OpenFOAM's basic solvers for linear systems of equations

Solvers, preconditioners, smoothers

Tim Behrens*

February 18, 2009

*Vestas PhD student at Technical University of Denmark, tibeh@mek.dtu.dk

Contents

1. Introduction

This report gives some insight into OpenFOAM's structure of linear solvers, i.e. iterative solvers for linear sets of equations $Ax = b$. Also matrix preconditioners and smoothers will be presented. In a tutorial section we will use the icoFoam application solver on the cavity test case. A comparison between DIC and FDIC preconditioner as well as between PCG/PBiCG and GAMG solvers will be done. It will be shown how to copy the PBiCG solver and implement it as a myPBiCG solver to demonstrate how to add new solvers.
The report was prepared for the course *CFD with OpenSource software*[1] at Chalmers University of Technology. Some of the material and general ideas is collected from the official manuals, the forum and other course material[2]. Also the book of Saad, *Iterative methods for sparse linear systems*[3] - whose 1st edition is free to download - was used and is recommended for further information about the topic.

[1] http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2008/
[2] Martin van Gijzen, Iterative Methods for Linear Systems of Equations, http://ta.twi.tudelft.nl/nw/users/gijzen/CURSUS_DTU/HOMEPAGE/PhD_Course.html
[3] I.T. Distinguished Professor, University of Minnesota, http://www-users.cs.umn.edu/~saad/

2. Structure in OpenFOAM

First the file structure of the linear solvers in OpenFOAM shall be presented. The discussed solvers all operate on the lduMatrix class, which can be found in:

`$FOAM_SRC/OpenFOAM/matrices/lduMatrix/`

with its subdirectories being

- `lduAddressing/`
- `lduMatrix/`
- `preconditioners/`
- `smoothers/`
- `solvers/`

lduMatrix is a matrix class in which the coefficients are stored as three different arrays. One for the upper triangle (u), one for the lower triangle (l) and a third array for the diagonal of the matrix (d). Addressing arrays (in `lduAdressing/`) must be supplied for the upper and lower triangles. The fvScalarMatrix (`$FOAM_SRC/finiteVolume/fvMatrices/fvScalarMatrix`) class for example is derived from fvMatrix (`$FOAM_SRC/finiteVolume/fvMatrices/fvMatrix`), which itself is derived from lduMatrix. Inside of `lduMatrix.H/C` you can find the member functions

- `const scalarField & lower () const`
- `const scalarField & diag () const`
- `const scalarField & upper () const`

which will return the list of coefficients for the lower, diagonal and upper part of the matrix. Then for the source terms b (or RHS) `fvMatrix` implements:

`Field<type> & source ()`

In general one can specify a `lduMatrix` with whatever addressing one may wish, but one will have to build the addressing oneself. Thus, if corner neighbours are needed for some reason, this is no longer a standard finite volume method and the FVM machinery cannot be reused. Please note that the sparse matrix is not dynamic, ie. its sparseness pattern is defined at creation and cannot be changed.

2.1. Preconditioners

A preconditioned iterative solver solves the system

$$M^{-1}Ax = M^{-1}b$$

with M being the preconditioner. The chosen preconditioner should make sure that convergence for the preconditioned system is much faster than for the original one. This lead to M (mostly) being an easily invertible approximation to A. All operations with M^{-1} should be computational cheap. For multiplications with vectors the matrix-matrix multiplication $M^{-1}A$ has not to be calculated explicitly, but can be substituted by two matrix-vector calculations. The above example was a left preconditioning, but also central and right precondition exist. In simple terms the preconditioner leads to a faster propagation of information through the computational mesh.
In the `preconditioners/` directory one can find

- `diagonalPreconditioner/` - Diagonal preconditioner for both symmetric and asymmetric matrices. This preconditioner actually does not help with faster propagation through the grid, but it is very easy and can be a good first step. Note: The reciprocal of the diagonal is calculated and stored for reuse because on most systems multiplications are faster than divisions.

- `DICPreconditioner/` - Simplified diagonal-based incomplete Cholesky preconditioner for symmetric matrices (symmetric equivalent of DILU). The reciprocal of the preconditioned diagonal is calculated and stored.

