Linear solvers & preconditioners

GOFUN 2018, Braunschweig

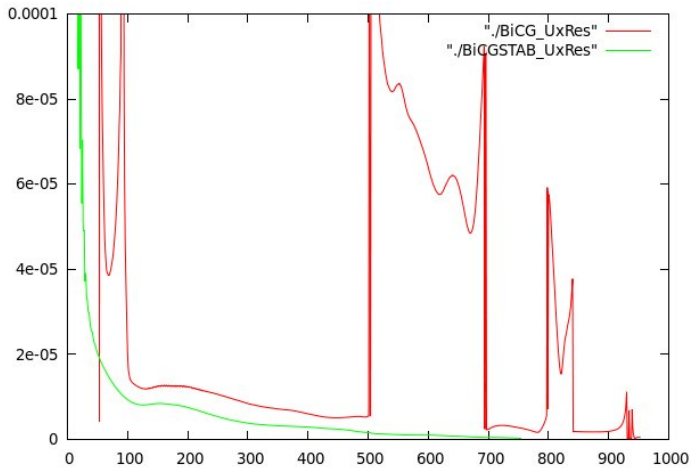
Dr. Thorsten Grahs

Feb. 22, 2018

Matrix solver

In this talk we cover

- Assembling the matrix
- Correspondence between mesh & matrix
- Class `lduMatrix`
- Solver in OpenFOAM
- Krylov sub-space methods
- BiCG method
- Tutorial `BiCGSTAB`



Matrix solver

- Results directly from the discretisation of the governing equations
- Assembled as
  - `fvScalarMatrix`
  - `fvVectorMatrix`
  - `fvTensorMatrix`
- Example

```
fvVectorMatrix UEqn
(
    fvm::ddt(U)
  + fvm::div(phi, U)
  - fvm::laplacian(nu, U)
);
```

Matrix solver

- An instance of the class
  - `fvScalarMatix` (e.g. for `pEqn`) or
  - `fvVectorMatrix` (e.g. for `UEqn`)

  is built, which represents the matrix
  (and also rhs, preconditioner and solver)
- The coefficients results from the discretisation of the `fvm` operators
  - diagonal entries f.i. from the temporal discretisation
  - off-diagonal entries from the fluxes from the neighbouring cells
  - Right hand side from source terms (`fvc`-operators) or old time steps

Solving an equation

- Solving an equation in OpenFOAM always follows the same pattern:
  - Initialisation
  - Assembling the matrix
  - Solving the system
  - Correcting the solution
- In the following, we consider the approach exemplified by the solver `icoFoam`

Matrix creation & initialisation

Example icoFoam

```
 #include "fvCFD.H"
// * * * * * * * * * * * * * * * * * * * * * *//
int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    #include "initContinuityErrs.H"

    Info<< "\nStarting time loop\n" << endl;

    while (runTime.loop())
    {
        Info<< "Time = " << runTime.timeName()
                         << nl << endl;
        #include "readPISOControls.H"
        #include "CourantNo.H"

        fvVectorMatrix UEqn
        (
            fvm::ddt(U)
          + fvm::div(phi, U)
          - fvm::laplacian(nu, U)
        );
        solve(UEqn == -fvc::grad(p));
```

- Creating the instance `fvVectorMatrix UEqn` from the class fvVectorMatrix
- Initialisation with `fvm::ddt(U) + fvm::div(phi,U) - fvm::laplacian(nu,U)`
- All operators are from `fvm` namespace (implicit), i.e. all contribute to the matrix
- Solved with rhs `-fvc::grad(p)` (explicit)
- Correcting the solution with the PISO-Loop

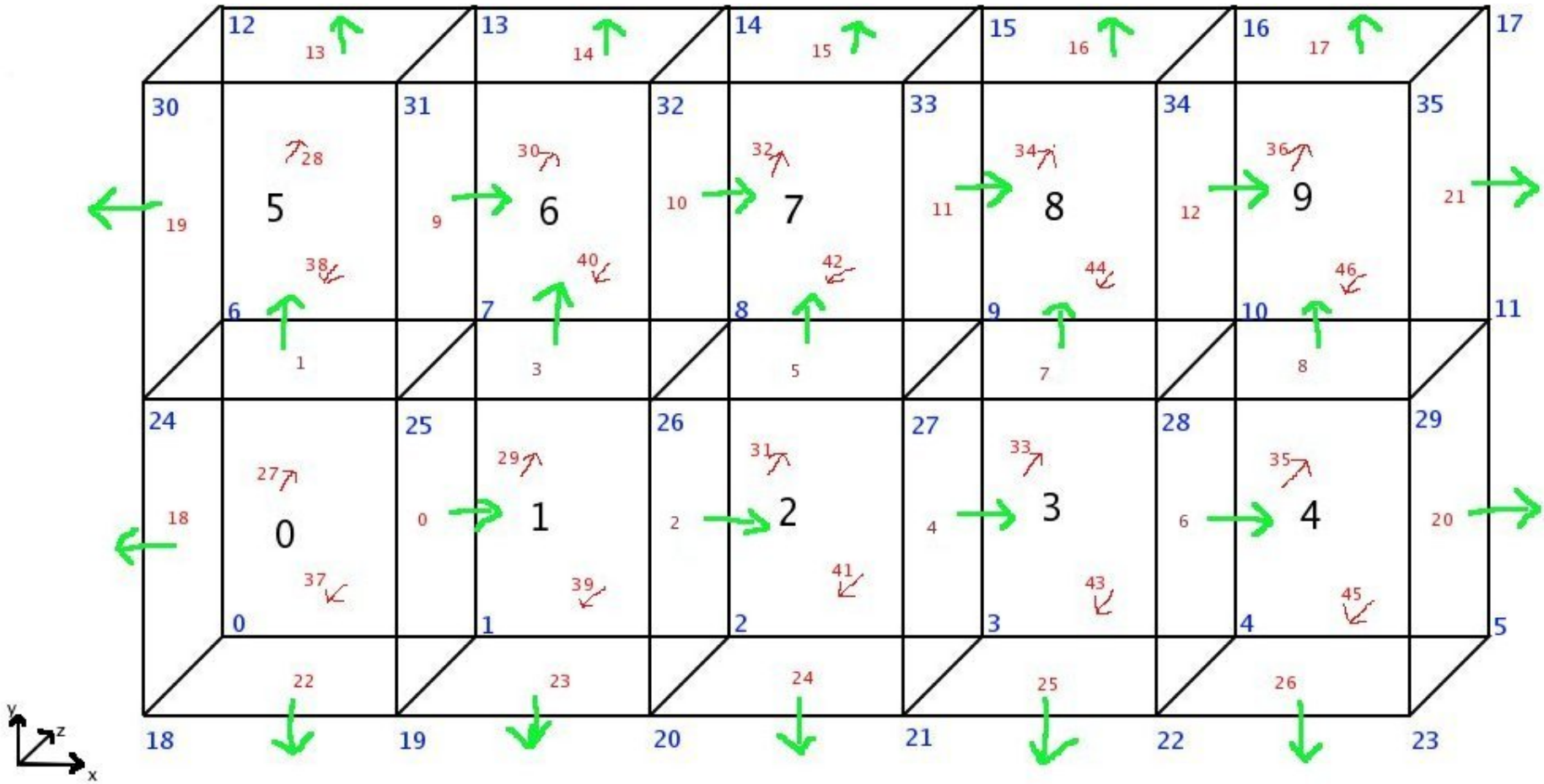
Mesh & matrix structure

Correspondence

$$\text{Mesh} \quad \Leftrightarrow \quad \mathbf{A}\,\mathbf{u} = \mathbf{b}$$

- Obviously, there is a strong correspondence between mesh and matrix
- The number of cells corresponds with the rank n of the matrix $\mathbf{A}$
- And so naturally with the dimension of the solution vector $\mathbf{u}$.
- The entries of the matrix stems from the weights of the numerical schemes
- In general, $\mathbf{A}$ is a sparse matrix

Mesh example



Relation Matrix & Mesh

- The number of diagonal elements equals the number of cells of the mesh.
- The number of non-zero upper and lower elements equal the number of internal faces of the mesh.
- The upper matrix contains values belonging to the owners of faces.
- The lower matrix contains values belonging to the neighbour cells.
- Off-diagonal matrix element contain a non-zero value if a face is shared between an owner and neighbour cell on that position (in the mesh)

Matrix structure

Matrix according to the mesh example

- 10 volumes/cells
- 36 nodes/corners
- 47 faces, 13 internal faces

$$\begin{pmatrix}
a_{0,0} & a_{0,1} & 0 & 0 & 0 & a_{0,5} & 0 & 0 & 0 & 0 \\
a_{1,0} & a_{1,1} & a_{1,2} & 0 & 0 & 0 & a_{1,6} & 0 & 0 & 0 \\
0 & a_{2,1} & a_{2,2} & a_{2,3} & 0 & 0 & 0 & a_{2,7} & 0 & 0 \\
0 & 0 & a_{3,2} & a_{3,3} & a_{3,4} & 0 & 0 & 0 & a_{3,8} & 0 \\
0 & 0 & 0 & a_{4,3} & a_{4,4} & 0 & 0 & 0 & 0 & a_{4,9} \\
a_{5,0} & 0 & 0 & 0 & 0 & a_{5,5} & a_{5,6} & 0 & 0 & 0 \\
0 & a_{6,1} & 0 & 0 & 0 & a_{6,5} & a_{6,6} & a_{6,7} & 0 & 0 \\
0 & 0 & a_{7,2} & 0 & 0 & 0 & a_{7,6} & a_{7,7} & a_{7,8} & 0 \\
0 & 0 & 0 & a_{8,3} & 0 & 0 & 0 & a_{8,7} & a_{8,8} & a_{8,9} \\
0 & 0 & 0 & 0 & a_{9,4} & 0 & 0 & 0 & a_{9,8} & a_{9,9}
\end{pmatrix}$$

lduMatrix

Matrix according to the mesh example

- The system, resulting from the FV discretisation, is stored as type `lduMatrix`
- `fvMatrix` is a sub class from `lduMatrix`
- In general we deal with sparse matrices.
- The non-zero coefficients are stored in three arrays:
  - `lower()`
  - `diag()`
  - `upper()`
- For our example we have the following entries for the arrays:

$$lower() = (a_{1,0}, a_{5,0}, a_{2,1}, a_{6,1}, a_{3,2}, a_{7,2}, a_{4,3}, a_{8,3}, a_{9,4}, a_{6,5}, a_{7,6}, a_{8,7}, a_{9,8})$$

