12th OpenFOAM Workshop *training session* **Iterative linear solvers**

Tessa Uroić

Supervisor: Professor Hrvoje Jasak
Chair of Turbomachinery
Department of Energy, Power Engineering and Environment
Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb

tessa.uroic@fsb.hr

July 24, 2017

Overview

From mesh to matrix

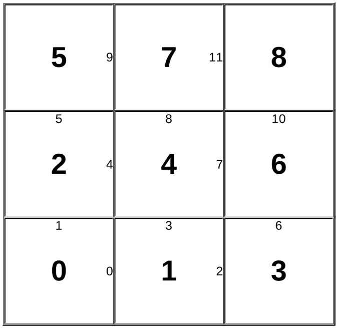


Figure 1: Numbers in bold are cell indices and the numbers in smaller font denote face indices. Notice that numbering of cells and faces in FOAM is executed in a way in which cells will have neighbours with the smallest possible index. This will result in organised face addressing. Note: use `renumberMesh`.

$$
\begin{bmatrix}
a_{00} & a_{01} & a_{02} & & & & & & \\
a_{10} & a_{11} & & a_{13} & a_{14} & & & & \\
a_{20} & & a_{22} & & a_{24} & a_{25} & & & \\
& a_{31} & & a_{33} & & & a_{36} & & \\
& a_{41} & a_{42} & & a_{44} & & a_{46} & a_{47} & \\
& & a_{52} & & & a_{55} & & a_{57} & \\
& & & a_{63} & a_{64} & & a_{66} & & a_{68} \\
& & & & a_{74} & a_{75} & & a_{77} & a_{78} \\
& & & & & & a_{86} & a_{87} & a_{88}
\end{bmatrix}
$$

From mesh to matrix

$$\begin{bmatrix}
a_{00} & a_{01} & a_{02} & & & & & & \\
a_{10} & a_{11} & & a_{13} & a_{14} & & & & \\
a_{20} & & a_{22} & & a_{24} & a_{25} & & & \\
& a_{31} & & a_{33} & & & a_{36} & & \\
& a_{41} & a_{42} & & a_{44} & & a_{46} & a_{47} & \\
& & a_{52} & & & a_{55} & & a_{57} & \\
& & & a_{63} & a_{64} & & a_{66} & & a_{68} \\
& & & & a_{74} & a_{75} & & a_{77} & a_{78} \\
& & & & & & a_{86} & a_{87} & a_{88}
\end{bmatrix}$$

- The matrix is stored in three arrays, one for the diagonal elements, one for the upper triangle and one for the lower triangle:
  - `diag()` - array of diagonal elements
  - `upper()` - array of upper-triangular elements
  - `lower()` - array of lower-triangular elements
- Each face has its owner (smaller cell index) and neighbour (larger cell index). The upper and lower arrays can be looped through by using face indices.
- The addressing is contained in two arrays (also using the face index):
  - `lowerAddr()` - containing the owner indices - row of the upper-triangular element
  - `upperAddr()` - containing the neighbour indices - column of the upper-triangular element
- The matrix cannot be read row by row sequentially using this addressing.

From mesh to matrix

The addressing and coefficients for the previous mesh are presented in the table:

Face index k	lowerAddr()	upperAddr()	upper()	lower()
0	0	1	a_{01}	a_{10}
1	0	2	a_{02}	a_{20}
2	1	3	a_{13}	a_{31}
3	1	4	a_{14}	a_{41}
4	2	4	a_{24}	a_{42}
5	2	5	a_{25}	a_{52}
6	3	6	a_{36}	a_{63}
7	4	6	a_{46}	a_{64}
8	4	7	a_{47}	a_{74}
9	5	7	a_{57}	a_{75}
10	6	8	a_{68}	a_{86}
11	7	8	a_{78}	a_{87}

- $owner =$ lowerAddr()[k]
- $neighbour =$ upperAddr()[k]
- $A_{ij} =$ upper()[k]
- $A_{ji} =$ lower()[k], and k is the face index.

Compressed row addressing

- Thinking in terms of matrix rather than mesh, it is necessary to access the coefficients row–by–row or column–by–column.
- For this purpose two storage formats are available: `ownerStartAddr()` and `losortStartAddr()`.
- `ownerStartAddr()`
  - (For cell index) returns the first face index k for which the cell is the owner.
  - This means you can move through the upper triangle row by row. In the lower triangle you move column by column.
- `losortStartAddr()`
  - (For cell index) returns the first face index k for which the cell is the neighbour (losort addressing is used).
  - Losort addressing is calculated by ordering the faces so that the upper addressing (neighbours) are listed in ascending order.
  - This enables looping through the lower triangle row–by–row and in the upper triangle column–by–column.

Compressed row addressing

Cell index	ownerStartAddr()
0	0
1	2
2	4
3	6
4	7
5	9
6	10
7	11
8	12

Index	Losort	upperAddr()
0	0	1
1	1	2
2	2	3
3	3	4
4	4	4
5	5	5
6	6	6
7	7	6
8	8	7
9	9	7
10	10	8
11	11	8

Cell index	losortStartAddr()
0	0
1	0
2	1
3	2
4	3
5	5
6	7
7	8
8	10

- The first table contains the pointer to the beginning of the row in upper triangle for each cell. It comes from the ordering of the lower addressing.
- The second table contains reoredered addressing (losort) which gives the lower triangle row–by-row.
- The third table gives the pointer to the beginning of the row in lower triangle.

How to use compressed row addressing?

Let's say we want to access the upper triangle and find row and column index and the coefficient value.

```
labelList ownerStart = matrix.lduAddr().ownerStartAddr();

for (label cell = 0; cell < numberOfCells; cell++)
{
    for
    (
        label face = ownerStart[cell];
        face < ownerStart[cell + 1];
        face++
    )
    {
        label row = cell;
        label column = matrix.lduAddr().upperAddr()[face];
        scalar aij = matrix.upper()[face];
    }
}
```

How to use compressed row addressing?