- `DILUPreconditioner/` - Simplified diagonal-based incomplete LU preconditioner for asymmetric matrices. The reciprocal of the preconditioned diagonal is calculated and stored.

- `FDICPreconditioner/` - Faster version of the DICPreconditioner diagonal-based incomplete Cholesky preconditioner for symmetric matrices (symmetric equivalent of DILU) in which the reciprocal of the preconditioned diagonal and the upper coefficients divided by the diagonal are calculated and stored.

- `GAMGPreconditioner/` - Geometric agglomerated algebraic multigrid preconditioner (also named *Generalised geometric-algebraic multi-grid* in the manual).

- `noPreconditioner/` - Null preconditioner for both symmetric and asymmetric matrices.

2.2. Linear Solvers

In the `solvers/` directory one will find the following linear solvers

- `BICCG/` - Diagonal incomplete LU preconditioned BiCG solver[1]
- `diagonalSolver/` - diagonal solver for both symmetric and asymmetric problems
- `GAMG/` - Geometric agglomerated algebraic multigrid solver (also named *Generalised geometric-algebraic multi-grid* in the manual)
- `ICC/` - Incomplete Cholesky preconditioned Conjugate Gradients solver[2]
- `PBiCG/` - Preconditioned bi-conjugate gradient solver for asymmetric lduMatrices using a run-time selectable preconditioner
- `PCG/` - Preconditioned conjugate gradient solver for symmetric lduMatrices using a run-time selectable preconditiioner
- `smoothSolver/` - Iterative solver using smoother for symmetric and asymmetric matrices which uses a run-time selected smoother

2.2.1. Krylov Subspace solvers

The standard solvers PBiCG and PCG are Krylov subspace solvers. Instead of a full description[3] only a brief overview close to the Wikipedia entry shall be given:
In linear algebra, the order-r Krylov subspace generated by an n-by-n matrix A, and a vector of n-dimension b, is the linear subspace spanned by the images of b under the first r powers of A (starting from A0 = I), that is:

$$\mathcal{K}_r(A, b) = \text{span}\,\{b, Ab, A^2b, \ldots, A^{r-1}b\}$$

Modern iterative methods for solving large systems of linear equations avoid matrix-matrix operations, but rather multiply vectors by the matrix and work with the resulting vectors. Starting with a vector b, one computes Ab, then one multiplies that vector by A to find A^2b and so on. All algorithms that work this way are referred to as *Krylov subspace methods*. They are among the most successful methods currently available in numerical linear algebra. Because the vectors tend very quickly to become almost linearly dependent, methods relying on Krylov subspace frequently involve some orthogonalization scheme, such as *Lanczos* iteration for Hermitian matrices or *Arnoldi* iteration for more general matrices.
The best known Krylov subspace methods are the *Arnoldi*, *Lanczos*, *GMRES* (generalized minimum residual) and *BiCGSTAB* (stabilized biconjugate gradient) methods.

[1] This solver is present for backward-compatibility and the PBiCG solver should be used for preference
[2] This solver is present for backward-compatibility and the PCG solver should be used for preference
[3] M. Gutknecht, ETH Zurich, "'A Brief Introduction to Krylov Space Methods for Solving Linear Systems"', http://www.sam.math.ethz.ch/ mhg/pub/biksm.pdf

2.2.2. A closer look at GAMG

The basic idea behind multi-grid solvers is to use a coarse grid with fast solution times to smoothen out high frequency errors and to generate a starting solutions for the finer grid. This can either be done be a geometric coarsening of the grid (geometric multi-grid), or by applying the same principles directly to the matrix, regardless of the geometry (algebraic multi-grid). Inside GAMG the mesh is coarsed in steps and the coarsening or agglomeration algorithm can be faceAreaPair (geometric) or algebraic pair (see extract from ReleaseNote 1.4.1 below).

```
    Generalised geometric/algebraic multi-grid (GAMG) solver
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    [...]
    - The GAMG solver now completely replaces the old AMG solver which is no longer
      supported.  The GAMG solver can operate in a similar manner to the old AMG
      solver by selecting the 'algebraicPair' agglomerator although the
      'faceAreaPair' has proved superior in all of our tests.
    [...]
```

