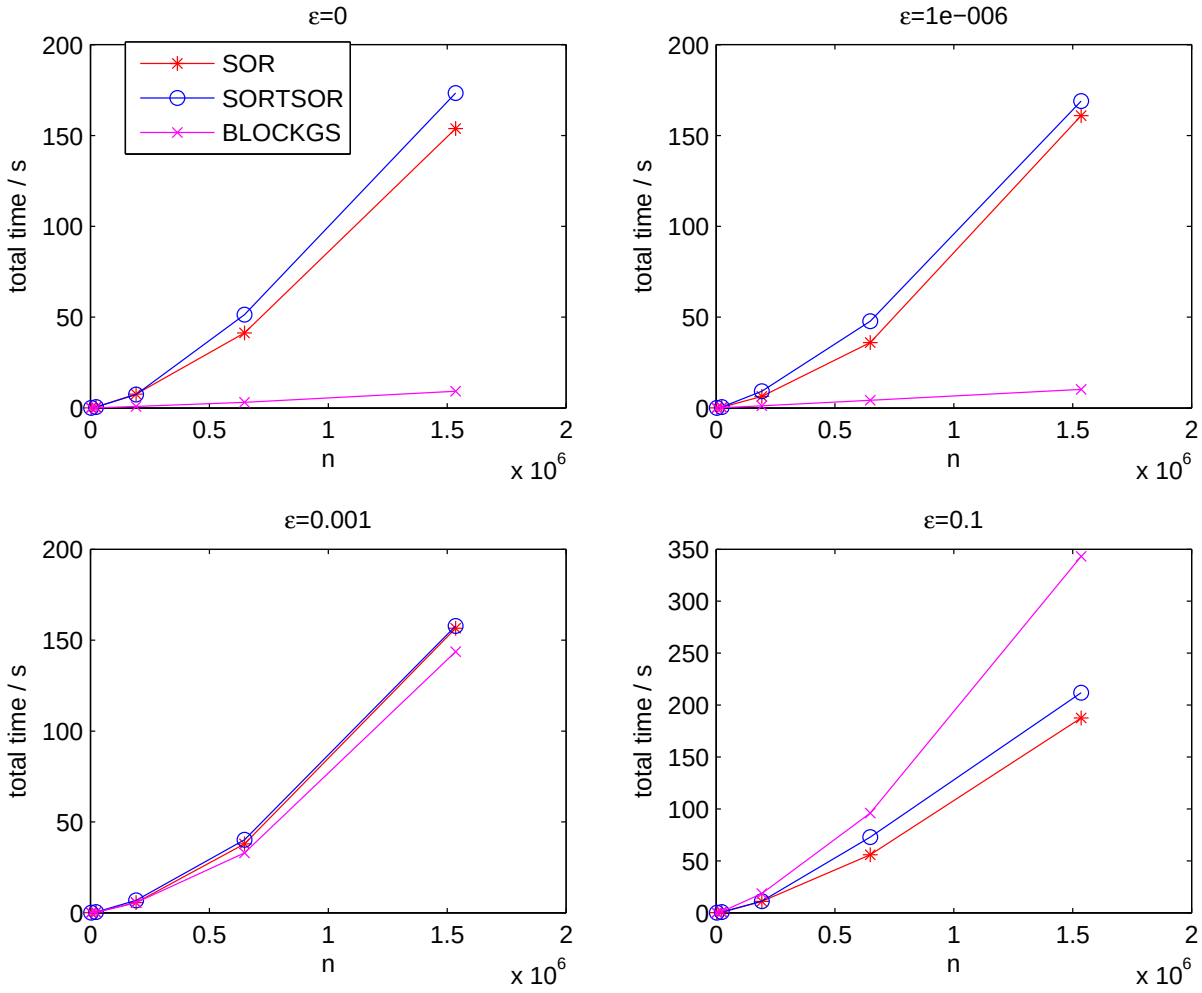


Figure 4.1: Total time to solve the linear system in dependence of the number of degrees of freedom n for different values of the diffusivity coefficient ϵ for the CONST case.