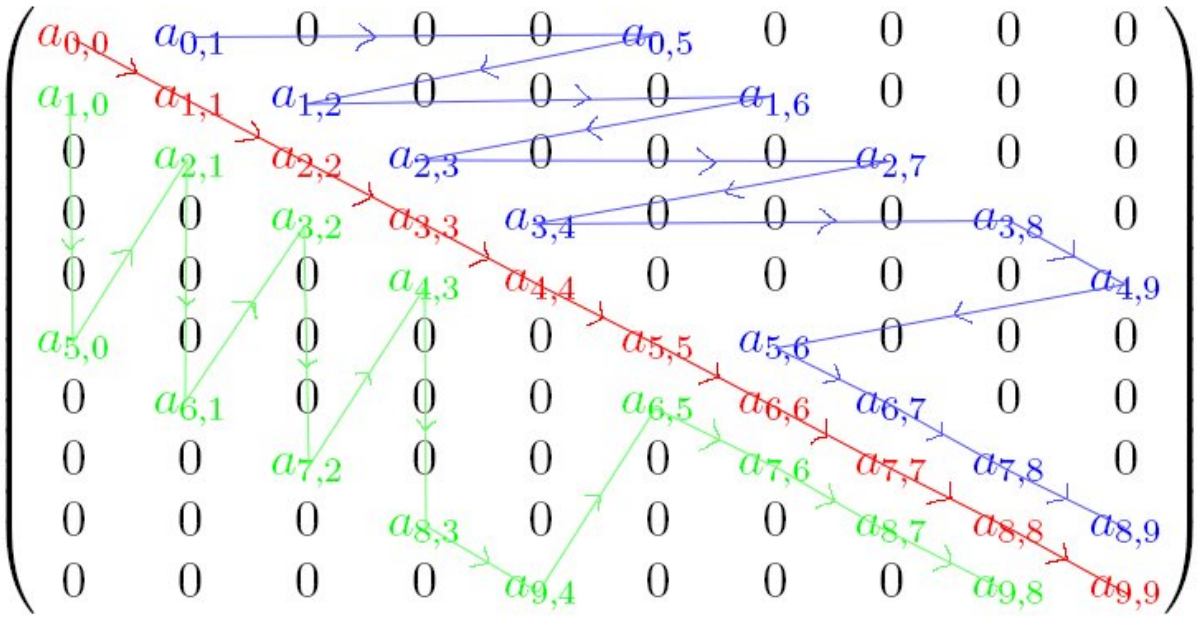
$$diag() = (a_{0,0}, a_{1,1}, a_{2,2}, a_{3,3}, a_{4,4}, a_{5,5}, a_{6,6}, a_{7,7}, a_{8,8}, a_{9,8})$$

$$upper() = (a_{0,1}, a_{0,5}, a_{1,2}, a_{1,6}, a_{2,3}, a_{2,7}, a_{3,4}, a_{3,8}, a_{4,9}, a_{5,6}, a_{6,7}, a_{7,8}, a_{8,9})$$

Run order

Matrix according to the mesh example

- The order of accessing matrix elements in OpenFOAM is as follows:



lower(), diag(), upper()

Addressing

- The addressing for the non-zero coefficients are stored in three lists of the class `lduAdressing`:
  - `lowerAddr()`
    Keeps the row number of the according element in array `lower()` and the column number for array `upper()`
  - `upperAddr()`
    Vice versa, i.e. keeps column number for `lower()` and row number for `upper()` entries
  - `ownerStartAddr()`
    Keeps the information, at which position in
    - `upper()` a new row starts
    - `lower()` a new column starts

Addressing

Matrix according to the mesh example

- The address arrays for the example matrix looks as follows:
  - `upperAddr()`
    `= (0, 0, 1, 1, 2, 2, 3, 3, 4, 5, 6, 7, 8)`
  - `lowerAddr()`
    `= (1, 5, 2, 6, 3, 7, 4, 8, 9, 6, 7, 8, 9)`
  - `ownerStartAddr() = (0, 2, 4, 6, 8, 9, 10, 11, 12)`
- `lduMatrix`
  - `lower()`, `diag()`, `upper()` and `lduAdressing` are function of the matrix class `lduMatrix`.
  - `lduMatrix` has further functions for solving algorithms and statistics i.e. `solver`, `solverPerformance`, `smoother` & `preconditioner`
- Class definition can be found at

  `$FOAM_SRC/OpenFOAM/matrices/lduMatrix`

Linear system solver functions

Solver sub-classes in `lduMatrix`

- `lduMatrix::solverPerformance` (solver statistics)
  Class which is returned by each matrix solver. Keeps informations about the solver characteristics, e.g.
  - solver name
  - field/variable name
  - Initial residual
  - Final residual
  - Number of iterations
- Sub-classes `solver`, `smoother` & `preconditioner` are virtual base classes:
  - They define an interface, which has to be implemented by a concrete matrix solver
  - This allows run-time selection for the matrix solver
  - An implementation of a concrete matrix solver algorithm has to be derived from the class `solver`

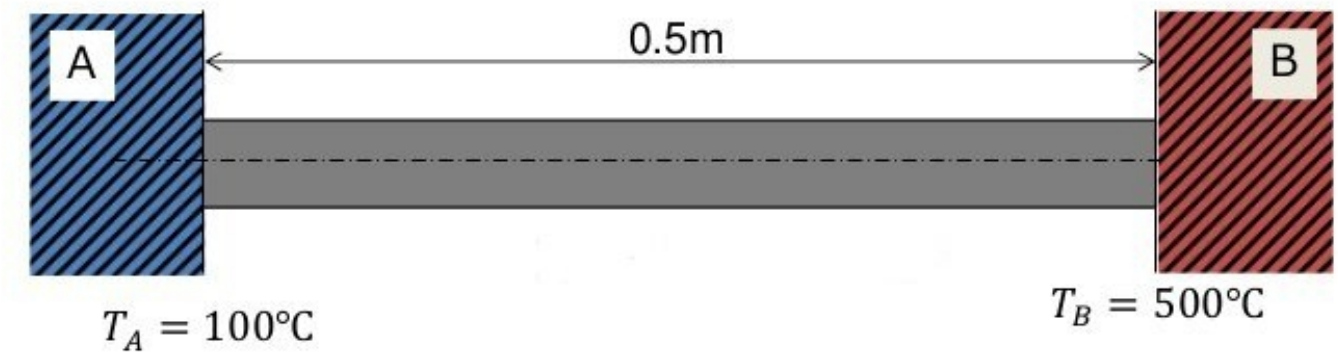
Matrix manipulation

- `fvMatrix` defines function for accessing matrix elements
  - `tmp<volScalarField> D()` returns diagonal elements
  - `tmp<volScalarField> A()` returns diagonal elements weighted with inverse of cell volume
  - `tmp<volScalarField> H()` returns off-diagonal elements
- The assembled matrix correspondence directly with the `fvm` operators
- This enables the construction of the matrix system directly from the implementation of the equation in top-level code

$$\texttt{fvm::ddt(U)} + \texttt{fvm::div(phi,U)} \Rightarrow \quad \mathbf{Au} = \mathbf{b}$$

Test case

- Consider heat conduction in a thin rod



- The temperature distribution is governed by a Laplace-equation (From Fourier's Law)

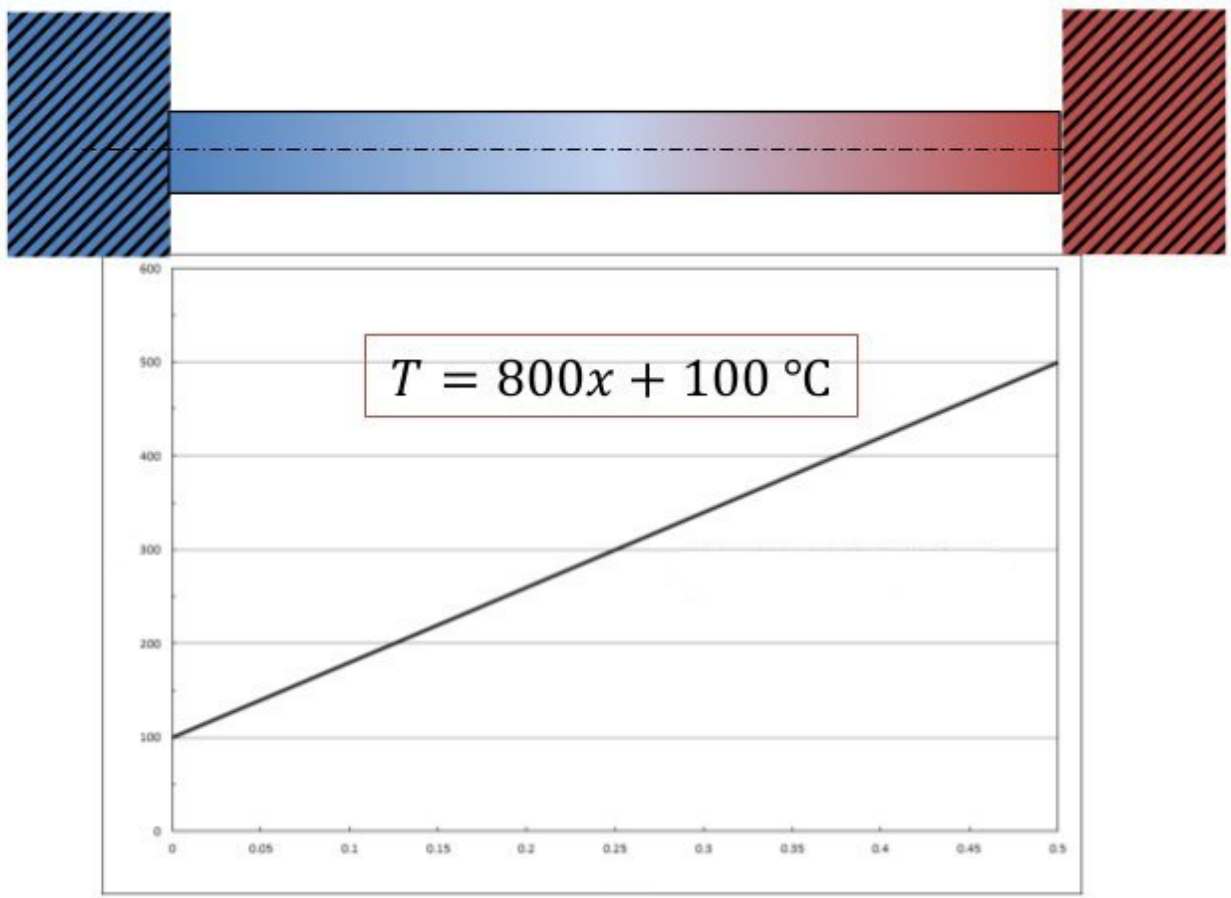
$$\frac{d}{dx}\left(k\frac{dT}{dx}\right) = 0$$

- Diameter $D = 0.001m^2$, Conductivity $k = 1000Wm^{-1}K^{-1}$

Test case

Expectations

What do we expect?