The main class for the faceAreaPair agglomeration can be found in

```
$FOAM_SRC/finiteVolume/fvMatrices/solvers/GAMGSymSolver/ \
GAMGAgglomerations/faceAreaPairGAMGAgglomeration$
```

In the `faceAreaPairGAMGAgglomeration.C` we find

```
    Foam::faceAreaPairGAMGAgglomeration::faceAreaPairGAMGAgglomeration
[...]
        //agglomerate(mesh, sqrt(fvmesh.magSf().internalField()));
        agglomerate
        (
            mesh,
            mag
            (
                cmptMultiply
                (
                    fvmesh.Sf().internalField()
                   /sqrt(fvmesh.magSf().internalField()),
                    vector(1, 1.01, 1.02)
                    //vector::one
                )
            )
        );
    }
```

The class calls the agglomerate process handing over the mesh and the magnitude of a multiplication of the face areas with a vector that has *not* unit components (as the part that was commented out would have had virtually). The vector was chosen presumably to have dominant agglomeration directions for meshes with uniform face areas. The choice of the scalar that is passed to the agglomeration is the main difference to the algebraicPair option which passes the mesh and the magnitude of the matrix' upper coefficients (for symmetric cases) to the agglomeration process:

`$FOAM_SRC/OpenFOAM/matrices/lduMatrix/solvers/GAMG/GAMGAgglomerations \`
`/algebraicPairGAMGAgglomeration/algebraicPairGAMGAgglomeration.C:`

```
Foam::algebraicPairGAMGAgglomeration::algebraicPairGAMGAgglomeration
(
    const lduMatrix& matrix,
    const dictionary& controlDict
)
:
    pairGAMGAgglomeration(matrix.mesh(), controlDict)
{
    agglomerate(matrix.mesh(), mag(matrix.upper()));
}
```

In `lduMatrix/solvers/GAMG/GAMGAgglomerations/pairGAMGAgglomeration` the agglomeration of the cells using the pair algorithm can be found with some comments on the process.
In `pairGAMGAgglomeration.C` one can see that the scalarField is named faceWeights which - as seen above - can either be the faceAreas or the matrix coefficients:

```
Foam::tmp<Foam::labelField> Foam::pairGAMGAgglomeration::agglomerate
(
    label& nCoarseCells,
    const lduAddressing& fineMatrixAddressing,
    const scalarField& faceWeights
)
```

The following is an overview of what happens in the loops:

1. Get the finest-level interfaces from the mesh
2. Start agglomeration from the given faceWeights
   a) For each cell calculate faces, afterwards go through the faces and create groups/clusters
      - Check faces to find ungrouped neighbour with largest face weight
      - Check if current cell is face owner or neighbour

- When a match is found, pick up all the necessary data and generate a new group/-cluster, else find the best neighbouring cluster and add the cell to it

b) Check that all cells are part of clusters, if not create a single-cell "cluster" for each

c) Reverse the map ordering to potentially improve the next level of agglomeration

3. Agglomerate the faceWeights field for the next level and continue from (1.) unless the user specified approximate mesh size at the most coarse level `nCoarsestCells` or the maximum number of grid levels `maxLevels` (hard coded to 50) is reached.