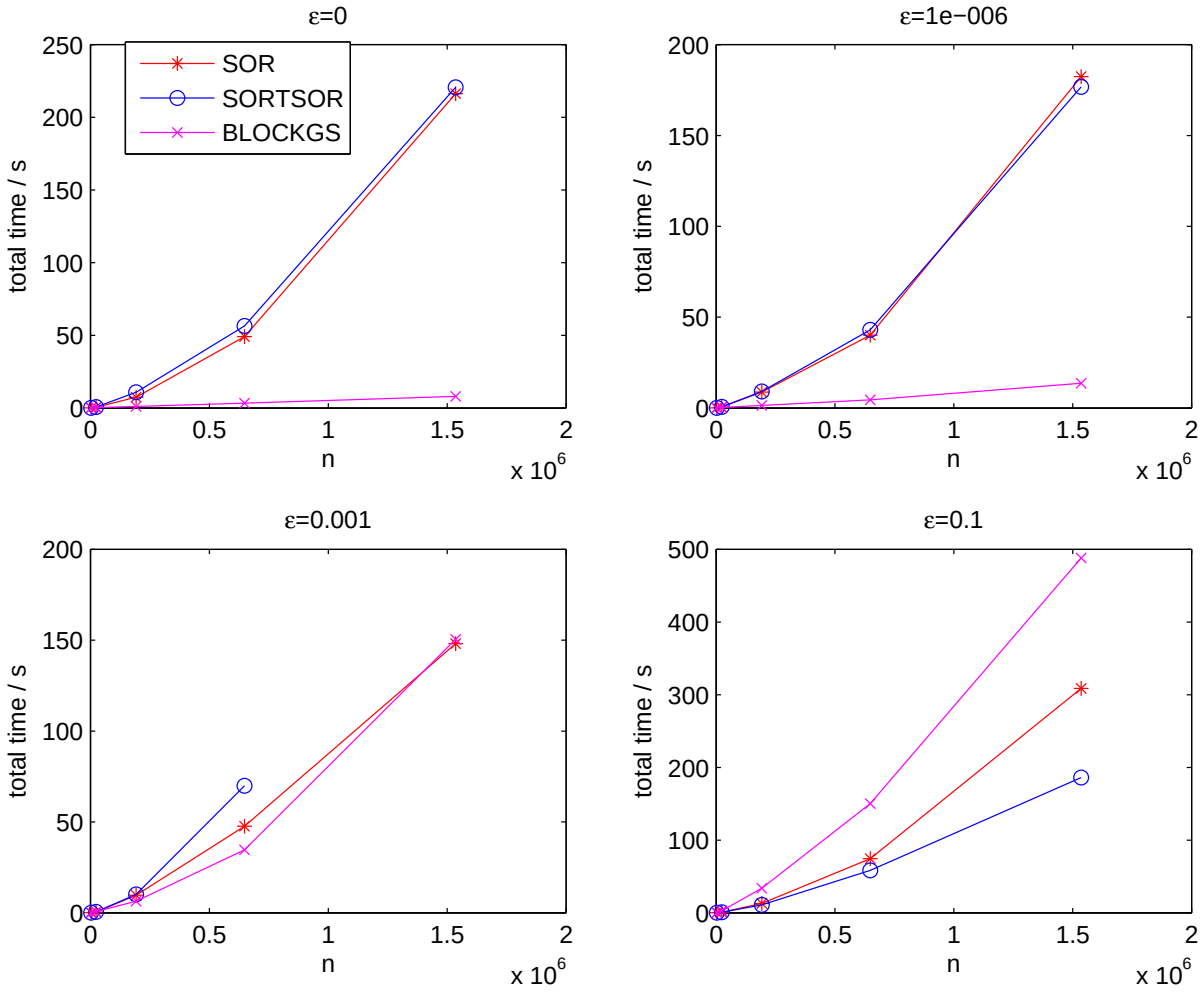


Figure 4.2: Total time to solve the linear system in dependence of the number of degrees of freedom n for different values of the diffusivity coefficient ϵ for the SIN case.