- Linear distribution inside the rod.

myLaplacianFoam

- We change the original OpenFOAM solver `laplacianFoam` to a customized (simplified) version: myLaplacianFoam

- Clone solver

```
> run
> mkdir programming
> cd programming
> cp -r $FOAM_SOLVERS/basic/laplacianFoam myLaplacianFoam
```

- Adapt source code & make-files in `myLaplacianFoam/Make/files`:

```
laplacianFoam.C                          ⇒   mylaplacianFoam.C
EXE = $(FOAM_APPBIN)/laplacianFoam           EXE = $(FOAM_USER_APPBIN)/mylaplacianFoam
```

Adapt code

myLaplacianFoam.c

- Change name

```
> mv laplacianFoam.C myLaplacianFoam.C
```

- In myLaplacianFoam.C change

```
while (simple.loop())
{
    Info<< "Time = " << runTime.timeName()
        << nl << endl;

    while (simple.correctNonOrthogonal())
    {
        solve
       (
          fvm::ddt(T)
          - fvm::laplacian(DT, T)
       );
    }
   #include "write.H"
   Info<< "ExecutionTime = "
            << runTime.elapsedCpuTime()
            << " s"
            << "  ClockTime = "
            << runTime.elapsedClockTime()
            << " s"
            << nl << endl;
    }
```

$\Rightarrow$

```
while (runTime.loop())
{
   fvScalarMatrix
       TEqn(fvm::laplacian(k, T));
       TEqn.solve();

   forAll(T, cellI)
   {
     Info<< "X = " <<
     mesh.C()[cellI].component(vector::X)
     << ", T = " << T[cellI] << endl;
   }

   runTime.writeAndEnd();
}
```

Adapt code

createFields.C

- In `create.C` change

```
Info<< "Reading diffusivity DT\n" << endl;

dimensionedScalar DT
(
   transportProperties.lookup("DT")
);
```

⇒

```
Info<< "Reading diffusivity k\n" << endl;

dimensionedScalar k
(
   transportProperties.lookup("k")
);
```

- Compile the modified code:

```
> cd myLaplacianFoam
> wmake
```

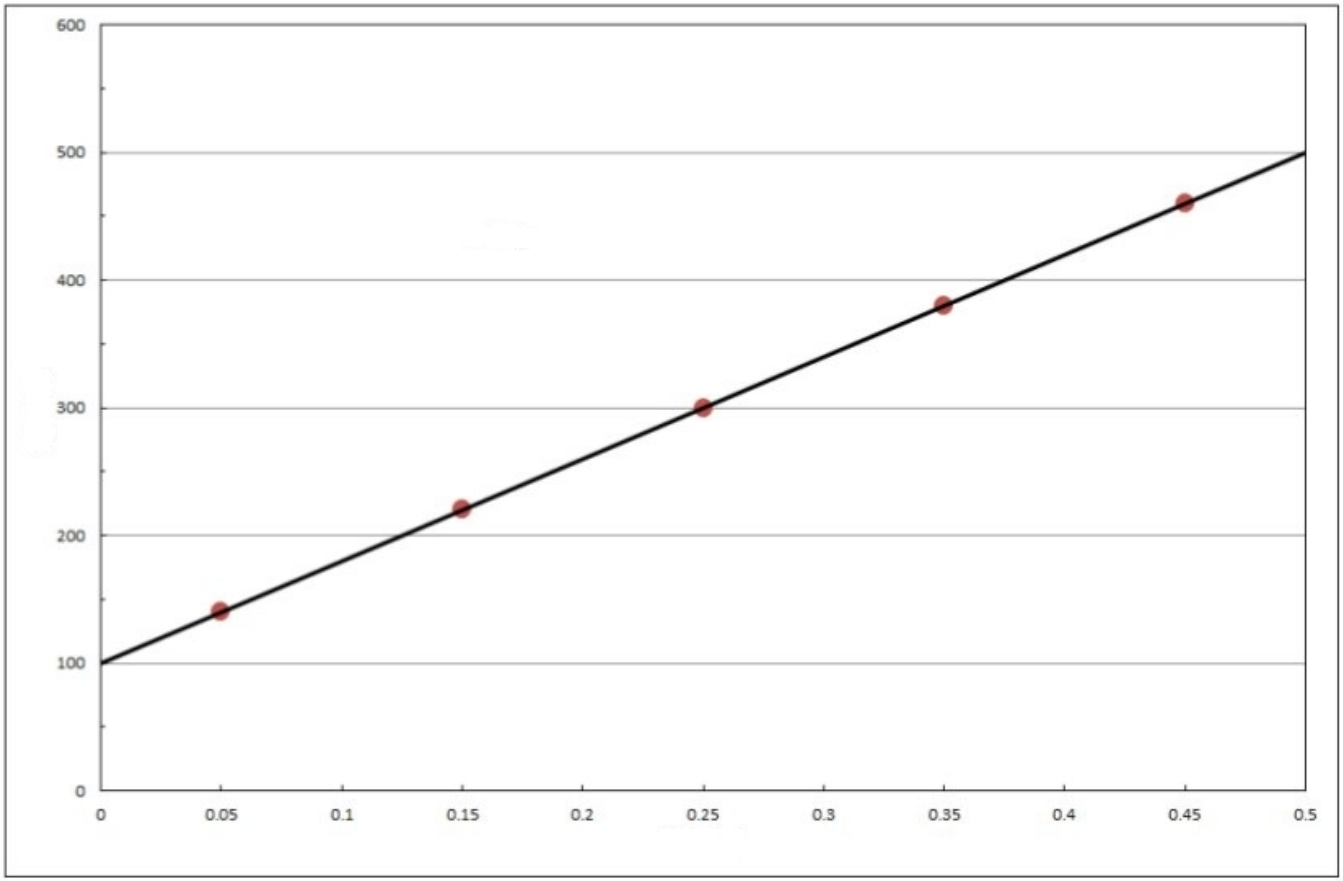
- Run the test case

```
> cd rodTemperature
> blockMesh
> myLaplacianFoam
...
Calculating temperature distribution

DICPCG: Solving for T,Initial residual = 1,Final residual = 1.21266e-16,No Iterations 1
X = 0.05, T = 140
X = 0.15, T = 220
X = 0.25, T = 300
X = 0.35, T = 380
X = 0.45, T = 460
End
```

Result

Result in accordance with our expectations



Extension

Change in `myLaplacianFoam.C`

```
#include "fvCFD.H"
#include "simpleMatrix.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    Info<< "\nCalculating temperature distribution\n" << endl;

   while (runTime.loop())
     {
       fvScalarMatrix TEqn(fvm::laplacian(k, T));
       TEqn.solve();
       forAll(T, cellI)
         {
           Info<< "X = " <<
             mesh.C()[cellI].component(vector::X)
               << ", T = " << T[cellI] << endl;
         }
    #include "writeMatCoeffs.H"
```

Adapt code

Create `writeMatCoeffs.C`

- Investigating the coefficient matrix

```
label NC = mesh.nCells(); //Number of cells
simpleMatrix<scalar> A(NC); //Coeff.matrix
// Initialization of matrix
  for(label i=0; i<NC; i++)
  {
      A.source()[i] = 0.0;
      for(label j=0; j<NC; j++)
      {
         A[i][j] = 0.0;
      }
  }
// Assigning diagonal coefficients
  for(label i=0; i<NC; i++)
  {
      A[i][i] = TEqn.diag()[i];
  }
// Assigning off-diagonal coefficients
  for(label faceI=0;
     faceI<TEqn.lduAddr().lowerAddr().size(); faceI++)
  {
    label l = TEqn.lduAddr().lowerAddr()[faceI];
    label u = TEqn.lduAddr().upperAddr()[faceI];
    A[l][u] = TEqn.upper()[faceI];
    A[u][l] = TEqn.upper()[faceI];
  }
```

```
// Assigning contribution from BC
  forAll(T.boundaryField(), patchI)
  {
    const fvPatch &pp =
         T.boundaryField()[patchI].patch();
    forAll(pp, faceI)
    {
       label cellI = pp.faceCells()[faceI];
       A[cellI][cellI]
    += TEqn.internalCoeffs()[patchI][faceI];
       A.source()[cellI]
    += TEqn.boundaryCoeffs()[patchI][faceI];
    }
  }
  Info  << "\n==Coefficients of Matrix A=="
        << endl;
  for(label i=0; i<NC; i++)
  {
    for(label j=0; j<NC; j++)
    {
      Info<< A[i][j] << " ";
    }
    Info<< A.source()[i] << endl;
  }
  Info<< "\n==> Solution: " << A.solve()
      << endl;
```

Running adapted code

- Recompile and rerun test case `rodTemperature`

Solver output

```
 Calculating temperature distribution

DICPCG:  Solving for T, Initial residual = 1, Final residual = 1.21266e-16, No Iterations 1
X = 0.05, T = 140
X = 0.15, T = 220
X = 0.25, T = 300
X = 0.35, T = 380
X = 0.45, T = 460

==Coefficient Matrix==
-300 100 0 0 0 -20000
100 -200 100 0 0 0
0 100 -200 100 0 0
0 0 100 -200 100 0
0 0 0 100 -300 -100000
Solution: 5(140 220 300 380 460)
End
```

Linear systems

Iterative solvers

- The system of equation in OpenFOAM is solved by iterative matrix algorithms
  - Based on the starting vector $\mathbf{x}^0$ the iteration rule

$$\mathbf{x}^{k+1} = \mathbf{A}\mathbf{x}^k$$

    is applied
  - The iteration stops, when the residual is smaller than a prescribed tolerance `tol`:

$$\|\mathbf{A}\mathbf{x}^{k+1} - \mathbf{b}\| < \texttt{tol}$$

- Sometimes one tries to accelerate the solution behaviour of the iterative process.
- This goal can be achieved by *pre-conditioning*

Preconditioning

Goal

- Enhance matrix properties in order to optimize condition number ($cond(\mathbf{A})$ close to 1)

  $\Rightarrow$ faster convergence of the solver
- One is looking for a matrix $\mathbf{M}$ with

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}$$

  and $\mathbf{M}$ easy to invert.
- The easiest choice would be $\mathbf{M} = \mathbf{I}$
  (To simple - yields just the original system)
- The best choice would be $\mathbf{M} = \mathbf{A}^{-1}$
  (To costly to invert $\mathbf{A}$)