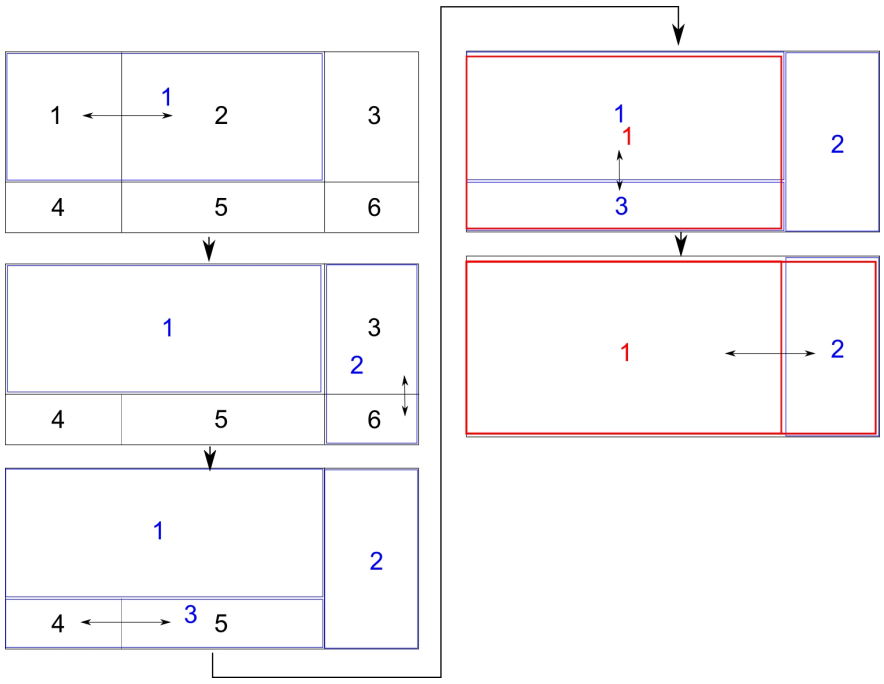

Figure 2.1.: Example of a geometric agglomeration process

In Fig. 2.1 a simple geometric example for this process is shown. Starting with the black grid consiting of 6 cells, the coarser mesh (blue) is built by joining two cells for each new cell. For the coarsest level (red) first cells 1 and 3 are joined. Cell 2 cannot find an ungrouped partner and will be joined with the neighbouring group. The mergeLevels keyword controls the step size at which coarsening or refinement of levels is performed. Assuming we would have used `mergeLevels 2` the blue mesh would have not been stored for calculation. It is often best to coarsen only by one level at a time, i.e. set `mergeLevels 1`. In some cases, particularly for simple meshes, the solution can be safely speeded up by coarsening/refining two levels at a time.
The solver is running a V-cycle at which the coarsest level matrix is solved directly (specifying `directSolve_Coarsest true`) or using the iterative ICCG/BICCG[4] as default. The number of sweeps used by the selected smoother when solving at different levels of mesh density are specified

[4]solver will be chosen automatically for symmetric/asymmetric matrices

by the `nPreSweeps`, `nPostSweeps` and `nFinestSweeps` keywords. The `nPreSweeps` entry is used when the V cycle is moving in coarser direction, `nPostSweeps` is used as the algorithm is refining again. The `nFinestSweeps` parameter is used when the solution is at its finest level. The user is only required to specify an approximate mesh size at the most coarse level in terms of the number of cells `nCoarsestCells`. The levels in between will be generated by the solver agglomeration.
The defaults for the GAMG solver are set in `GAMGSolver.C` and can be overwritten by definition inside `fvSolution`.

```
cacheAgglomeration_(false),
nPreSweeps_(0),
nPostSweeps_(2),
nFinestSweeps_(2),
scaleCorrection_(matrix.symmetric()),
directSolveCoarsest_(false),
```

MGridGen

The UserGuide states that there is an MGridGen[5] option that requires an additional entry specifying the shared object library for MGridGen:

```
geometricGamgAgglomerationLibs ("libMGridGenGamgAgglomeration.so");
```

The source codes can be found in
`$FOAM_SRC/decompositionAgglomeration/MGridGenGamgAgglomeration`.

[5]MGridGen is a parallel library written entirely in ANSI C that implements (serial) algorithms for obtaining a sequence of successive coarse grids that are well-suited for geometric multigrid methods. The quality of the elements of the coarse grids is optimized using a multilevel framework. See http://www-users.cs.umn.edu/~moulitsa/software.html

2.3. Smoothers

Although the preconditioners discussed before can considerably reduce the number of iterations, they do not normally reduce the mesh dependency of the numbers of iterations. OpenFOAM supplies the following smoothers to be used with the solvers in the `smoothers/` directory:

- `DIC/` - Simplified diagonal-based incomplete Cholesky smoother for symmetric matrices.
- `DICGaussSeidel/` - Combined DIC/GaussSeidel smoother for symmetric matrices in which DIC smoothing is followed by GaussSeidel to ensure that any "spikes" created by the DIC sweeps are smoothed-out.
- `DILU/` - Simplified diagonal-based incomplete LU smoother for asymmetric matrices. ILU smoothers are good smoothers for linear multigrid methods.
- `DILUGaussSeidel/` - Combined DILU/GaussSeidel smoother for asymmetric matrices in which DILU smoothing is followed by GaussSeidel to ensure that any "spikes" created by the DILU sweeps are smoothed-out.
- `GaussSeidel/` - The GaussSeidel method is a technique used to solve a linear system of equations. The method is an improved version of the Jacobi method. It is defined on matrices with non-zero diagonals, but convergence is only guaranteed if the matrix is either diagonally dominant, or symmetric and positive definite (spd).

3. Tutorial

In this tutorial we will use the icoFoam application solver on the cavity test case and

- compare DIC with FDIC preconditioner
- compare PCG/PBiCG with GAMG solver
- copy PBiCG and implement as myPBiCG solver

3.1. Cavity case

First one should make sure to have a working version of the cavity test case by copying a fresh version to the user's run directory.

```
cp -r $FOAM_TUTORIALS/icoFoam/cavity $FOAM_RUN/cavity
cd $FOAM_RUN/cavity
```

The `system/fvSolution` file should now look like below, which could be called a standard configuration.

```
solvers
{
    p PCG
    {
        preconditioner   DIC;
        tolerance        1e-06;
        relTol           0;
    };

    U PBiCG
    {
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0;
    };
}
```

3.2. Preconditioner Test

In the preconditioner section one could see that there exists a faster version (**F**DIC) of the DIC preconditioner in which the reciprocal of the preconditioned diagonal and the upper coefficients divided by the diagonal are calculated and stored. We will now try to see how large the speed-up actually is for our test case. We change the mesh size in `constant/polyMesh/blockMeshDict` to 150*150.

```
blocks
(
    hex (0 1 2 3 4 5 6 7) (150 150 1) simpleGrading (1 1 1)
);
```

Changing the `system/controlDict` to

```
endTime         0.04;
deltaT          0.0005;
writeControl    timeStep;
writeInterval   20;
```

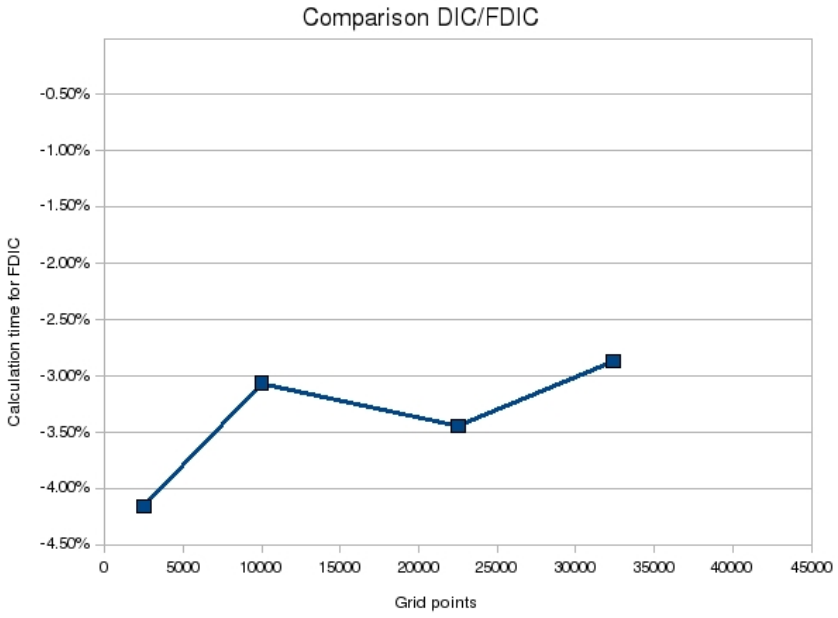

Figure 3.1.: Calculation time speed-up for FDIC compared to DIC

generating the new mesh with blockMesh and running the icoFoam application solver.

```
blockMesh
icoFoam > logDIC
```