We want to find the coefficient in the lower triangle and also find the value of the diagonal coefficient in its column.

```
labelList losort = matrix.lduAddr().losortAddr();
labelList losortStart = matrix.lduAddr().losortStartAddr();

for (label cell = 0; cell < numberOfCells; cell++)
{
    for
    (
        label index = losortStart[cell];
        index < losortStart[cell + 1];
        index++
    )
    {
        label face = losort[index];
        label row = cell;
        label column = matrix.lduAddr().lowerAddr()[face];
        scalar aji = matrix.lower()[face];
        scalar diag = matrix.diag()[column];
    }
}
```

What are eigenvectors?

- **What is an eigenvector?** An "eigenvector" v of a matrix $\mathbf{B}$ is a nonzero vector that does not rotate when the matrix is applied (multiplied) to it (except if it points precisely the opposite direction) - the eigenvector might change length or reverse its direction, but it won't turn sideways.
- In other words, there is some scalar constant λ such that

$$\mathbf{B}v = \lambda v. \tag{1}$$

- The value λ is an "eigenvalue" of $\mathbf{B}$.
- **Why are eigenvectors significant?** Iterative methods often depend on applying $\mathbf{B}$ to a vector over and over again. When $\mathbf{B}$ is repeatedly applied to an eigenvector v, two things can happen:
  - If $|\lambda| < 1$, then $\mathbf{B}^i v = \lambda^i v$ will vanish as i approaches infinity.
  - If $|\lambda| > 1$, then $\mathbf{B}^i v$ will grow to infinity.

What are eigenvectors?

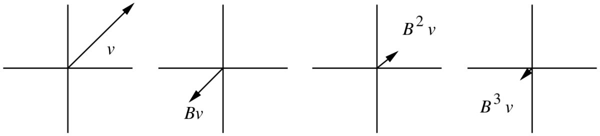


Figure 2: v is an eigenvector of $\mathbf{B}$ with a corresponding eigenvalue of -0.5. As i increases, $\mathbf{B}^i v$ converges to zero.

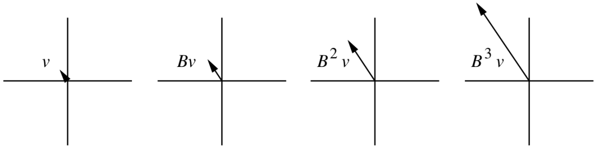


Figure 3: v is an eigenvector of $\mathbf{B}$ with a corresponding eigenvalue of 2. As i increases, $\mathbf{B}^i v$ diverges to infinity.

Each time $\mathbf{B}$ is applied, the vector grows or shrinks according to the value of λ.

FSB

What are eigenvectors?

- **But, what if $\mathbf{B}$ is applied to a vector that is not an eigenvector?** Think of a vector as a sum of other vectors whose behaviour is understood.
- Consider that the set of eigenvectors v_i forms a basis for $\mathbb{R}^n$ (because a symmetric matrix has n eigenvectors that are linearly independent). Any n–dimensional vector can be expressed as a linear combination of these eigenvectors.
- Matrix-vector multiplication is distributive, so we can examine the effect of $\mathbf{B}$ on each eigenvector separately.
- Applying $\mathbf{B}$ to a vector $x = v_1 + v_2$ is equivalent to applying $\mathbf{B}$ to the eigenvectors and summing the result.
- On repeated application: $\mathbf{B}^i x = \mathbf{B}^i v_1 + \mathbf{B}^i_v 2 = \lambda_1^i v_1 + \lambda_2^i v_2$
- If the magnitudes of all eigenvalues are smaller than one, $\mathbf{B}^i x$ will converge to zero, because the eigenvectors that compose x converge to zero when $\mathbf{B}$ is repeatedly applied. If one of the magnitudes is greater than one, x will diverge into infinity.

Jacobi method

$$\begin{bmatrix}
a_{00} & a_{01} & a_{02} & & & & & & \\
a_{10} & a_{11} & & a_{13} & a_{14} & & & & \\
a_{20} & & a_{22} & & a_{24} & a_{25} & & & \\
 & a_{31} & & a_{33} & & & a_{36} & & \\
 & a_{41} & a_{42} & & a_{44} & & a_{46} & a_{47} & \\
 & & a_{52} & & & a_{55} & & a_{57} & \\
 & & & a_{63} & a_{64} & & a_{66} & & a_{68} \\
 & & & & a_{74} & a_{75} & & a_{77} & a_{78} \\
 & & & & & & a_{86} & a_{87} & a_{88}
\end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \end{bmatrix}$$

- Jacobi relies on decomposing the matrix $\mathbf{A}$ into a sum of diagonal $\mathbf{D}$, and off-diagonal part $\mathbf{E}$.
  $\mathbf{A} = \mathbf{D} + \mathbf{E}$
- Then, write the system
  $\mathbf{A}x = b$
  as
  $(\mathbf{D} + \mathbf{E})x = b$
  and then
  $\mathbf{D}x = -\mathbf{E}x + b$.
- $\mathbf{D}$ is diagonal, thus, easy to invert:
  $x = -\mathbf{D}^{-1}\mathbf{E}x + \mathbf{D}^{-1}b$
- Replace $\mathbf{B} = -\mathbf{D}^{-1}\mathbf{E}$ and $z = \mathbf{D}^{-1}b$:
  $x_{i+1} = \mathbf{B}x_i + z$