Preconditioner in OpenFOAM

- `diagonalPreconditioner`
  for symmetric & nonsymmetric matrices (not very effective)
- `DICPreconditioner`
  Diagonal Incomplete Cholesky preconditioner
  for symmetric matrices
- `DILUPreconditioner`
  Diagonal Incomplete LU preconditioner
  for nonsymmetric matrices
- `FDICPreconditioner`
  Fast Diagonal Incomplete Cholesky preconditioner
- `GAMGPreconditioner`
  Geometric Agglomerated algebraic MultiGrid preconditioner
- `noPreconditioner`

Preconditioners

- Diagonal preconditioner $\mathbf{M} = \mathbf{D}$
  Uses the diagonal as preconditioner. Not very efficient.
- Incomplete LU preconditioner $\mathbf{A} = \mathbf{LU} + \mathbf{R}, \quad \mathbf{M} = \mathbf{LU}$
  - Apply Gaussian elimination algorithm, but only on allowed pattern $\Rightarrow$ incomplete LU factorization called ILU.
  - Reduce in the for-loops the indices to the indices with
    – allowed pattern, e.g. ILU(0) for pattern of $\mathbf{A}$
    – values that are not too small, ILUT for ILU with treshold
  - Leads to approximate LU factorization

$$\mathbf{A} = \mathbf{LU} + \mathbf{R}, \quad \text{preconditioner } \mathbf{M} = \mathbf{LU}$$

  with all ignored fill-in entries collected in $\mathbf{R}$.
- Incomplete Cholesky preconditioner $\mathbf{A} = \mathbf{GG}^T + \mathbf{R}, \quad \mathbf{M} = \mathbf{GG}^T$
  Similar to ILU, but based on incompleted Cholesky factorization

Matrix solver in OpenFOAM

- `BICCG`
  Diagonal incomplete LU preconditioned BiCG solver
- `diagonalSolver`
  Solver for symmetric and nonsymmetric matrices
- `GAMG`
  Geometric Agglomerated algebraic Multi-Grid solver
- `ICC`
  Incomplete Cholesky Conjugate Gradient solver
- `PBiCG`
  Bi-Conjugate Gradient solver with preconditioner
- `PCG`
  Conjugate Gradient solver with preconditioner
- `smoothSolver`
  Iterative solver with run-time selectable smoother

Most of these solvers are *Krylov-subspace* solvers

Krylov subspace methods

- A Krylov[1] subspace method is an iterative matrix algorithm for solving the linear system $\mathbf{Ax} = \mathbf{b}$ of the form:

  $$\mathbf{x}^m \in \mathbf{x}^0 + K_m, \quad K_m = K_M(\mathbf{A}, \mathbf{r}_0) = \text{span}(\mathbf{r}^0, \mathbf{Ar}^0, \ldots, \mathbf{A}^{m-1}\mathbf{r}^0)$$

  with

  $$\mathbf{r}^0 = \mathbf{b} - \mathbf{Ax}^0, \qquad (\mathbf{b} - \mathbf{Ax}^m) \perp L_m, \qquad K_m, L_m \in \mathbb{R}^m \subset \mathbb{R}^n$$

- K_m represents the *Krylov subspace* where the solution is sought.

Why is this desirable?

[1] Aleksey Nikolaevich Krylov (1863 - 1945), russian naval engineer and mathematician

Krylov method

Motivation

- Idea of a Krylov method stems from the fact that
  - the inverse of a $N \times N$ non-singular Matrix $\mathbf{A}$ can be expressed by a polynomial in $\mathbf{A}$ of degree $(N-1)$.
  - i.e.
$$\mathbf{A}^{-1} = c_0\mathbf{I} + c_1\mathbf{A} + c_2\mathbf{A}^2 + \ldots + c_{N-1}\mathbf{A}^{N-1}$$
  - This is a direct conclusion from the *Cayley-Hamilton theorem*
- Since the true solution $\mathbf{x}^\star$ is

$$\mathbf{x}^\star = \mathbf{A}^{-1}\mathbf{b} = c_0\mathbf{b} + c_1\mathbf{A}\mathbf{b} + c_2\mathbf{A}^2\mathbf{b} + \ldots + c_{N-1}\mathbf{A}^{N-1}\mathbf{b} = p(\mathbf{A})$$

  with polynomial $p(\mathbf{A}) = \sum_{i=0}^{N-1} c_i\mathbf{A}^i\mathbf{b}$
- Hence we can approximate the solution $\mathbf{x}^\star = \mathbf{A}^{-1}\mathbf{b}$ by applying a polynomial with $m \ll n$
- We only need matrix-vector multiplication!

Krylov method

The iteration approach

Krylov subspace methods for the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

attempt to approximate the solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ with

$$\mathbf{x}^j \in \mathbf{x}^0 + K_j, \quad K_j(\mathbf{A}, \mathbf{r}_0), \quad \mathbf{r}^0 = \mathbf{b} - \mathbf{A}\mathbf{x}^0.$$

The specific choice of the polynomial (i.e. the coefficients) depends on the concrete method.

Krylov methods

Remarks

- If **A** has a dominant eigenvector, then it is likely that

$$W_j = [\mathbf{b}, \mathbf{Ab}, \mathbf{A}^2\mathbf{b}, \ldots, \mathbf{A}^{j-1}\mathbf{b}]$$

  is very, very ill conditioned for large values of j.
- In general, one needs orthonormal vectors $\{\mathbf{v}_i\}_{i=0}^{j}$ such that

$$K_j(\mathbf{A}, \mathbf{b}) = \text{span}[\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_j]$$

  in order to avoid numerical problems.
- The vectors $\mathbf{v}_i$ can be computed using an standard orthogonalization approach, f.i. the Arnoldi algorithm.

Krylov methods

Standard methods

- **CG**: The Conjugate Gradient algorithm applies to systems where **A** is symmetric positive definite (SPD)
- **GMRES**: The Generalized Minimal RESidual algorithm is the first method to try if **A** is not SPD.
- **BiCG**: The BiConjugate Gradient algorithm applies to general linear systems, but the convergence can be quite erratic.
- **BiCGstab**: The stabilized version of the BiConjugate Gradient algorithm.

Remark

- We focus her on CG and BiCG. In the tutorial we show how to extend BiCG to BiCGstab.

The Conjugated Gradient Method (CGM)

- CG is the most popular iterative method for solving large systems of linear equation of the form

$$\mathbf{Ax} = \mathbf{b}$$

  where $\mathbf{A} \in \mathbb{R}^{n \times n}$ be symmetric positive definite and $\mathbf{b} \in \mathbb{R}^n$.
- If **A** would be dense, the best way would be to factor (LU, QR,..) and solve by back-substitution
- For sparse systems CG is your method.

Conjugated Gradients

- Associated with the system $\mathbf{Ax} = \mathbf{b}$ we have a quadratic function, f , given by

$$f(\mathbf{x}) := \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{x}^T\mathbf{b}$$

- The most instructive explanation of the CG method starts from the fact, that (if $\mathbf{A}$ is symmetric positive definite) the solution of

$$\mathbf{Ax} = \mathbf{b}$$

minimizes the quadratic function f.

Why????

Conjugated Gradients

- To this uses we calculate the first and second order derivatives of f:

$$\begin{aligned} \nabla f(\mathbf{x}) &= \mathbf{A}\mathbf{x} - \mathbf{b} \\ \mathbf{H}_f(\mathbf{x}) &= \mathbf{A} \end{aligned}$$

  with
  - the gradient $\nabla f = (\partial_1 f, \ldots, \partial_n f)$
  - the Hessian $\mathbf{H}_f = (\partial^2_{ij} f)_{ij}$.
- Since all higher order derivatives vanish, we can rewrite f as its Taylor polynomial.

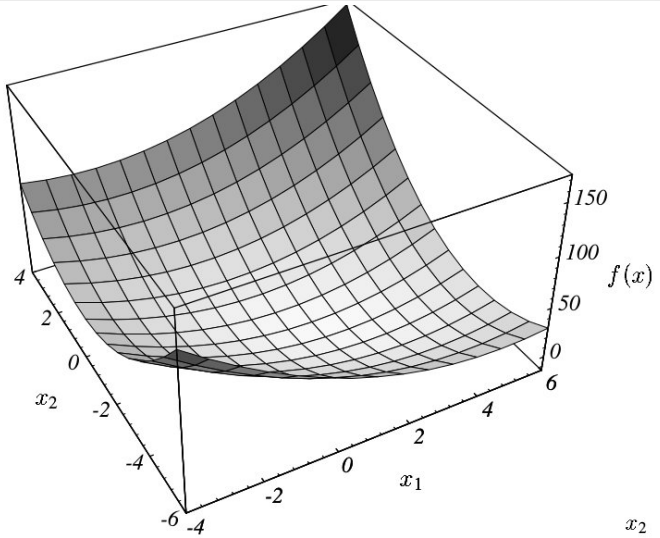
Taylor polynomial

- We rewrite f as its Taylor polynomial of degree two, i.e.

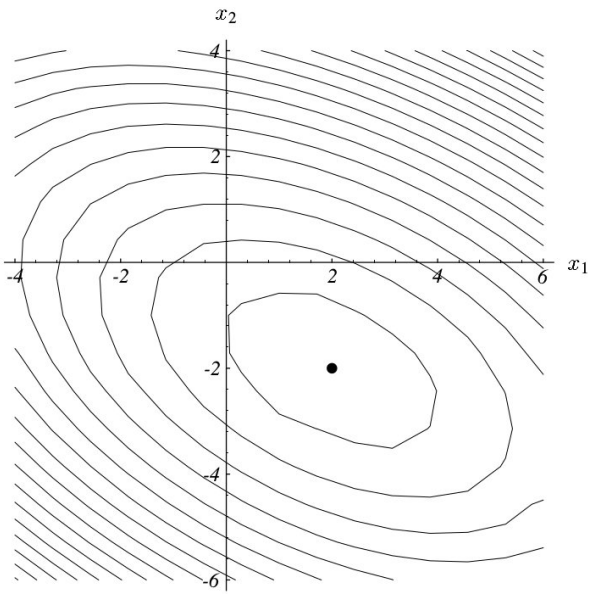
$$\begin{aligned} f(\mathbf{x}) &= f(\mathbf{y}) - (\mathbf{x}-\mathbf{y})^T \nabla f(\mathbf{y}) + \frac{1}{2}(\mathbf{x}-\mathbf{y})^T \mathbf{H}_f(\mathbf{y})(\mathbf{x}-\mathbf{y}) \\ &= f(\mathbf{y}) - (\mathbf{x}-\mathbf{y})^T (\mathbf{A}\mathbf{y}-\mathbf{b}) + \frac{1}{2}(\mathbf{x}-\mathbf{y})^T \mathbf{A}(\mathbf{x}-\mathbf{y}) \\ &\geq f(\mathbf{y}) - (\mathbf{x}-\mathbf{y})^T (\mathbf{A}\mathbf{y}-\mathbf{b}) \end{aligned}$$