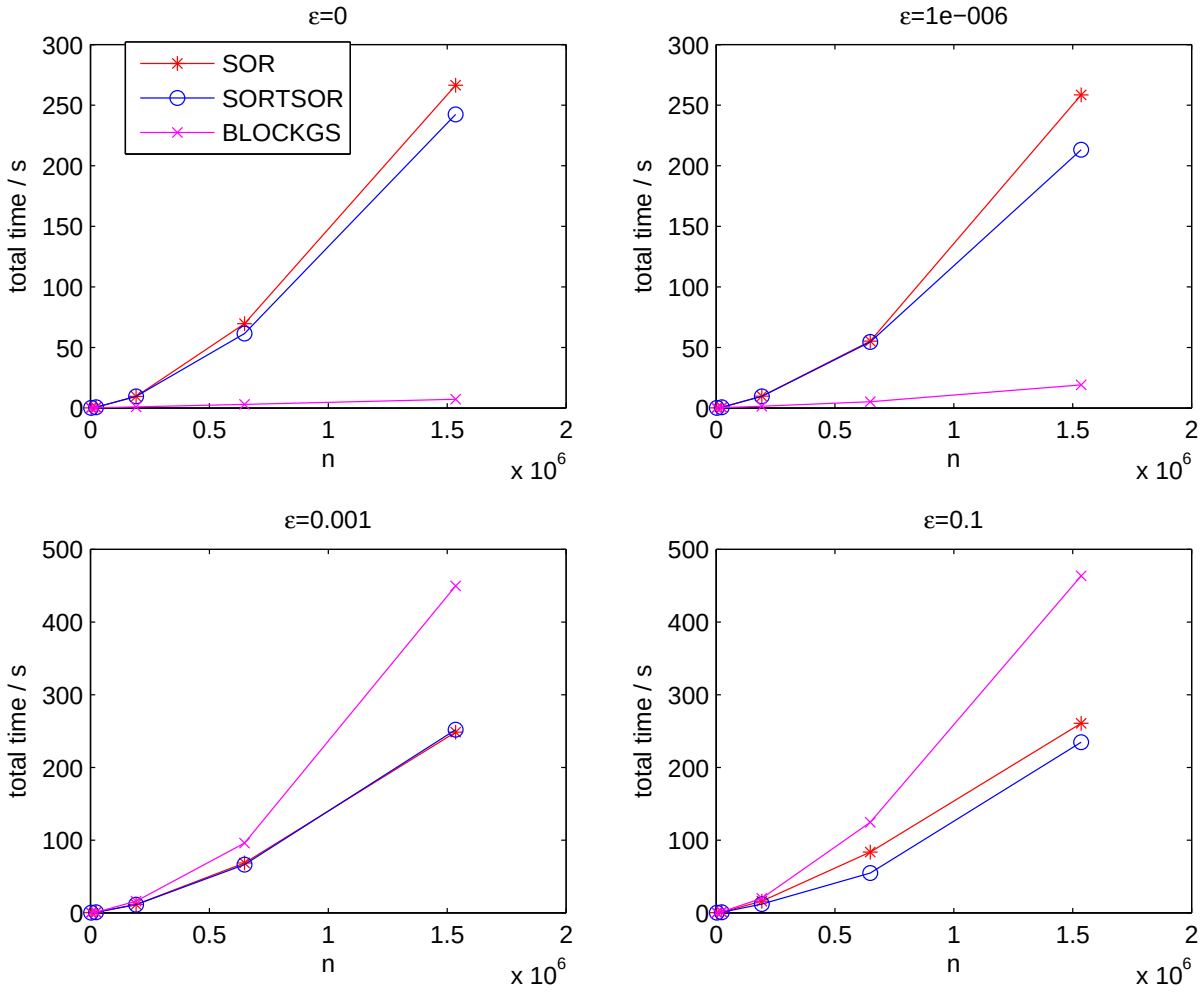


Figure 4.3: Total time to solve the linear system in dependence of the number of degrees of freedom n for different values of the diffusivity coefficient ϵ for the UTURN case.