Eigenvalues and error smoothing

- Given a starting vector x_0, this formula generates a sequence of vectors.
- Each successive vector should be closer to the solution x than the previous. x is called a "stationary point", because if $x_i = x$, then $x_{i+1} = x$.
- Express the current solution as a sum of the correct solution x and the error e_i. Insert into the system:

$$x_{i+1} = \mathbf{B}x_i + z$$

$$x_{i+1} = \mathbf{B}(x + e_i) + z$$

$$x_{i+1} = \mathbf{B}x + \mathbf{B}e_i + z$$

Eigenvalues and error smoothing

Stationary point is characterised by:

$$\mathbf{B}x + z = x$$

If we insert this expression for stationary point into 2:

$$x_{i+1} = x + \mathbf{B}e_i$$

- Thus, iterations do not affect the correct part of x_i, but they affect the error term.
- We saw before that, if $\rho(\mathbf{B}) < 1$, then the error term will converge to zero as i approaches infinity.
- The choice of the initial solution x_0 does not determine whether the method will converge, but it affects the number of iterations.
- Suppose that v_j is the eigenvector of $\mathbf{B}$ with the largest eigenvalue ($\rho(\mathbf{B}) = \lambda_j$). If the initial error e_0, expressed as a linear combination of eigenvectors, includes a component in the direction of v_j, this component will be the slowest to converge.

Eigenvalues and error smoothing

Relation between the residual and error:

$$r_i = b - \mathbf{A}x_i$$

$$r_i = b - \mathbf{A}x_i = \mathbf{A}x - \mathbf{A}x_i = A(x - x_i)$$

$$r_i = Ae_i$$

Residual is used as an indicator for the size of the error (which is not known). Solve the residual equation and use it to compute a correction to the solution x. Fixed–point methods tend to quickly reduce the *high frequency solution errors*, i.e. the errors whose direction corresponds to the largest eigenvalues of the matrix. However, the low frequency errors remain and this is why the convergence of these methods deteriorates.

$$Ae_i = r_i$$

If the error is an eigenvector $e_i = v_i$:

$$Av_i = r_i = \lambda v_i$$

Small residual, but large error!

Gauss–Seidel method

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & & & & & & \\ a_{10} & a_{11} & & a_{13} & a_{14} & & & & \\ a_{20} & & a_{22} & & a_{24} & a_{25} & & & \\ & a_{31} & & a_{33} & & & a_{36} & & \\ & a_{41} & a_{42} & & a_{44} & & a_{46} & a_{47} & \\ & & a_{52} & & & a_{55} & & a_{57} & \\ & & & a_{63} & a_{64} & & a_{66} & & a_{68} \\ & & & & a_{74} & a_{75} & & a_{77} & a_{78} \\ & & & & & & a_{86} & a_{87} & a_{88} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \end{bmatrix}$$

- Gauss-Seidel principle:

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

$$\mathbf{A} = \mathbf{U} + \mathbf{D} + \mathbf{L}$$

$$(\mathbf{U} + \mathbf{D} + \mathbf{L})x = b$$

$$\mathbf{U}x + (\mathbf{D} + \mathbf{L})x = b$$

$$(\mathbf{D} + \mathbf{L})x^{new} = b - \mathbf{U}x^{old}$$

$$x^{new} = \mathbf{D}^{-1}(b - \mathbf{L}x^{new} - \mathbf{U}x^{old})$$

Gauss–Seidel: example

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & & & & & & \\ a_{10} & a_{11} & & a_{13} & a_{14} & & & & \\ a_{20} & & a_{22} & & a_{24} & a_{25} & & & \\ & a_{31} & & a_{33} & & & a_{36} & & \\ & a_{41} & a_{42} & & a_{44} & & a_{46} & a_{47} & \\ & & a_{52} & & & a_{55} & & a_{57} & \\ & & & a_{63} & a_{64} & & a_{66} & & a_{68} \\ & & & & a_{74} & a_{75} & & a_{77} & a_{78} \\ & & & & & & a_{86} & a_{87} & a_{88} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \end{bmatrix}$$

Forward sweep:

$$x_0^{new} = \frac{1}{a_{00}}(b_0 - a_{01}x_1^{old} - a_{02}x_2^{old})$$

$$x_1^{new} = \frac{1}{a_{11}}(b_1 - a_{10}x_0^{new} - a_{13}x_3^{old} - a_{14}x_4^{old})$$

Backward sweep:

$$x_8^{new} = \frac{1}{a_{88}}(b_8 - a_{87}x_7^{old} - a_{86}x_6^{old})$$

$$x_7^{new} = \frac{1}{a_{77}}(b_7 - a_{78}x_8^{new} - a_{75}x_5^{old} - a_{74}x_4^{old})$$

Gauss–Seidel converges for strictly diagonally dominant or symmetric and positive–definite matrices.

What is a positive–definite matrix?

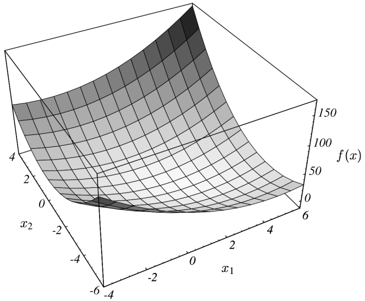
- A matrix is positive-definite if, for every nonzero vector x:
  $x^T \mathbf{A} x > 0$
- It is not a very intuitive idea, and it's hard to imagine how a matrix that is positive-definite might look differently from one that isn't. We will get a feeling for what positive-definiteness is about when we see how it affects the shape of quadratic forms.
- A quadratic form is simply a scalar, quadratic function of a vector with the form
  $f(x) = \frac{1}{2} x^T \mathbf{A} x - b^T x + c$
  where $\mathbf{A}$ is a matrix, x and b are vectors, and c is a scalar constant.
- Take this example:

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ -8 \end{bmatrix},$$

  where $c = 0$.
- The solution lies at the intersection of n hyperplanes, each having dimension $n-1$. For this problem, the solution is

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ -2 \end{bmatrix}$$

Quadratic form



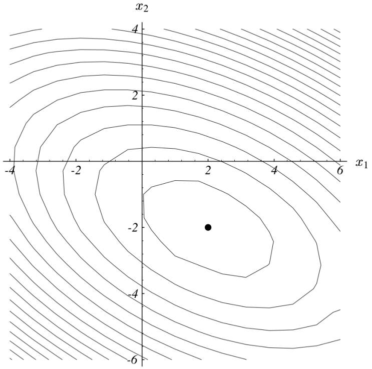


Figure 4: Left: Graph of a quadratic form. The minimum point is the solution to $\mathbf{A}x = b$.
Right: Contours of the quadratic form. Each ellipsoidal curve has constant $f(x)$.

Quadratic form

- The gradient of a quadratic form is a vector field that, for a given point x, points in the direction of greatest increase of $f(x)$:

$$f'(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \frac{\partial}{\partial x_2} f(x) \\ \vdots \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix}$$

- Now, calculate the gradient of the quadratic form: $f'(x) = \frac{1}{2}\mathbf{A}^T x + \frac{1}{2}\mathbf{A}x - b$
- If $\mathbf{A}$ is symmetric, this reduces to: $f'(x) = \mathbf{A}x - b$
- Setting the gradient to 0, we obtain the system of equations we wish to solve. Therefore, the solution to $\mathbf{A}x = b$ is a critical point of $f(x)$. If $\mathbf{A}$ is positive-definite and symmetric, solution of the system is a minimum of $f(x)$.

Quadratic form

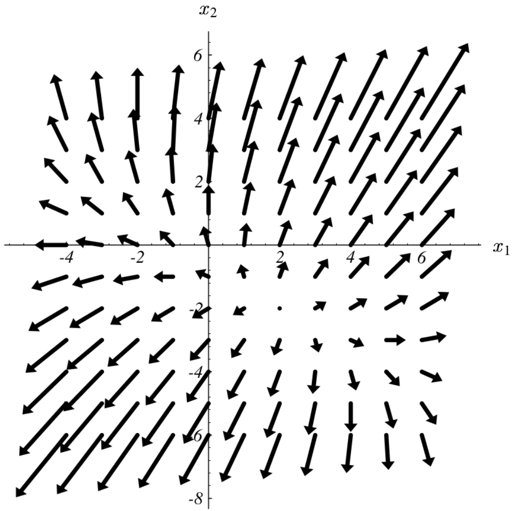


Figure 5: Gradient $f'(x)$ of the quadratic form. For every x, the gradient points in the direction of steepest increase of $f(x)$, and is orthogonal to the contour lines.

Quadratic form

- The fact that $f(x)$ is a paraboloid is our best intuition of what it means for a matrix to be positive-definite. If is not positive-definite, there are several other possibilities:
  - $\mathbf{A}$ could be negative-definite - the result of negating a positive-definite matrix (hold the paraboloid upside-down).
  - $\mathbf{A}$ might be singular, in which case no solution is unique; the set of solutions is a line or hyperplane having a uniform value for f.
  - If $\mathbf{A}$ is none of the above, then x is a saddle point, and techniques like Steepest Descent and CG will likely fail.
  - The values of b and c determine where the minimum point of the paraboloid lies, but do not affect the paraboloid's shape.

Quadratic form

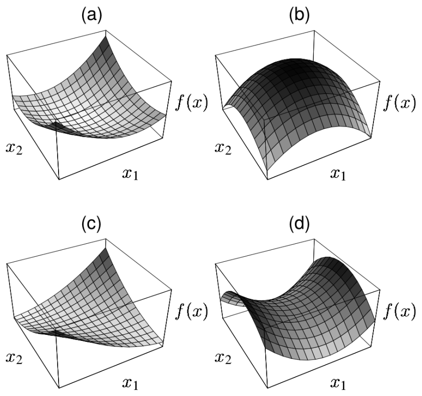


Figure 6:
(a) Quadratic form for a positive-definite matrix.
(b) For a negative-definite matrix.
(c) For a singular (and positive-indefinite) matrix. A line that runs through the bottom of the valley is the set of solutions.
(d) For an indefinite matrix. Because the solution is a saddle point, Steepest Descent and CG will not work. In three dimensions or higher, a singular matrix can also have a saddle.

Method of Steepest Descent

- Let's recap:
  - Error is a vector which indicates how far we are from the solution: $e_i = x_i - x$
  - Residual indicates how far we are from the correct value of the right hand side of the system: $r_i = b - \mathbf{A}x_i$
  - The residual is actually the error which is transformed by $\mathbf{A}$ into the space of b: $r_i = -\mathbf{A}e_i$
  - More importantly: $r_i = -f'(x_i)$
- In the method of the Steepest Descent, we start at an arbitrary point x_0 and take a series of steps $(x_1, x_2, ...)$ sliding down the paraboloid, until we are satisfied that we are close enough to the solution x. When we take a step, we choose the direction in which f decreases most quickly, which is the direction opposite to $f'(x_i)$. **So the residual is the *direction of the steepest descent.***
- But, how big a step should we take?

Line search

- A *line search* is a procedure that chooses α to minimize f along a line. We are restricted to choose a point on the intersection of the vertical plane and the paraboloid (Figure, b). Figure, c shows the parabola at the intersection. What is the value of α at the bottom of the parabola?
- From calculus, α minimizes f when $\frac{d}{d\alpha}f(x_1) = 0$
- By the chain rule: $\frac{d}{d\alpha}f(x_1) = f'(x_1)\frac{d}{d\alpha}x_1 = f'(x_1)r_0$
- Setting this expresion to zero (scalar product of two vectors) leads us to conclusion that α should be chosen so that r_0 and $f'(x_1)$ are orthogonal (Figure, d).