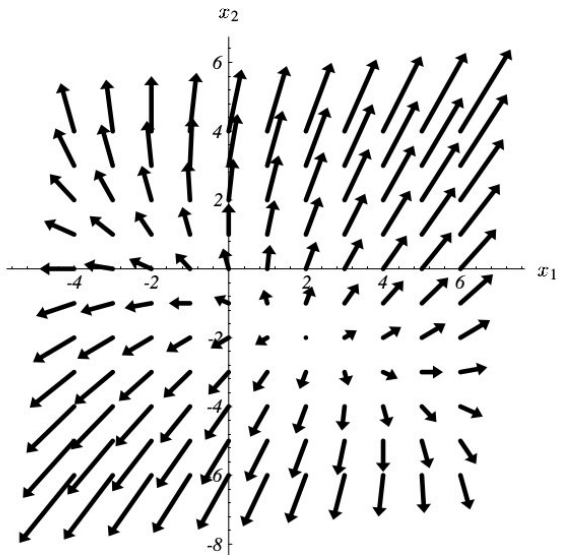
- In the last step, we have used that $\mathbf{A}$ is positive definite.
- If $\mathbf{y} = \mathbf{x}^*$ solves $\mathbf{A}\mathbf{x} = \mathbf{b} \Rightarrow f(\mathbf{x}) \geq f(\mathbf{x}^*)$, i.e. $\mathbf{x}^*$ minimises f .
- On the other hand, every minimiser $\mathbf{y}$ of f has to satisfy the necessary condition $0 = \nabla f(\mathbf{y}) = \mathbf{A}\mathbf{y} - \mathbf{b}$.
- This is only possible for $\mathbf{y} = \mathbf{x}^*$

Example: Graph, Contours Gradient of f



- Here we show the graph of a quadratic form f.
- The minimum point of this surface is $\mathbf{x}$ with $\mathbf{Ax} = \mathbf{b}$.
- Contours of f are ellipsoidals (i.e. f=const.)





Constructing an iterative method

- Now we construct an iterative method which tries to find the minimum of f.
- We start from the iteration rule

$$\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j$$

with search direction $\mathbf{p}_j$ and step length α_j

Question?

- How to chose α.
- How to choose $\mathbf{p}_j$

Determine step length α

Assume for a moment that we already have chosen the search direction $\mathbf{p}_j$.

- With given $\mathbf{p}_j$ the best possible step-length can be determined by finding the minimum of f along $\mathbf{x}_j + \alpha\mathbf{p}_j$.
- Setting $\psi(\alpha) = f(\mathbf{x}_j + \alpha\mathbf{p}_j)$ the necessary condition for a minimum is

$$\begin{aligned} 0 = \psi'(\alpha) &= \mathbf{p}_j^T \nabla f(\mathbf{x}_j + \alpha\mathbf{p}_j) \\ &= \mathbf{p}_j^T [\mathbf{A}(\mathbf{x}_j + \alpha\mathbf{p}_j) - \mathbf{b}] \\ &= \mathbf{p}_j^T \mathbf{A}\mathbf{x}_j + \alpha\mathbf{p}_j^T \mathbf{A}\mathbf{p}_j - \mathbf{p}_j^T \mathbf{b}. \end{aligned}$$

Step-length α_j

Theorem

Assume that the search direction $\mathbf{p}_j \neq 0$ has already been determined. Then we can compute the optimal step-length α for the new iteration

$$\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j$$

by

$$\alpha_j = \frac{\mathbf{p}_j^T(\mathbf{b} - \mathbf{A}\mathbf{x}_j)}{\mathbf{p}_j^T \mathbf{A}\mathbf{p}_j} = \frac{\mathbf{p}_j^T \mathbf{r}_j}{\mathbf{p}_j^T \mathbf{A}\mathbf{p}_j} \tag{1}$$

- Thus, we have to determine the search direction.

Search direction – 1st try

- Use the unity vectors $\mathbf{e}_i$ as search directions, i.e.

$$\mathbf{p}_0 = \mathbf{e}_1, \mathbf{p}_1 = \mathbf{e}_2, \ldots, \mathbf{p}_{n-1} = \mathbf{e}_n, \mathbf{p}_n = \mathbf{e}_1, \mathbf{p}_{n+1} = \mathbf{e}_2, \ldots$$

- Than we have

$$\mathbf{e}_i^T \mathbf{A}\mathbf{e}_i = a_{ii} \quad \text{and} \quad \mathbf{e}_i^T (\mathbf{A}\mathbf{x} - \mathbf{b})\mathbf{e}_i = \sum_{j=1}^{n} a_{ij} x_j - b_i$$

- Inserting (1) – the definition for α – yields

$$\mathbf{x}_k^k = \mathbf{x}^{k-1} + \alpha_{k-1}\mathbf{p}^{k-1} = \mathbf{x}^{k-1} - \frac{1}{a_{kk}} \left(\sum_{j=1}^{n} a_{kj} \mathbf{x}_j^{k-1} - b_l \right) \mathbf{e}_k$$

for $k = 0, 1, \ldots, n-1$.

Search direction – 1st try

Problematic:

- Only the k-th component of the vector $\mathbf{x}^k$ is updated.
- If we consider one cycle, we get

$$\mathbf{x}^k = \frac{1}{a_{kk}} \left(b_k - \sum_{j<k}^{n} a_{kj} x_j^k - \sum a_{kj} x_j^0 \right),$$

$$x_i^k = x_i^{k-1}, k \neq i$$

- Therefore, one cycle is one Gauss-Seidel iteration.

Search direction – 2nd try

- One natural way to choose the search direction is the direction of the steepest descent.
- This is given by the negative gradient of the target function:

$$\begin{aligned} \mathbf{p}_j &= -\nabla f(\mathbf{x}_j) \\ &= -(\mathbf{A}\mathbf{x}_j - \mathbf{b}) \\ &= \mathbf{r}_j \end{aligned}$$

Steepest Descent

Algorithm

Choose $\mathbf{x}_1$ and set $\mathbf{p}_1 = \mathbf{b} - \mathbf{A}\mathbf{x}_1$

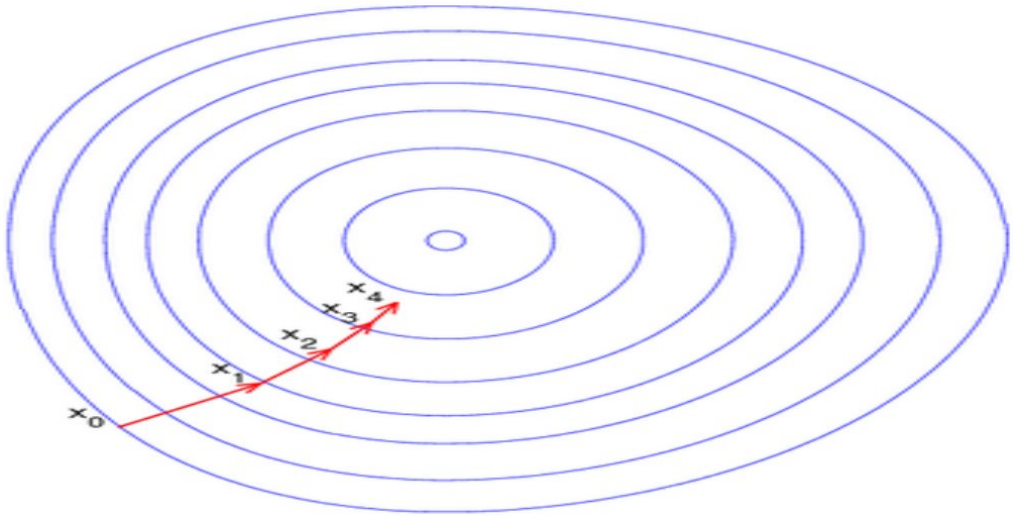
for $j = 1, 2, \ldots$ **do**

$$\alpha_j = \frac{\mathbf{p}_j^T \mathbf{r}_j}{\mathbf{p}_j^T \mathbf{A}\mathbf{p}_j}$$

$$x_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j$$

$$\mathbf{p}_{j+1} = \mathbf{b} - \mathbf{A}\mathbf{x}_{j+1}$$

end



Orthogonal search direction

- Notice that, by choosing α_j and $\mathbf{p}_j$ as constructed, we have

$$0 \quad = \quad \mathbf{p}_j^T(\mathbf{A}\mathbf{x}_{j+1} - \mathbf{b}) = -\mathbf{p}_j^T\mathbf{p}_{j+1}$$

- Thus, the derived search directions are orthogonal
- Furthermore, there are successive orthogonal.
- On a first glance ...
  - orthogonal search direction are very good.
- On a second view
  - steepest descent converges rather slowly.

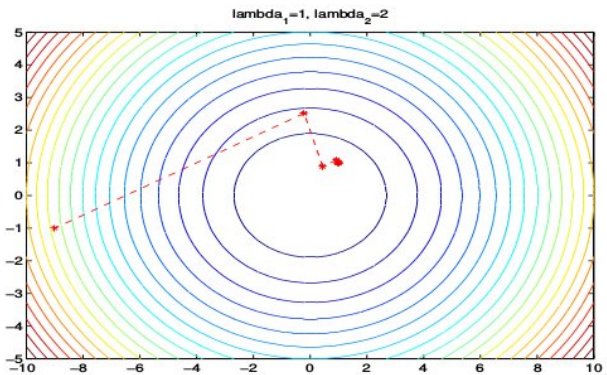
Steepest descent method

Bad if $\lambda_{max}/\lambda_{min}$ is large.