When icoFoam has completed we change DIC to FDIC in `fvSolution` file

```
    preconditioner   FDIC;
```

and run icoFoam again with output to a new log file

```
icoFoam > logFDIC
```

The result - which should be reproducable on most machines - is that the execution time for the FDIC preconditioner is 3-4% lower. Results for some different mesh sizes are presented in Fig. 3.1. Adding just one letter to your fvSolution file may result in a speed-up of about 3%. On the other hand the FDIC calculation for the configuration crashed at a grid size of 200*200, while the DIC still finished normally (of course one could adjust the time-stepping).

3.3. Geometric agglomerated algebraic multigrid

We will now compare solving the cavity test case with the PCG/PBiCG and the multigrid solver. If you have not run the above DIC/FDIC case, first run icoFoam on the test case with the above `blockMeshDict`, `controDict` and `fvSolution` that utilizes Krylov subspace solvers (KSS) .

```
icoFoam > logKSS &
```

When icoFoam has finished we change `fvSolution` to make use of the multi grid solver implemented in OpenFOAM:

```
solvers
{
    p GAMG
    {
        preconditioner    FDIC;
        mergeLevels       1;
        smoother          GaussSeidel;
        agglomerator      faceAreaPair;
        nCellsInCoarsestLevel   100;
        tolerance         1e-05;
        relTol            0;
    };

    U GAMG
    {
        preconditioner    DILU;
        mergeLevels       1;
        smoother          GaussSeidel;
        agglomerator            faceAreaPair;
        nCellsInCoarsestLevel   100;
        tolerance         1e-05;
        relTol            0;
    };
}
```

Running icoFoam now the execution time should be around one third of the calculation utilizing PCG/PBiCG (with same tolerances!).

```
icoFoam > logGAMG
```

3.4. Adding your own solver

In general there are two ways to add a new solver hard-linking or linking through a dynamic library. The shorter version (hard-linking) will be discussed in the following. We will add the solver directly to our application solver of choice myIcoFoam.
Make a copy of the PBiCG solver

```
cp -r $FOAM_SRC/OpenFOAM/matrices/lduMatrix/solvers/PBiCG $FOAM_RUN/myPBiCG
```

Replace all occurences of PBiCG inside `PBiCG.H/.C` with myPBiCG and rename these files to `myPBiCG.H/.C`

```
sed s/PBiCG/myPBiCG/g <PBiCG.C >myPBiCG.C
sed s/PBiCG/myPBiCG/g <PBiCG.H >myPBiCG.H
rm PBiCG.*
```

It is particularly important that the new solver adds itself to the list of allowed symmetric/asymmetric solvers in `myPBiCG.C`, otherwise the application solver will not be able to select it.

```
   namespace Foam
   {
        defineTypeNameAndDebug(myPBiCG, 0);

        lduMatrix::solver::addasymMatrixConstructorToTable<myPBiCG>
            addmyPBiCGAsymMatrixConstructorToTable_;
    }
```

Copy the icoFoam solver

```
cp -r $FOAM_APP/solvers/incompressible/icoFoam \
  $FOAM_RUN/myIcoFoam

cd $FOAM_RUN/myIcoFoam
```

and change your `Make/files` to

```
    icoFoam.C
    $(FOAM_RUN)/myPBiCG/myPBiCG.C
    EXE = $(FOAM_USER_APPBIN)/myIcoFoam
```

compile

```
wclean
rm -r Make/linux*
wmake
```

and myPBiCG will be included automatically.
Now copy and switch into your favourite test case directory

```
cp -r $FOAM_TUTORIALS/icoFoam/cavity $FOAM_RUN/cavity
cd $FOAM_RUN/cavity
```

edit your system/fvSolution so your solver for velocity U is now myPBiCG

```
    U myPBiCG
    {
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0;
    };
```

and start your calculation after building the mesh

```
blockMesh
myIcoFoam > log &
```

Check that the code uses myPBiCG by looking at the log file.

```
    DILUmyPBiCG:  Solving for Ux, [...]
    DILUmyPBiCG:  Solving for Uy, [...]
    FDICPCG:  Solving for p, [...]
```