Line search

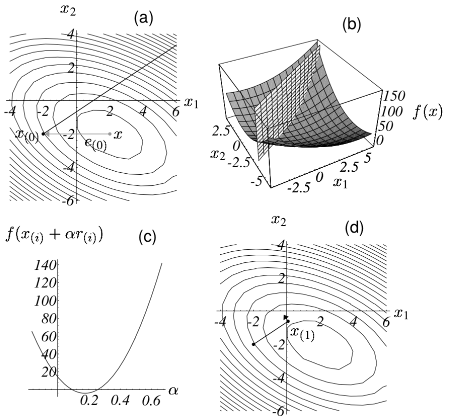


Figure 7: The method of Steepest Descent.

(a) Starting at $x_0 = \begin{bmatrix} -2 \\ -2 \end{bmatrix}$, take a step in the direction of steepest descent of f

(b) Find the point on the intersection of these two surfaces that minimizes f

(c) This parabola is the intersection of surfaces. The bottommost point is our target.

(d) The gradient at the bottommost point is orthogonal to the gradient of the previous step.

Line search

A visual explanation: Take a look at the Figure. It shows the gradient vectors at various points along the search line. The slope of the parabola at any point is equal to the magnitude of the projection of the gradient onto the line. These projections actually represent the rate of increase of f as one traverses the search line. f is minimised where projection is zero (gradient is orthogonal to the search line).

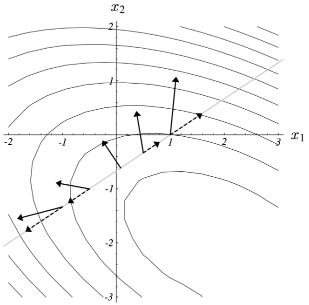


Figure 8: The gradient f' is shown at several locations along the search line (solid arrows). Each gradient's projection onto the line is also shown (dotted arrows). The gradient vectors represent the direction of steepest increase of f, and the projections represent the rate of increase as one traverses the search line. On the search line, f is minimised where the gradient is orthogonal to the search line.

Line search

To determine α:

$$f'(x_1) = -r_1$$

$$r_1 r_0 = 0$$

$$(b - \mathbf{A}x_1)r_0 = 0$$

$$(b - \mathbf{A}(x_0 + \alpha r_0))r_0 = 0$$

$$(b - \mathbf{A}x_0)r_0 - \alpha(\mathbf{A}r_0)r_0 = 0$$

$$(b - \mathbf{A}x_0)r_0 = \alpha(\mathbf{A}r_0)r_0$$

$$r_0 r_0 = \alpha r_0(\mathbf{A}r_0)$$

$$\alpha = \frac{r_0 r_0}{r_0 \mathbf{A} r_0}$$

Method of Steepest Descent

Finally, the method of Steepest Descent is:

$$r_i = b - \mathbf{A}x_i$$

$$\alpha_i = \frac{r_i^T r_i}{r_i^T \mathbf{A} r_i}$$

$$x_{i+1} = x_i + \alpha_i r_i$$

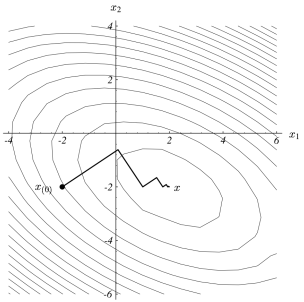


Figure 9: Here, the method of Steepest Descent starts at $x_0 = \begin{bmatrix} -2 \\ -2 \end{bmatrix}$ and converges at $x = \begin{bmatrix} 2 \\ -2 \end{bmatrix}$.
Note the zig–zag path, which appears because each gradient is orthogonal to the previous gradient.

Method of Conjugate Directions

- Steepest Descent often finds itself taking steps in the same direction as earlier steps. Wouldn't it be better if, every time we took a step, we got it right the first time?
- Let's pick a set of orthogonal search directions $d_0, d_1, ..., d_n$. In each search direction, we'll take exactly one step, and that step will be just the right length to line up evenly with x. After n steps, we'll be done.
- For each step choose a point:
  $x_{i+1} = x_i + \alpha_i d_i$
- From the fact that the search direction should be orthogonal to the error term, find α:
  $d_i^T e_{i+1} = 0$
  ...
  $\alpha_i = -\frac{d_i^T e_i}{d_i^T d_i}$
- But, we don't know the error term, so we haven't accomplished anything.

Method of Conjugate Directions

- Solution: make the search directions *A-orthogonal* or *conjugate*. Two vectors d_i and d_j are conjugate if:
  $d_i^T \mathbf{A} d_j = 0$
- The orthogonality condition corresponds to finding the minimum point along the search direction d_i.
- Fig. a shows what A orthogonal vectors look like. Imagine if you could stretch the paper to deform the picture until the ellipses appeared circular. The vectors would then appear orthogonal as in Fig. b.