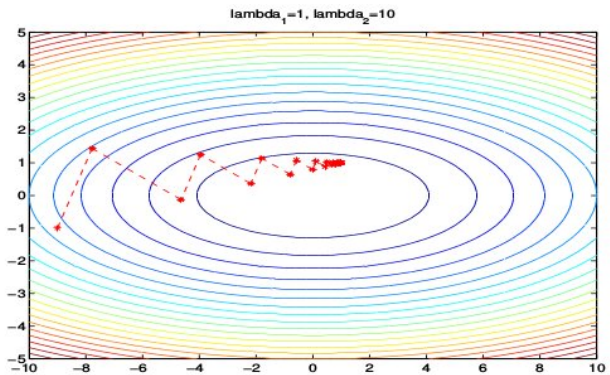
- Consider $\mathbf{Ax} = \mathbf{b}$ with

$$\mathbf{A} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix}.$$

- Use start vector $\mathbf{x}_0 = (-9, 1)^T$. The solution is $(1, 1)^T$.
- Looking for $\epsilon = \|\mathbf{r}_k\| < 10^{-4}$:



$\lambda_1 = 1, \lambda_2 = 2$
9 iterations



$\lambda_1 = 1, \lambda_2 = 10$
41 iterations

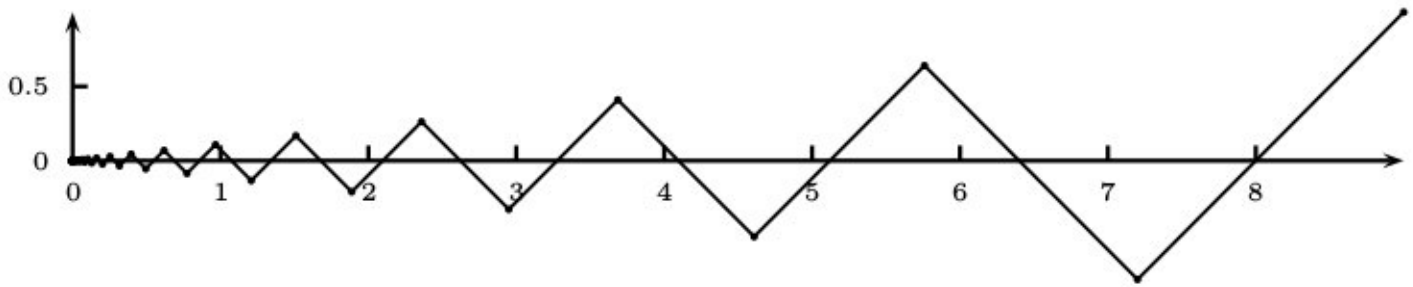
Example

Example Let us consider the following setting:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 9 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} 9 \\ 1 \end{pmatrix}.$$

Convergence history:

$$\mathbf{x}_j = (0.8)^{j-1} \begin{pmatrix} 9 \\ (-1)^{j-1} \end{pmatrix}$$



Search direction 3rd approach

Idea:

- Determine search direction $\mathbf{p}^k$ such, that $\mathbf{p}^k$ is w.r.t. the previous directions $\mathbf{p}^0, \mathbf{p}^1, \ldots \mathbf{p}^{k+1}$

Definition

Two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ are called A-conjugated if

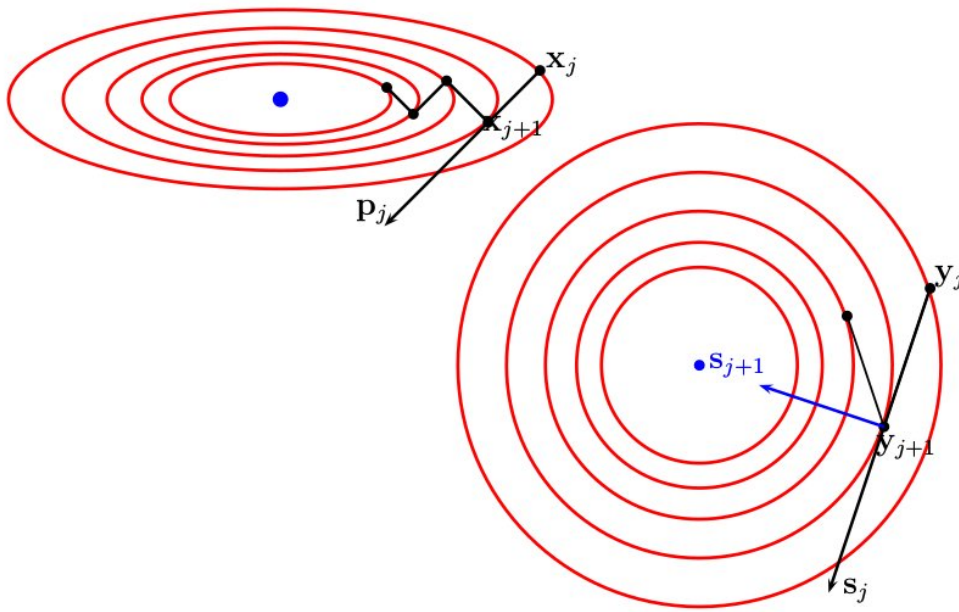
$$\mathbf{x}^T \mathbf{A} \mathbf{y} = 0$$

holds.

A-conjugates search directions

Level curves of $f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x} = \frac{1}{2}(x_1^2 + 9x_2^2)$

- The level curses of f are ellipses.
- The orthogonality forces the search direction tangential to the level sets
- If we introduce A-conjugated search directions, i.e. $\mathbf{s}_j = \mathbf{A}^{1/2}\mathbf{p}_j$ one finds the minimum in two steps.



We have $0 = \mathbf{s}_j^T\mathbf{s}_{j+1} = \mathbf{p}_j^T\mathbf{A}\mathbf{p}_{j+1}$

Conjugated Gradient (CG) method

Theorem

Assume $\mathbf{p}^0, \mathbf{p}^1, \ldots, \mathbf{p}^{n-1} \neq \mathbf{0}$ are pairwise A-conjugated vectors. Then, the scheme

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha_k \mathbf{p}^k$$

with

$$\alpha_k = -\frac{(\mathbf{r}^k)^T \mathbf{r}^k}{(\mathbf{p}^k)^T \mathbf{A}\mathbf{p}^k}$$

converges in at most n steps against the exact solution.

Conjugated Gradients

Algorithm: CG method (Hestenes & Stiefel 1952)

Choose $\mathbf{x}_0$

Set $\mathbf{p}_0 = \mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$ and $j = 0$.

while $\|r_j\|_2 > \varepsilon$ **do**

$$\mathbf{v}_j = \mathbf{A}\mathbf{p}_j, \quad \alpha_j = \frac{\mathbf{r}_j^T \mathbf{r}_j}{\mathbf{v}_j^T \mathbf{p}_j}$$

$$\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j$$

$$\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j \mathbf{v}_j$$

$$\beta_j = \frac{\|\mathbf{r}_{j+1}\|_2^2}{\|\mathbf{r}_j\|_2^2}$$

$$\mathbf{p}_{j+1} = \mathbf{r}_{j+1} + \beta_j \mathbf{p}_j$$

$$j = j + 1$$

end

Conjugated Gradients

Remarks

- It can be proved, that the vectors $\mathbf{p}^k$ are pair-wise A-conjugated.
- Theoretically, after at most n steps the solution can be computed. Due to rounding errors in practice you will not get the solution after n steps.
- In practice we have $n \gg 1$. Therefore, the CG method is used as an iterative method.
- In each iteration step there are
  - 1 matrix-vector product,
  - 2 scalar products
  - 3 scalar multiplications

  necessary.

CG as Krylov subspace method

Theorem

The k-th iteration $\mathbf{x}^k$ of the CG method minimizes the functional $f(\cdot)$ w.r.t. the subspace

$$\mathcal{K}_k(\mathbf{A}, \mathbf{r}^0) = span\{\mathbf{r}^0, \mathbf{A}\mathbf{r}^0, \mathbf{A}^2\mathbf{r}^0, \ldots, \mathbf{A}^{k-1}\mathbf{r}^0\},$$

i.e. there holds

$$f(\mathbf{x}^k) = \min_{c_i} f(\mathbf{x}^0 + \sum_{i=0}^{k-1} c_i \mathbf{A}^i \mathbf{r}^0).$$

The subspace $\mathcal{K}_k(\mathbf{A}, \mathbf{r}^0)$ is called the k-th Krylov subspace.

CG method: Error estimate

- The error $\mathbf{e}^k := \mathbf{x}^k - \mathbf{x}^*$ is measured in the energy norm

$$\|\mathbf{x}\|_A := (\mathbf{x}^T \mathbf{A} \mathbf{x})^{1/2}$$

We get the error estimate

Theorem

$$\|\mathbf{x}^k - \mathbf{x}^*\|_A \leq 2 \left(\frac{\sqrt{\kappa(\mathbf{A})} - 1}{\sqrt{\kappa(\mathbf{A})} + 1} \right)^k \|\mathbf{x}^0 - \mathbf{x}^*\|_A$$

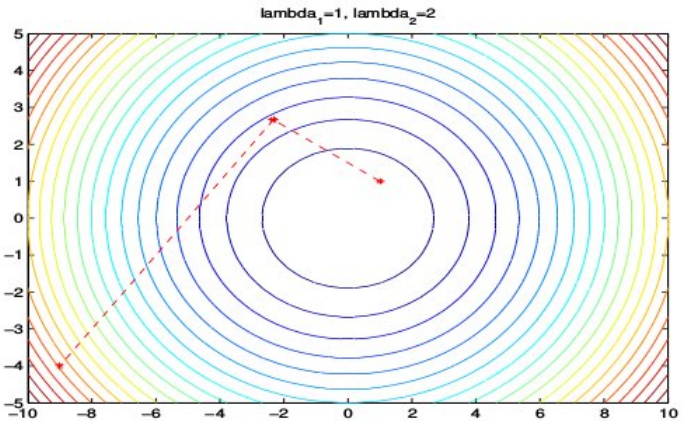
with $\kappa(\mathbf{A}) := cond_2(\mathbf{A}) = \frac{\lambda_{max(\mathbf{A})}}{\lambda_{min(\mathbf{A})}} \geq 1.$

CG method: Example

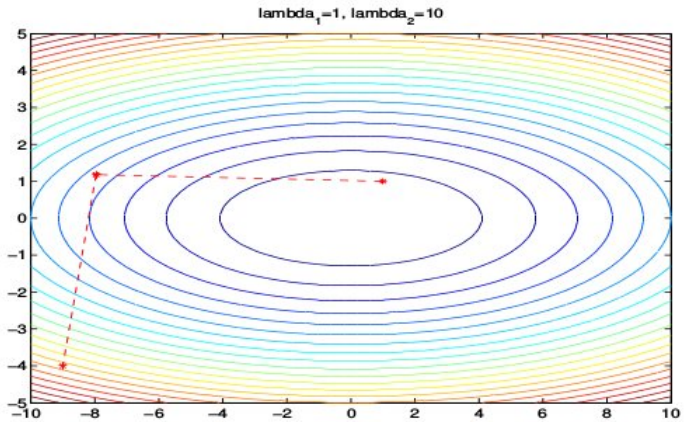
- Consider $\mathbf{Ax} = \mathbf{b}$ with

$$\mathbf{A} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix}.$$

- Use start vector $\mathbf{x}_0 = (-9, 1)^T$. The solution is $(1, 1)^T$.
- Looking for $\epsilon = \|\mathbf{r}_k\| < 10^{-4}$:



$\lambda_1 = 1, \lambda_2 = 2$
2 iterations



$\lambda_1 = 1, \lambda_2 = 10$
2 iterations

BiCG solver

Bi-Conjugated Gradient method

- The BiCG method belongs to the class of Krylov subspace methods.
- It represents an further development of the CG method.
- Results from the simultaneous evaluation of $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{A}^T\mathbf{x} = \mathbf{b}$
- In the subsequent tutorial we extend the BiCG method to BiCGstab
- Due to this intention we examine the implementation of the BiCG method in OpenFOAM.