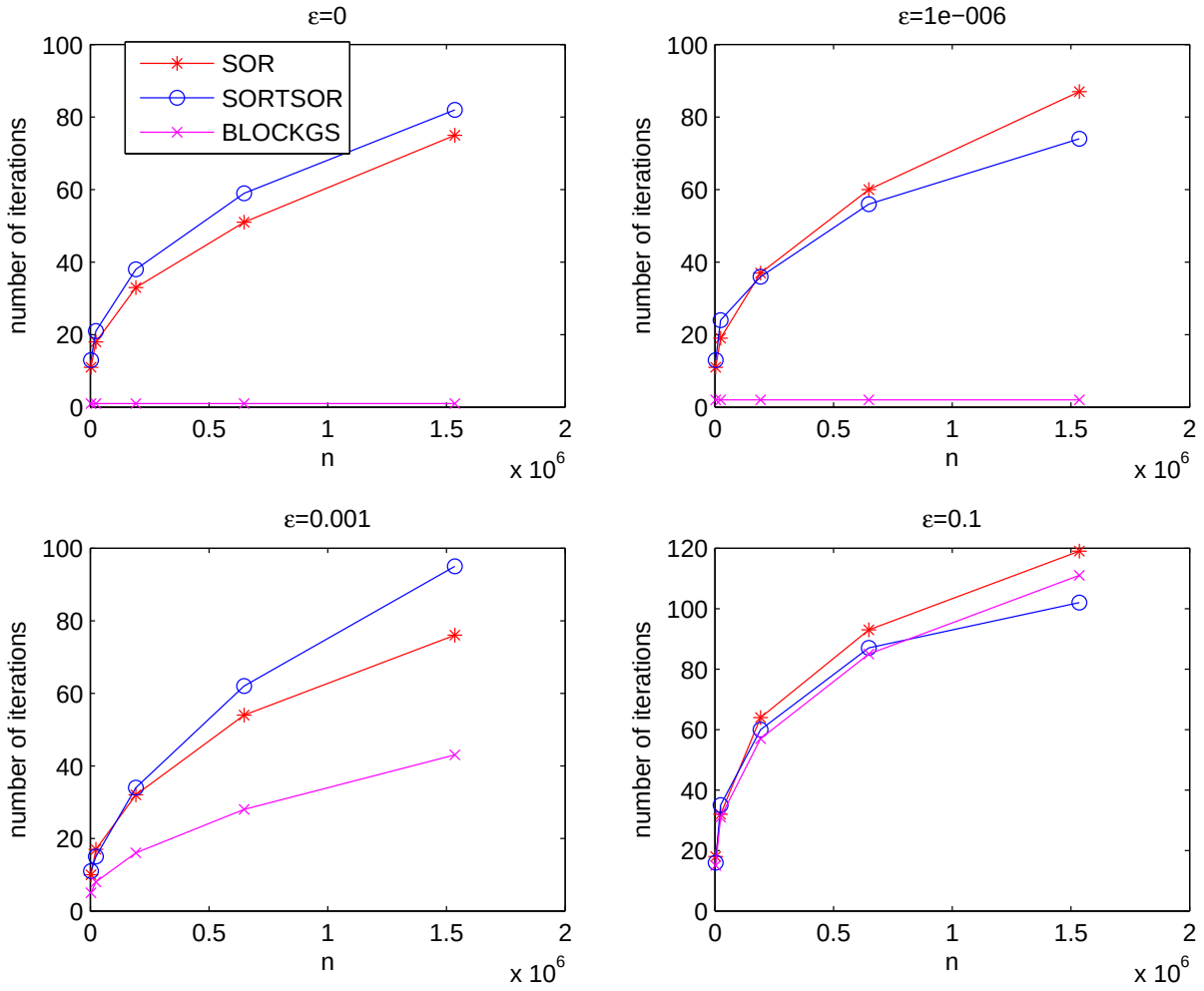


Figure 4.4: Number of iterations needed to the linear system in dependence of the number of degrees of freedom n for different values of the diffusivity coefficient ϵ for the CONST case.

block size	n=3000	n=24000	n=192000	n=648000	n=1536000
4	272	3824	38364	131285	320490
8	20	238	1278	12839	21918
12	5	260	1800	1679	6558
16	5				
24	5				
28		20			
36		20	80		
48			80		
52	1				
60	24				
120		20			

Table 4.2: Number of blocks of a certain size for the SIN case.

block size	n=3000	n=24000	n=192000	n=648000	n=1536000
4	504	5802	47202	160202	380802
8	53	99	399	899	1599
12	10				
16	5				
24	5				
48	5				

Table 4.3: Number of blocks of a certain size for the UTURN case.

faster. BLOCKGS needs clearly less iterations for $\epsilon = 10^{-3}$ and about as many iterations for the case $\epsilon = 0.1$ (see fig. 4.4). We expect that with some implementation improvements BLOCKGS could beat SOR in first case because of the big difference in the number of iterations.

Concerning SORTSOR for $\epsilon = 0$ and $\epsilon = 10^{-6}$ in all cases (top of fig. 4.1, 4.2 and 4.3) it needs about as much time as SOR. Even worse, although the preconditioned residuum reduces during the solution process in these cases, the final residuum is still big because it starts with a very high value. The true residuum is significantly enlarged in the first iteration and only then reduced. In these cases SORTSOR is therefore useless.