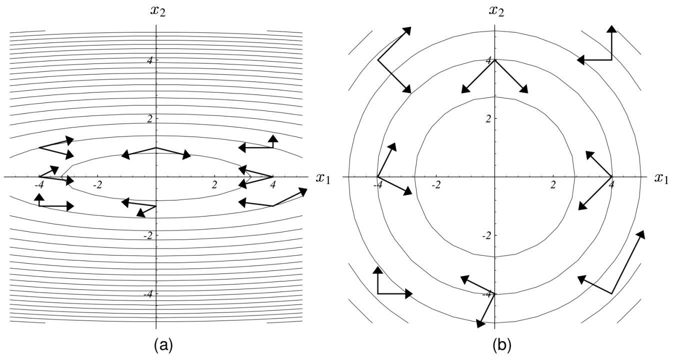


Figure 10: (a): A-orthogonal vectors, (b): orthogonal vectors

Method of Conjugate Directions

- Expression for α using conjugate directions:
  $$\alpha_i = -\frac{d_i^T \mathbf{A} e_i}{d_i^T \mathbf{A} d_i}$$
  $$\alpha_i = -\frac{d_i^T r_i}{d_i^T \mathbf{A} d_i}$$
- For searching A-orthogonal search directions there is a simple process called the *conjugate Gram–Schmidt process*
- When we take a step in a search direction, we don't need to step in that direction ever again: the error term is A-orthogonal to the old search directions. So is the residual because $r_i = -\mathbf{A} e_i$.

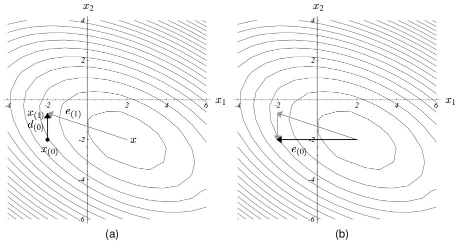


Figure 11: The method of Conjugate Directions converges in n steps. (a) The first step is taken along some direction d_0. The minimum point is chosen by the constraint that $e1$ must be A-orthogonal to d_0. (b) The initial error e_0 can be expressed as a sum of A-orthogonal components (gray arrows). Each step of Conjugate Directions eliminates one of these components.

Gram–Schmidt conjugation

- We have a set of n linearly independent vectors $u_0, u_1, ..., u_{n-1}$. To construct the search direction d_i, take u_i and subtract out any components that are not A-orthogonal to the previous d vectors.
- Set $d_0 = u_0$
  and for $i > 0$: $d_i = u_i + \sum_{k=0}^{i-1} \beta_{ik} d_{ik}$
- To find β_{ik} - because the search directions are A-orthogonal, it is possible to eliminate all the β_{ik} values but one by premultiplying the expression by $d^T\mathbf{A}$:

$$d_i^T\mathbf{A}d_j = u_i^T\mathbf{A}d_j + \sum_{k=0}^{i-1} \beta_{ik} d_k^T\mathbf{A}d_j$$

$$0 = u_i^T\mathbf{A}d_j + \beta_{ij} d_j^T\mathbf{A}d_j, i > j$$

$$\beta_{ij} = -\frac{u_i^T\mathbf{A}d_j}{d_j^T\mathbf{A}d_j}$$

- The difficulty of using the method is that all the old search vectors must be kept in memory to construct each new one.

Gram–Schmidt conjugation

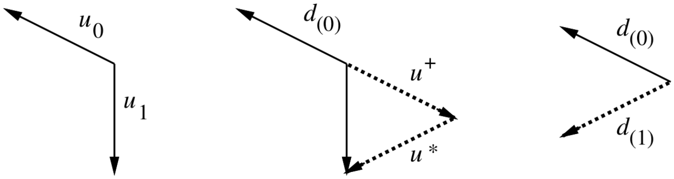


Figure 12: Gram-Schmidt conjugation of two vectors. Begin with two linearly independent vectors u_0 and u_1. Set $d_0 = u_0$. the vector u_1 is composed of two components: u^* which is A-orthogonal or conjugate to d_0 and u^+ which is parallel to d_0. After conjugation, only the A-orthogonal component remains and $d_1 = u^*$

Conjugate Gradient Method

- Conjugate Gradient Method is actually the method of Conjugate Directions where the search directions are constructed by conjugation of the residuals.
- Why the residual? The residual has the nice property that it's orthogonal to the previous search directions, so it's guaranteed always to produce a new, linearly independent search direction unless the residual is zero, in which case the problem is already solved.

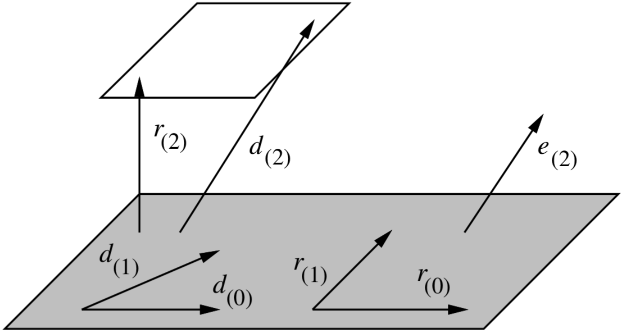


Figure 13: In the method of Conjugate Gradients, each new residual is orthogonal to all the previous residuals and search directions; and each new search direction is constructed (from the residual) to be A-orthogonal to all the previous residuals and search directions. The endpoints of r_2 and d_2 lie on a plane parallel to $\mathbb{D}_2$ (the shaded subspace). d_2 is a linear combination of r_2 and d_1.

Conjugate Gradient Method

- Because the search vectors are built from the residuals, the subspace span $r_0, r_1, ..., r_{i-1}$ is equal to $\mathbb{D}_i$.
- As each residual is orthogonal to the previous search directions, it is also orthogonal to the previous residuals: $r_i^T r_j = 0$.
- It can be shown that each new residual is a linear combination of the previous residual and $\mathbf{A}d_{i-1}$:

$$r_{i+1} = -\mathbf{A}ei + 1$$

$$r_{i+1} = -\mathbf{A}(ei + \alpha_i d_i)$$

$$r_{i+1} = r_i - \alpha_i \mathbf{A} d_i$$

- Since $d_i \in \mathbb{D}$, each new subspace $\mathbb{D}_{i+1}$ is formed from the union of the previous subspace $\mathbb{D}_i$ and the subspace $\mathbf{A}\mathbb{D}_i$.
- This subspace is called a *Krylov subspace* and it is a subspace created by repeatedly applying a matrix to a vector.
- Because $\mathbf{A}\mathbb{D}_i$ is included in $\mathbb{D}_{i+1}$, the next residual r_{i+1} is orthogonal to $\mathbb{D}_{i+1}$ means that r_{i+1} is A-orthogonal to $\mathbb{D}_i$. Gram-Schmidt conjugation becomes easy because r_{i+1} is already A-orthogonal to all of the previous search directions except d_i.