BiCG solverPseudocode BiCG method

Choose $\psi_0 \in \mathbb{R}^n$ and $\epsilon > 0$
$\mathbf{r}_0 = \mathbf{p}_0 := \mathbf{b} - \mathbf{A}\psi_0$, $\mathbf{w}_0 := \mathbf{P}_L\mathbf{r}_0$, $\mathbf{r}_0^\star = \mathbf{p}_0^\star := \mathbf{b} - \mathbf{A}^T\psi_0$, $\mathbf{w}_0^\star := \mathbf{P}_L\mathbf{r}_0^\star$
$\mathbf{r}_1 = \mathbf{r}_0$, $\mathbf{r}_1^\star = \mathbf{r}_0^\star$, $j := 1$

While $\|\mathbf{r}_j\|_2 > \epsilon$

$$\mathbf{w}_j := \mathbf{P}_L\mathbf{r}_j, \quad \mathbf{w}_j^\star := \mathbf{P}_L\mathbf{r}_j^\star$$

$$\beta_j := \frac{(\mathbf{w}_j, \mathbf{r}_j^\star)_2}{(\mathbf{w}_{j-1}, \mathbf{r}_{j-1}^\star)_2}$$

$$\mathbf{p}_j := \mathbf{w}_j + \beta_j\mathbf{p}_{j-1}, \quad \mathbf{p}_j^\star := \mathbf{w}_j^\star + \beta_j\mathbf{p}_{j-1}^\star$$

$$\alpha_j := \frac{(\mathbf{w}_j, \mathbf{r}_j^\star)_2}{(\mathbf{A}\mathbf{p}_j, \mathbf{p}_j^\star)_2}$$

$$\mathbf{r}_{j+1} := \mathbf{r}_j - \alpha_j\mathbf{A}\mathbf{p}_j, \quad \mathbf{r}_{j+1}^\star := \mathbf{r}_j^\star - \alpha_j\mathbf{A}^T\mathbf{p}_j^\star$$

$$\psi_j := \psi_{j-1} + \alpha_j\mathbf{p}_j$$
$$j := j + 1$$

PBiCG solver code

PBICG.C

- The PBiCG method in OpenFOAM is a modified version. The order of calculation has been inverted.
- ψ_n in the code equates to the iteration vector $\mathbf{x}_n$

```
// --- Precondition residuals
preconPtr->precondition(wA, rA, cmpt);
preconPtr->preconditionT(wT, rT, cmpt);
...
// --- Update search directions:
        wArT = gSumProd(wA, rT);
scalar beta = wArT/wArTold;
```

$$\mathbf{w}_j := \mathbf{P}_L \mathbf{r}_j$$
$$\mathbf{w}_j^\star := \mathbf{P}_L \mathbf{r}_j^\star$$
$$\beta_j := \frac{(\mathbf{w}_j, \mathbf{r}_j^\star)_2}{(\mathbf{w}_{j-1}, \mathbf{r}_{j-1}^\star)_2}$$

PBiCG solver code

PBICG.C

```
for (register label cell=0;
                    cell<nCells; cell++)
{
    pAPtr[cell] = wAPtr[cell]
                + beta*pAPtr[cell];
    pTPtr[cell] = wTPtr[cell]
                + beta*pTPtr[cell];
}
// --- Update preconditioned residuals
matrix_.Amul(wA,pA interfaceBouCoeffs_,
                    interfaces_, cmpt);
matrix_.Tmul(wT,pT interfaceIntCoeffs_,
                    interfaces_, cmpt);
scalar wApT = gSumProd(wA, pT);
...
// --Update solution and residual:
   scalar alpha = wArT/wApT;
```

$$\mathbf{p}_j := \mathbf{w}_j + \beta_j \mathbf{p}_{j-1},$$
$$\mathbf{p}_j^\star := \mathbf{w}_j^\star + \beta_j \mathbf{p}_{j-1}^\star$$

$$\alpha_j := \frac{(\mathbf{w}_j, \mathbf{r}_j^\star)_2}{(\mathbf{A}\mathbf{p}_j, \mathbf{p}_j^\star)_2}$$

PBiCG solver code (cont.)

PBICG.C

```
for (register label cell=0;
             cell<nCells; cell++)
{
 psiPtr[cell]+= alpha*pAPtr[cell];

 rAPtr[cell] -= alpha*wAPtr[cell];

 rTPtr[cell] -= alpha*wTPtr[cell];
}
```

$$\psi_j := \psi_{j-1} + \alpha_j \mathbf{p}_j$$

$$\mathbf{r}_{j+1} := \mathbf{r}_j - \alpha_j \mathbf{A}\mathbf{p}_j$$

$$\mathbf{r}^\star_{j+1} := \mathbf{r}^\star_j - \alpha_j \mathbf{A}^T \mathbf{p}^\star_j$$

BiCG method

- For the BiCG method, orthogonality is assured by the Petrov-Galerkin condition i.e.
- $\mathbf{r}_0$ residual vector with $\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
- $\mathbf{x}_m \in \mathbf{x}_0 + K_m$ and

$$L_m = K_m^T = span\{\mathbf{r}_0, \mathbf{A}^T\mathbf{r}_0, \ldots, (\mathbf{A}^T)^{m-1}\mathbf{r}_0\}$$

Disadvantages BICG method

- There are multiplications with A^T needed.
- For a regular matrix, the method could terminate without solution
- The method has no minimization properties for the iteration vector. This could cause an oscillatory behaviour in the convergence

Tutorial PBiCGstab

Motivation

- We have just seen that the method has some disadvantages.
- We consider the BiCGstab method from van der Vorst [8] which avoids these disadvantages.
  - In general, BiCGstab has smoother convergence properties
  - Multiplication with A^T is not necessary.
- Furthermore, we want to know how to expand a linear system solver in OpenFOAM.

Tutorial PBiCGstab

Goal

Goal

- Extension of the BiCG solver (`PBiCG.C`) to the stabilized version BiCGStab

Proceeding

- Change to the directory `$WM_PROJECT_USER_DIR$`
- Clone the folder `PBICG`

```
cp -r $FOAM_SRC/OpenFOAM/matrices/lduMatrice/solver/PBICG .
mv PBiCG PBiCGSTAB
cd PBiCGSTAB
```

- Change all names, pre- & post-fixes in `PBiCGSTAB.C/.H`: string `PBICGSTAB` $\Rightarrow$ `PBICGSTAB`:

```
rename 's/PBiCG/PBiCGSTAB/g' *
sed -i 's/PBiCG/PBiCGSTAB/g' *
```

Tutorial BiCGstab

Pseudo-code BiCGstab (from: Meister [9])

Choose $\mathbf{x}_0 \in \mathbb{R}^n$ und $\epsilon > 0$,

$\mathbf{r}_0 = \mathbf{p}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0,\ \mathbf{r}_0^P = \mathbf{p}_0^P := \mathbf{P}_L\mathbf{r}_0, \quad \rho_0^P := (\mathbf{r}_0^P, \mathbf{r}_0^P)_2,\ j := 0$

While $\|\mathbf{r}_j\|_2 > \epsilon$

$\mathbf{v}_j := \mathbf{A}\mathbf{p}_j^P,\ \mathbf{v}_j^P := \mathbf{P}_L\mathbf{v}_j$

$\alpha_j^P := \dfrac{\rho_j^P}{(\mathbf{v}_j^P,\ \mathbf{r}_0^P)_2},\ \mathbf{s}_j := \mathbf{r}_j - \alpha_j^P\mathbf{v}_j,\ \mathbf{s}_j^P := \mathbf{P}_L\mathbf{s}_j$

$\mathbf{t}_j := \mathbf{A}\mathbf{s}_j^P,\ \mathbf{t}_j^P := \mathbf{P}_L\mathbf{t}_j$

$\omega_j^P := \dfrac{(\mathbf{t}_j^P, \mathbf{s}_j^P)_2}{(\mathbf{t}_j^P, \mathbf{t}_j^P)_2}$

$\mathbf{x}_{j+1}^P := \mathbf{x}_j^P + \alpha_j^P\mathbf{p}_j^P + \omega_j^P\mathbf{s}_j^P$

$\mathbf{r}_{j+1} := \mathbf{s}_j - \omega_j^P t_j,\ \mathbf{r}_{j+1}^P := \mathbf{s}_j^P - \omega_j^P\mathbf{t}_j^P$

$\rho_{j+1}^P := (\mathbf{r}_{j+1}^P, \mathbf{r}_0^P)_2,\ \beta_j^P := \dfrac{\alpha_j^P}{\omega_j^P}\dfrac{\rho_{j+1}^P}{\rho_j^P}$

$\mathbf{p}_{j+1}^P := \mathbf{r}_{j+1}^P + \beta_j^P(\mathbf{p}_j^P - \omega_j^P\mathbf{v}_j^P),\ j := j+1$

$\mathbf{x}_j = \mathbf{P}_R\mathbf{x}_j^P$

Tutorial `BiCGstab`

Procedure

- Translate the pseudo-code of the previous slide into source code
- You only have to change the file `PBiCGSTAB.C`
- Use the code from `PBiCG.C` and his pseudo-code as pattern
- If you get in trouble look after the example code `BiCGSTAB.C`
- Try to track the algorithm in `BiCGSTAB.C`
- In OpenFOAM code:
  - $\mathbf{x}$ = `psi`
  - ρ = `alpha1`
  - $\mathbf{r}_0$ = `r0`
  - $\mathbf{r}^P$ = `rP`

  ...