For higher values of the diffusion coefficient (bottom of fig. 4.1, 4.2 and 4.3) we do no longer observe this problems or only in a very reduced form. But SORTSOR does not lead to a significant improve compared to SOR.

To conclude: For the pure advection case BLOCKGS is the fastest. When the diffusivity is increased BLOCKGS remains the best preconditioner up to a certain value. For even bigger diffusion SOR takes the lead. SORTSOR leads to some convergence problems especially for a small diffusion coefficient. Only sorting the system does not yield a fast preconditioner, but combined with a block Gauss-Seidel approach it performs very well for a small diffusion coefficient.

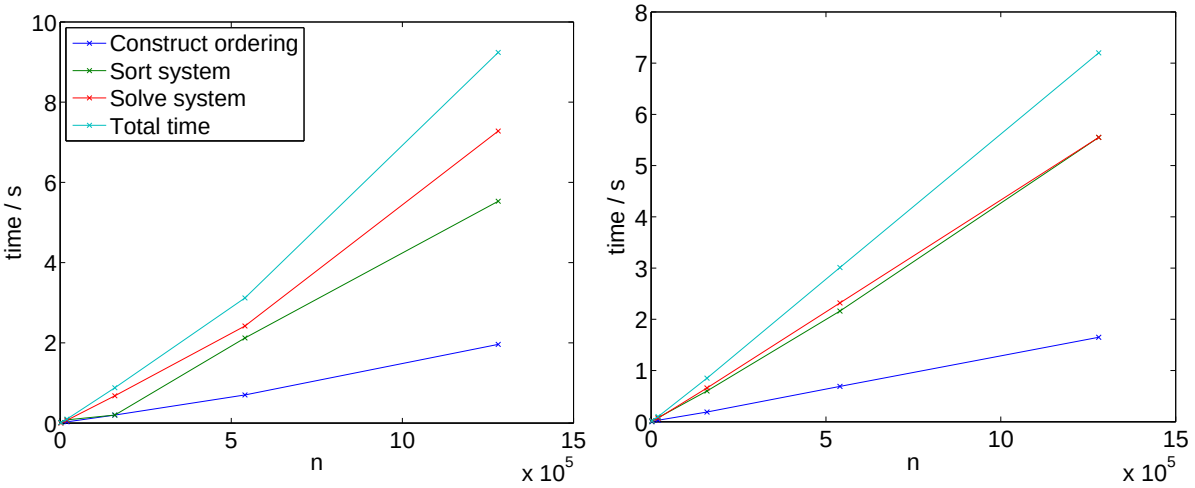


Figure 4.5: Time needed for the different parts of the BLOCKGS approach for the CONST flow (left) and the UTURN flow (right) with $\epsilon = 0$ together with the time that an explicit sorting of the matrix would need.

The last thing done in this section is an investigation, where the algorithm spend their time. We use the BLOCKGS case and compare how much time the construction of the ordering and the solution of the system take (see fig. 4.5). We include also the time a sorting of the system takes. For $\epsilon = 0$ the Krylov solver needs for both the CONST and the UTURN case only one iteration. Hence, this part is small here. The construction of the ordering takes in general at most as much time as a few Krylov iterations. Indeed, in the considered cases it is the part with the lowest costs. Sorting of the system would take about as much time as solution of the system takes. So here it is not worth to sort the system explicitly. But as the diffusion increases we need more iterations and the solution of the system needs more and more time. A sorting would then become almost negligible and would even lead to an increased performance of the block Gauss-Seidel iterations because the memory would be accessed more linearly. So a block Gauss-Seidel preconditioner based on the explicitly sorted system would be faster for at least moderate diffusion, but due to the bounded time of this work it has not yet been implemented.

4.2 Generalized diffusion-convection problem in 3D

We consider now a non-scalar diffusion-convection problem. The equation that we study is

$$\epsilon \boldsymbol{\nabla} \times \boldsymbol{\nabla} \times \boldsymbol{u} - \boldsymbol{b} \times \boldsymbol{\nabla} \times \boldsymbol{u} + \boldsymbol{\nabla}(\boldsymbol{u} \cdot \boldsymbol{b}) = \boldsymbol{f} \tag{4.6}$$

on the unit cube $[0,1]^3$. $\boldsymbol{u}$ is the unknown vector field. $\boldsymbol{b}$ is the known velocity field and $\boldsymbol{f}$ is the source term.