A. Comments

A.1. Induced Dimension Reduction Method

The Induced Dimension Reduction (IDR) method is a Krylov subspace method for solving large sparse nonsymmetric systems of linear equations method and was recently revived as IDR(s)[1] by *Sonneveld* and *van Gijzen*[2]. In Fig. A.1 solutions of an ocean circulation test problem[3] are shown. One can see that the IDR(4) algorithm requires considerably less matrix-vector computations than BiCG and BiCGSTAB. Unfortunatly implementation of IDR(s) could not be done within the scope

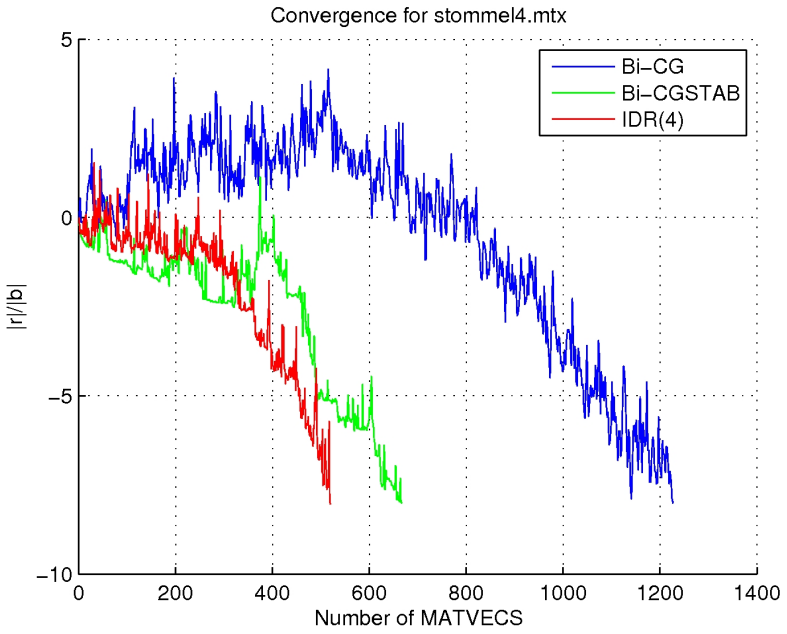

Figure A.1.: Solutions of an ocean circulation test problem

of this work, but it sure is a very interesting method and could contribute to OpenFOAM development.

A.2. LIS

If one is interested in different solvers one should have a look at the Lis (Library of Iterative Solvers for Linear Systems) project[4] which is a library written in C and Fortran 90 for solving linear equations and eigenvalue problems with iterative methods.

[1] http://ta.twi.tudelft.nl/nw/users/gijzen/IDR.html
[2] Delft University of Technology
[3] M. B. van Gijzen, C. B. Vreugdenhil, and H. Oksuzoglu, The Finite Element Discretization for Stream-Function Problems on Multiply Connected Domains, J. Comp. Phys., 140, 1998, pp. 30-46.
[4] Scalable Software Infrastructure for Scientific Computing http://ssi.is.s.u-tokyo.ac.jp/lis/

LSODE (Livermore Solver for Ordinary Differential Equations) is the basic solver of the collection. It solves stiff and nonstiff systems of the form dy/dt = f. In the stiff case, it treats the Jacobian matrix df/dy as either a dense (full) or a banded matrix, and as either user-supplied or internally approximated by difference quotients. It uses Adams methods (predictor-corrector) in the nonstiff case, and Backward Differentiation Formula (BDF) methods (the Gear methods) in the stiff case. The linear systems that arise are solved by direct methods (LU factor/solve). LSODE supersedes the older Among the iterative methods for solving large linear systems with a sparse (or, possibly, structured) nonsymmetric matrix, those that are based on the Lanczos process feature short recurrences for the generation of the Krylov space. This means low cost and low memory requirement. This review article introduces the reader not only to the basic forms of the Lanczos process and some of the related theory, but also describes in detail a number of solvers that are based on it, including those that are considered to be the most efficient ones. Possible breakdowns of the algorithms and ways to cure t