Tutorial BiCGstab

Make

- Add the following entries to your Make files
  - files

```
./PBiCGSTAB.C

LIB = \$(FOAM_USER_LIBBIN)/PBiCGSTAB
```

  - options

```
EXE_INC = -I$(OBJECTS_DIR)

LIB_LIBS = \
    $(FOAM_LIBBIN)/libOSspecific.o \
    -L$(FOAM_LIBBIN)/dummy -lPstream \
    -lz
```

- Or add the entries to the Make file for our common user-defined library libmylib

Tutorial BiCGstab

PBiCGSTAB.C -- ::solve I

```
// * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * //

Foam::lduMatrix::solverPerformance Foam::PBiCGSTAB::solve
(
    scalarField& psi,
    const scalarField& source,
    const direction cmpt
) const
{
    // --- Setup class containing solver performance data
    lduMatrix::solverPerformance solverPerf
    (
        lduMatrix::preconditioner::getName(controlDict_) + typeName,
        fieldName_
    );

    register label nCells = psi.size();

    scalar* __restrict__ psiPtr = psi.begin();
    scalarField pA(nCells);
    scalar* __restrict__ pAPtr = pA.begin();
    scalarField wA(nCells);
    scalar* __restrict__ wAPtr = wA.begin();
    scalarField vA(nCells);
    scalar* __restrict__ vAPtr = vA.begin();
    scalarField sA(nCells);
    scalar* __restrict__ sAPtr = sA.begin();
    scalarField tA(nCells);
    scalar* __restrict__ tAPtr = tA.begin();
    // --- Calculate A.psi
```

Tutorial BiCGstab

PBiCGSTAB.C -- ::solve II

```
// --- Calculate initial residual
scalarField rA(source - wA);
scalar* __restrict__ rAPtr = rA.begin();
scalarField rP(nCells);
scalar* __restrict__ rPPtr = rP.begin();
scalarField r0(nCells);
scalar* __restrict__ r0Ptr = r0.begin();

// --- Define help-fields
scalarField vP(nCells);
scalar* __restrict__ vPPtr = vP.begin();

scalarField sP(nCells);
scalar* __restrict__ sPPtr = sP.begin();

scalarField tP(nCells);
scalar* __restrict__ tPPtr = tP.begin();

// --- Calculate normalisation factor
scalar normFactor = this->normFactor(psi, source, wA, pA);

if (lduMatrix::debug >= 2)
{
    Info<< "   Normalisation factor = " << normFactor << endl;
}

// --- Calculate normalised residual norm
solverPerf.initialResidual() = gSumMag(rA)/normFactor;
solverPerf.finalResidual() = solverPerf.initialResidual();
```

Tutorial BiCGstab

PBiCGSTAB.C -- ::solve III

```
// --- Check convergence, solve if not converged
if (!solverPerf.checkConvergence(tolerance_, relTol_))
{
    // --- Select and construct the preconditioner
    autoPtr<lduMatrix::preconditioner> preconPtr =
    lduMatrix::preconditioner::New
    (
        *this,
        controlDict_
    );

    // --- precondition r0
    preconPtr->precondition(rP, rA, cmpt);

    // --- initialize "pA and r0" with the preconditioned r0
    for (register label cell=0; cell<nCells; cell++)
    {
        pAPtr[cell] = rPPtr[cell];
        r0Ptr[cell] = rPPtr[cell];
    }

    // --- initialize rPr0
    scalar rPr0 = gSumProd(rP,r0);

    // --- Solver iteration
    do
    {
        // --- Calculate v=A*p and precondition v
        matrix_.Amul(vA, pA, interfaceBouCoeffs_, interfaces_, cmpt);
        preconPtr->precondition(vP, vA, cmpt);
```

Tutorial BiCGstab

PBiCGSTAB.C -- ::solve IV

```
// --- Calculate alpha=rPr0/vPr0
scalar alpha2 = gSumProd(vP, r0);
scalar alpha = rPr0/alpha2;


// --- Calculate s=rA-alpha*v
for (register label cell=0; cell<nCells; cell++)
{
    sAPtr[cell] = rAPtr[cell] - alpha*vAPtr[cell];
}

// --- Precondition s
preconPtr->precondition(sP, sA, cmpt);

// --- Calculate t=A*sP and precondition t
matrix_.Amul(tA, sP, interfaceBouCoeffs_, interfaces_, cmpt);
preconPtr->precondition(tP, tA, cmpt);

// --- Calculate alpha1=tPsP/tPtP
scalar tPsP=gSumProd(tP,sP);
scalar tPtP = gSumProd(tP,tP);
if (solverPerf.checkSingularity(mag(tPtP)/normFactor)) break;
scalar alpha1 = tPsP/tPtP;

// --- Update solution and residual
for (register label cell=0; cell<nCells; cell++)
{
    psiPtr[cell] += alpha*pAPtr[cell] + alpha1*sPPtr[cell];
    rAPtr[cell] = sAPtr[cell] - alpha1*tAPtr[cell];
   rPPtr[cell] = sPPtr[cell] - alpha1*tPPtr[cell];
```

Tutorial BiCGstab

PBiCGSTAB.C -- ::solve V

```
            solverPerf.finalResidual() = gSumMag(rA)/normFactor;

            // --- Update rPr0 and calculate beta
            scalar rPr0old = rPr0;
            rPr0 = gSumProd(rP, r0);
            scalar beta = (alpha*rPr0)/(alpha1*rPr0old);


            // --- Update search directions
            for (register label cell=0; cell<nCells; cell++)
            {
                pAPtr[cell] = rPPtr[cell] + beta*(pAPtr[cell] - alpha1*vPPtr[cell]);
            }

        } while
        (
            solverPerf.nIterations()++ < maxIter_
        && !(solverPerf.checkConvergence(tolerance_, relTol_))
        );
    }

    return solverPerf;
}


// ************************************************************************* //
```

Tutorial BiCGstab

Test case pitzDaily

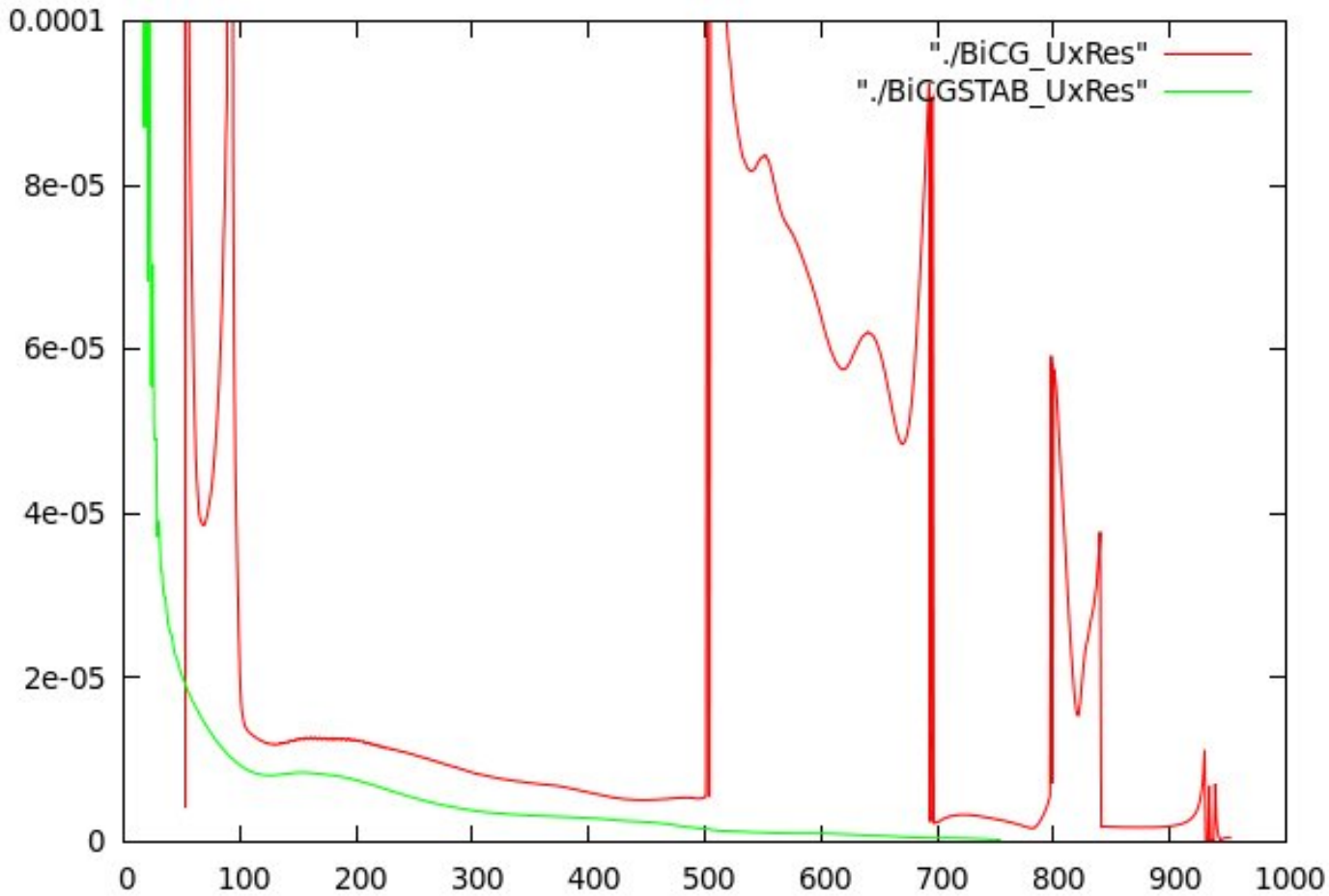
- Solver `simpleFoam`
- Settings

```
"U|k|epsilon|R"
{
    solver          PBiCGSTAB;
    preconditioner  DILU;
    tolerance       1e-05;
    relTol          0.1;
}
```

- **BiCG** analogue
- **Convergence after**
- **BiCG** 952 Iterations
- **BiCGSTAB** 754 Iterations

Tutorial BiCGstab

Comparison BiCG/BiCGSTAB – Residuals



Comparison of the residuals BiCG/BiCGSTAB

Literature I

[1] Hirt, C. W., Nichols, B. D. *Volume of Fluid (VOF) Method for the Dynamic sof Free Boundaries*, Journal of Computational Physics, 39: 201-225-65, 1981.

[2] Ubbink, O, *Numerical prediction of two fluid systems with sharp interfaces*, Ph.D Thesis, Imperial College of Science, Technology and Medicine, London, 1997.

[3] Rusche, H. *Computational Fluid Dynamics of Dispersed Two-Phase Flows at High Phase Fractions*, Ph.D Thesis, Imperial College of Science, Technology and Medicine, London, 2002.

[4] Damián, S. M. *Description and utilization of* `interFoam` *multiphase solver*, -Final Work-Computational Fluid Dynamics. `http://infofich.unl.edu.ar/upload/5e6dfd7ff282e2deabe7447979a16d49b0b1b675.pdf`

[5] Eslamdoost, A. *Forced Roll Motion of a 2D Box and Interaction with Free-Surface*, PhD course in CFD with OpenSource software, Chalmers University, 2009 `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2009/ArashEslamdoost/RollMotionofaBoxandInteractionwithFreeSurface.pdf`

[6] Issa, R. I. *Solution of the implicitly discretised fluid flow equations by operator splitting*, Journal of Computational Physics, 62: 40-65, 1985.

[7] Van Leer, B. *Towards the ultimate conservative difference scheme III. Upstream-centered finite-difference schemes for ideal compressible flow*, J. Comp. Phys. 23 (3): 263–275, 1977.

[8] van der Vorst, H. A. *BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 13: 631-644, 1992.

[9] Meister, A. *Numerik linearer Gleichungssysteme*, Vieweg Verlag, 2011.