Preconditioning

- In practice, for large scale problems, CG methods are always used with a preconditioner.
- Suppose $\mathbf{M}$ is a symmetric, positive–definite matrix which is easier to invert than $\mathbf{A}$. Solve $\mathbf{A}x = b$ indirectly by solving:

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

- If the spectral properties of the matrix $\mathbf{M}^{-1}\mathbf{A}$ are better, we can solve the system more quickly than the original.
- But, $\mathbf{M}^{-1}\mathbf{A}$ does not have to be symmetric or positive–definite.
- For every positive–definite $\mathbf{M}$, there is a matrix $\mathbf{E}$ that has the property $\mathbf{E}\mathbf{E}^T = \mathbf{M}$.
- The matrices $\mathbf{M}^{-1}\mathbf{A}$ and $\mathbf{E}^{-1}\mathbf{A}\mathbf{E}^{-T}$ have the same eigenvalues.
- Transform the system $\mathbf{A}x = b$ to

$$\mathbf{E}^{-1}\mathbf{A}\mathbf{E}^{-T}x^* = \mathbf{E}^{-1}b$$

$$x^* = \mathbf{E}^T x$$

Preconditioning

- The system
$$\mathbf{E}^{-1}\mathbf{A}\mathbf{E}^{-T}x^* = \mathbf{E}^{-1}b$$
$$x^* = \mathbf{E}^T x$$
is solved first for x^* and then for x.
- The process is called the *Transformed Preconditioned Conjugate Gradient Method* and requires an explicit calculation of $\mathbf{E}$, but this can be avoided by a few manipulations. The final *Untransformed Preconditioned Conjugate Gradient Method*:
$$r_0 = b - Ax_0$$
$$d_0 = \mathbf{M}^{-1}r_0$$
$$\alpha_i = \frac{r_i^T\mathbf{M}^{-1}r_i}{d_i^T\mathbf{A}d_i}$$
$$x_{i+1} = x_i + \alpha_i d_i$$
$$r_{i+1} = r_i - \alpha_i\mathbf{A}d_i$$
$$\beta_i = \frac{r_{i+1}^T\mathbf{M}^{-1}r_{i+1}}{r_i^T\mathbf{M}^{-1}r_i}$$
$$d_{i+1} = \mathbf{M}^{-1}r_{i+1} + \beta_{i+1}d_i$$

Preconditioning

- Preconditioning can be considered as an attempt to stretch the quadratic form to make it appear more spherical (eigenvalues are more clustered, scaling along eigenvector axes). A perfect preconditioner is $\mathbf{M} = \mathbf{A}$ because $\mathbf{M}^{-1}\mathbf{A}$ is a unitary matrix and the system can be solved in a single iteration. But, computing $\mathbf{A}^{-1}$ is actually solving the system.
- The simplest preconditioner is the *Jacobi* or *diagonal* preconditioner (which scales the quadratic form along the coordinate axes). It's fast because diagonal matrix is easy to invert.
- A better preconditioner is *Cholesky preconditioning* which is a technique for factoring a matrix into the form $\mathbf{L}\mathbf{L}^T$, where $\mathbf{L}$ is a lower triangular matrix. It is a version of the *LU factorization* for symmetric matrices.
- The LU factorization is performed by a modified Gaussian elimination: eliminate the entries below the main diagonal to obtain the upper triangular matrix $\mathbf{U}$ and remember the multiple of row j that is subtracted from row i to zero the coefficient a_{ij}. This multiple is the subdiagonal entry of $\mathbf{L}$.

$$\forall_{i,j>k} a_{ij} = a_{ij} - a_{ik} a_{kk}^{-1} a_{kj} \tag{2}$$

Other variants of Krylov subspace solvers

Generalized Minimal Residual - GMRES

- Applicable to unsymmetric systems.
- Generates a sequence of orthogonal vectors, but in absence of symmetry all previously computed vectors in the orthogonal sequence have to be retained.
- Minimizes the residual norm.
- Restarted versions of the method are used to save memory.

Biconjugate Gradient - BiCG

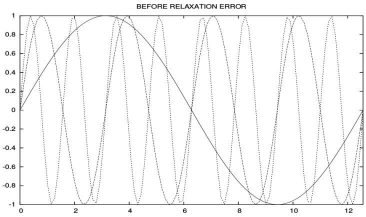
- Replaces the orthogonal sequence of residuals by two mutually orthogonal sequences, but at a price of no longer providing a minimization.
- Update relations for residuals in CG are augmented in BiCG by relations that are similar but based on $\mathbf{A}^T$ instead of $\mathbf{A}$.

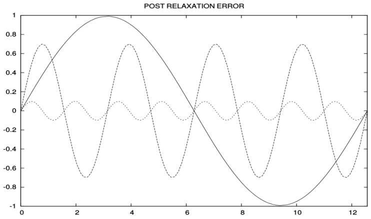
Biconjugate Gradient Stabilized - BiCGStab

- Can be interpreted as a product of BiCG and repeatedly applied GMRES. At least locally, a residual vector is minimized which leads to smoother convergence.