The boundary conditions and the source term are adjusted such that the solution is given by

$$\boldsymbol{u}(x,y,z) = \left(\sin(\pi z), (1-y^2)(1-x^2), \sin(\pi y)\right)^{\mathrm{T}}. \tag{4.7}$$

For the velocity a constant flow is used (CONST)

$$\boldsymbol{b}(x, y, z) = (0.5, 0.25, 0.75)^{\mathrm{T}} \tag{4.8}$$

or a slightly more complicated flow (NORMAL)

$$\boldsymbol{b}(x, y, z) = \big(0.66(1 - y^2)(1 - x^2), 0.4(1 - z^2)(1 - x^2), 0.2 + \sin(\pi z)\sin(\pi x)\big)^{\mathrm{T}}. \tag{4.9}$$

For the diffusivity coefficient we use 0 or 10^{-6}. For higher values the solver did no longer converge for big systems.

The rest of the setting is the same. In particular we investigate the same preconditioners: SOR, SORTSOR and BLOCKGS. (The relative tolerance used in the stopping criterion is here set to its default value of 10^{-6}.)

The total time and the number of iterations needed to solve the system is depicted in fig. 4.6 (CONST) and in fig. 4.7 (NORMAL). BLOCKGS is the fastest method in all cases. For the CONST case only one Krylov iteration is needed, because without diffusion the sorted system is lower block triangular with each block corresponding to one element. For the NORMAL flow a few iterations are needed, especially when n is small and the therefore the flow is not yet well resolved. But also in this case without diffusion the sorted system is lower block triangular (see table 4.4). The reason that nevertheless more than one iteration is needed is that the blocks are not considered to be small and therefore SOR instead of LU is used to solve the corresponding systems. One could increase `MAXSIZELU` or the number of iterations in one block and it would be solvable in one iteration.

block size	n=4608	n=36864	n=294912
12	227	2292	21249
24	30	246	1399
36	23	74	135
48	1		1
72	4	11	20

Table 4.4: Number of blocks of a certain size for the NORMAL case.

SOR converges, but needs more time and iterations in all cases. The number of iterations increases as the system grows, therefore it is to be expected to perform worse for even bigger systems. Sorting the system and then performing SOR does not help here too. It leads to the same convergence problems as described in the previous section.

This supports the conclusions of the last section that sorting alone does not lead to a fast solver, but together with the block Gauss-Seidel approach advection problems with small diffusion can be solved quickly.

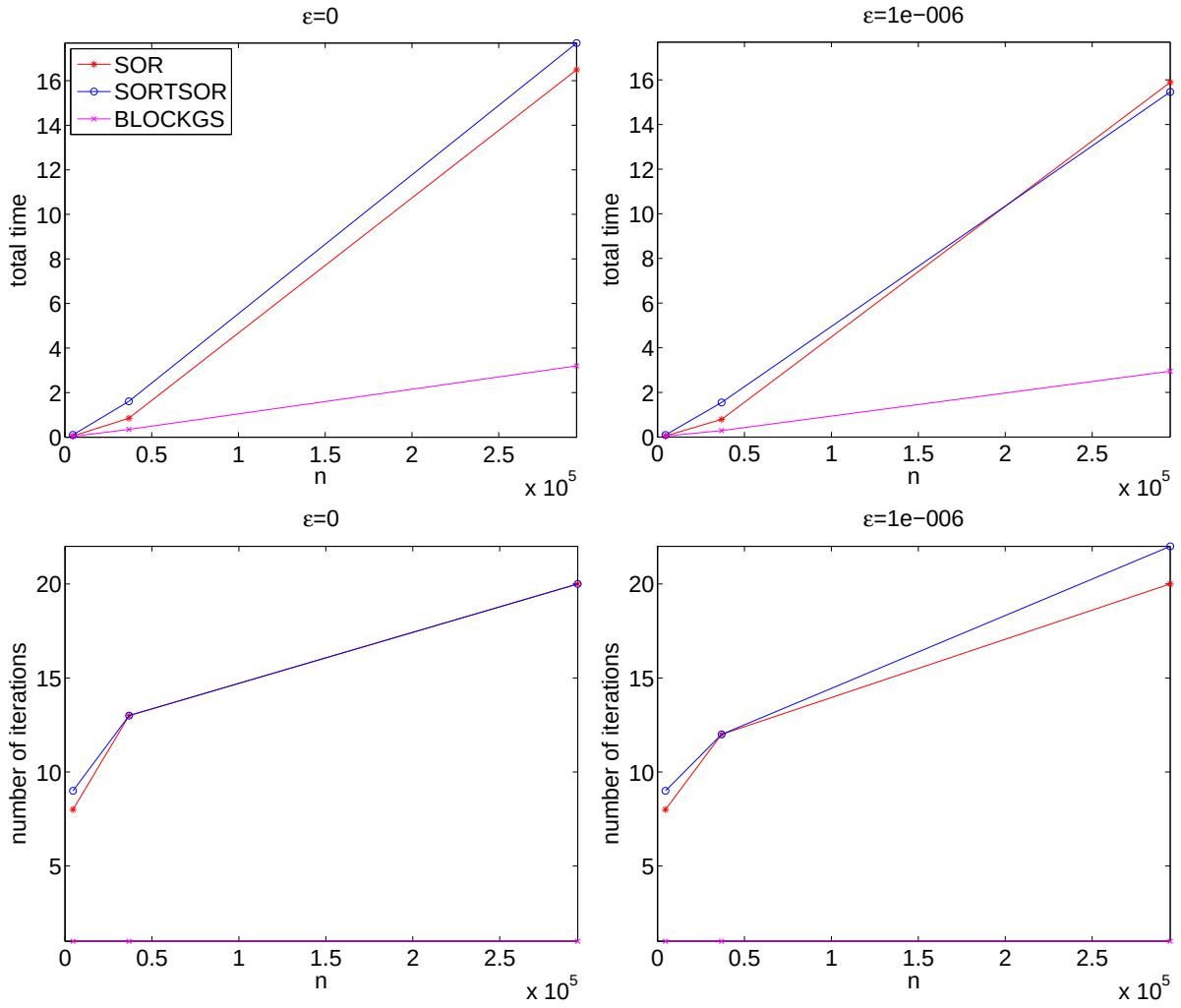


Figure 4.6: Total time and number of iterations needed for the CONST flow with $\epsilon = 0$ (left) and $\epsilon = 10^{-6}$ (right).

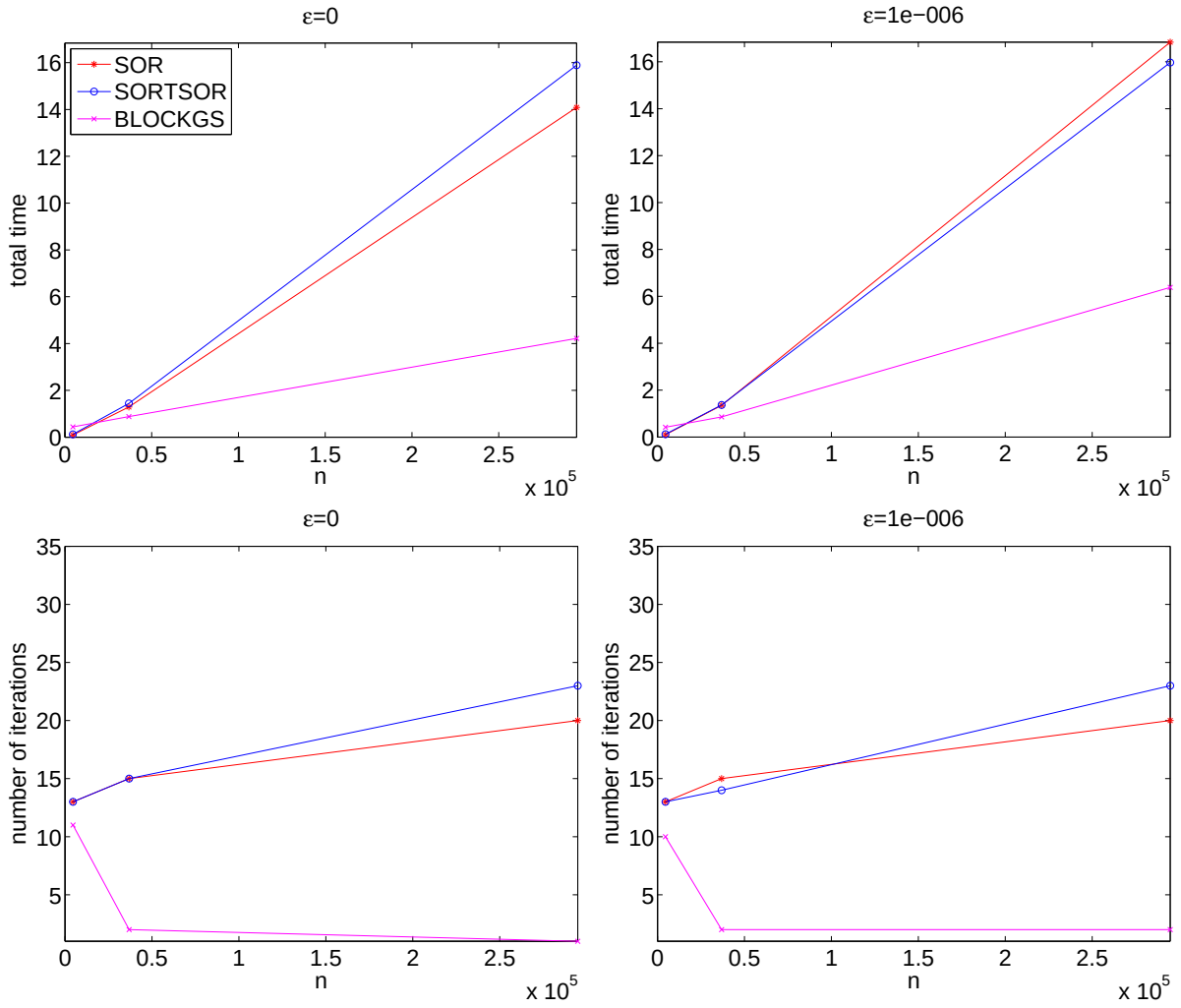


Figure 4.7: Total time and number of iterations needed for the NORMAL flow with $\epsilon = 0$ (left) and $\epsilon = 10^{-6}$ (right).

5 Conclusions

Our goal is to solve quickly the linear systems resulting of discontinuous Galerkin methods applied to pure advection and advection dominated problems. When upwinding is used we have shown experimentally that the system matrix is a permutation of a lower block triangular matrix in the cases without diffusion. To exploit this feature an ordering has to be explicitly constructed. Hence, we look at the matrix graph or at its element-wise condensation. If we are lucky, it is acyclic and a topological sorting suffices to determine an optimal ordering. In case there are cycles Tarjan's algorithm is used to identify the strongly connected components and further to deduce an ordering that leads to blocks that are as small as possible. In the cases with diffusion the sorted system matrix is almost lower block triangular, the upper part will be non-zero, but small. To construct the ordering in this case, we work only on the advection part.

Once the ordering is determined one can do a block Gauss-Seidel method on the sorted system to solve the problem approximatively. To avoid sorting the matrix one can also only implicitly work on the sorted system, but use the original system and additional index transformations.

The results show that BiCGSTAB together with the block Gauss-Seidel method converges fast if the diffusion is missing or small. Sorting alone does not help, only the combination of sorting and using a block Gauss-Seidel method yields a fast solver.

The complexity of the ordering algorithm is linear in the number of degrees of freedom, in other words the order is optimal. Each iteration of the introduced block Gauss-Seidel method is as well linear in the number of degrees of freedom, which is also confirmed by the results.

Questions that remain are: Up to which strength of diffusion is the block Gauss-Seidel approach appropriate in general? How much faster would the block Gauss-Seidel approach with an explicit sorting of the system be? Does an additional mass term that enters for the non-steady problems hurt, in contrast to what we have seen so far?

Bibliography

[1] Brezzi, F., Marini, L. D., and Süli, E. Discontinuous galerkin methods for first-order hyperbolic problems. *Mathematical Models and Methods in Applied Sciences 14*, 12 (2004), 1893–1903.

[2] Hackbusch, W., and Probst, T. Downwind gauß-seidel smoothing for convection dominated problems. *Numerical linear algebra with applications 4*, 2 (1997), 85–102.

[3] Tarjan's strongly connected components algorithm. `http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm`.

[4] Tarjan, R. Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1*, 2 (1972), 146–160.