Multigrid

- Fixed–point iterative solvers (Jacobi, Gauss–Seidel) can quickly reduce (local) high–frequency errors and when the (global) low–frequency errors remain, the convergence stalls.
- It was noticed that the (global) low–frequency errors are removed at a rate inversely proportional to the size of the computational mesh.
- Multigrid algorithms help reduce the low–frequency error by successively solving the system on coarser meshes.
- The low–frequency errors on a fine mesh become high–frequency errors on a coarse mesh and can be efficiently eliminated.





The basic multigrid algorithm

1. On the **fine** level, use one of the fixed–point methods (Jacobi, Gauss–Seidel, ...) to calculate the approximate solution of the system: $\mathbf{A}x = b$.
2. Calculate the residual: $r_F = b_F - \mathbf{A}x_F$.
3. On the **coarse** level, use a fixed–point method to solve: $\mathbf{A}_C e_C = r_C$.
4. Use the error calculated on the coarse level to correct the solution on the fine level: $x_{new} = x_F + e_F$

How to define operators for transferring the residual and correction from fine level to coarse level, and vice versa? There are multiple possibilities, some of which will be explained in the following section.
Please note, algebraic multigrid operates only on matrix level and does not need information from the computational mesh (geometric multigrid does).

The basic multigrid algorithm

- Operator for transferring the residual from fine level to coarse level is called the *restriction operator*.
- The restriction operator can be a simple injection:
  $\mathbb{I}_h^{2h}\mathbf{v}^h = \mathbf{v}^{2h}$
  $\mathbf{v}_j^{2h} = \mathbf{v}_{2j}^h$
  which means that the coarse level vector will directly copy the components of the fine level vector.
- It can also be a weighted sum of vector components which correspond to neighbouring points on the fine level:
  $\mathbb{I}_h^{2h}\mathbf{v}^h = \mathbf{v}^{2h}$
  $$\mathbf{v}_j^{2h} = \tfrac{1}{4}(\mathbf{v}_{2j-1}^h + 2\mathbf{v}_{2j}^h + \mathbf{v}_{2j+1}^h), j = 1, ..., \frac{n-1}{2}$$
- Operator for transferring the solution correction from coarse to fine level is called the *prolongation operator*.
- It can also be an injection or a linear interpolation (which is a transpose of the restriction operator - variational principle).

The basic multigrid algorithm

- Options for forming interpolation by injection or weighted residual method:
- **Aggregative multigrid - AAMG**: equations are grouped into clusters (multiple fine cells are joined to form a single coarse cell). The groups are formed based on the size of off-diagonal coefficients: $\frac{a_{ij}^2}{a_{ii}a_{jj}}$ should exceed a certain size. The interpolation operator is piecewise constant (injection). The coarse level matrix coefficients are obtained as a sum of fine level matrix coefficients.
- **Selective multigrid - SAMG**: The set of coarse equations is a subset of fine level equations. The choice of the coarse equations is based on the number of equations which depend on a single equation. The interpolation operator is obtained from approximation of the algebraically smooth error. Coarse level matrix is obtained by a triple matrix product of restriction, fine level matrix and prolongation operator (Galerkin weighted residuals method).

Multigrid cycles

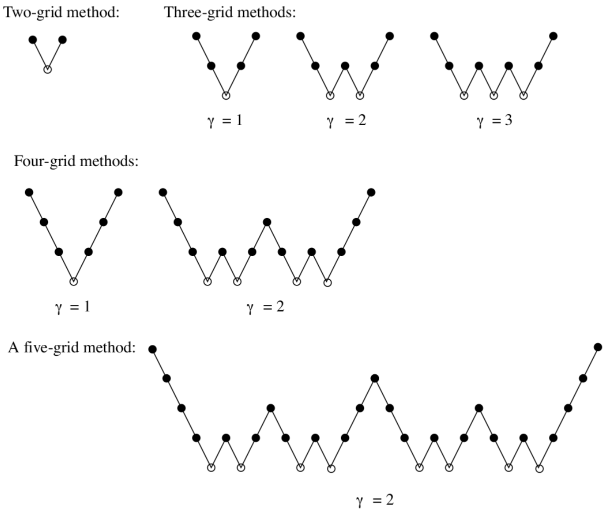


Figure 14: Structure of one multigrid cycle for different number of levels (grids), from *Trottenberg et al.: Multigrid (2001)*

Multigrid smoothers

- Classical fixed–point methods: **Gauss–Seidel**, symmetrical Gauss–Seidel...
- More effective (but also more computationally expensive) are matrix factorizations such as incomplete LU factorization (**ILU**).
- Smoothing is done before the restriction of the residual (*pre-smoothing*) and after the correction of the solution (*post-smoothing*). Post-smoothing should be more efficient.
- The fine level solver can be a Krylov subspace method and for coarse levels, a direct solver can be used.

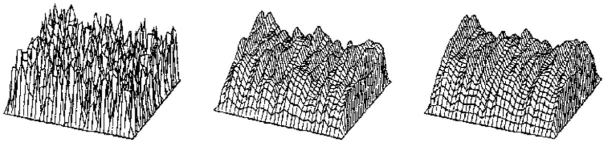


Figure 15: Effect of the Gauss–Seidel smoother

Summary

- Matrices arising from Finite Volume Method discretisation are square and sparse. The structure and number of non–zero coefficients is a consequence of the connectivity from the computational mesh.
- Classical iterative methods, e.g. Gauss–Seidel are not very effective in solving the system as they require matrices with special properties and converge very slowly. However, they are good smoothers.
- Krylov subspace methods are very robust and effective solvers for symmetric and unsymmetric matrices but require preconditioning.
- Multigrid solvers are extremely efficient and rely on the ability of the fixed–point solvers to smooth the error. Multigrid can be used as a stand-alone solver or as a preconditioner with the Krylov subspace solvers.

Used and recommended literature:

- Y. Saad: Iterative methods for sparse linear systems, 2000
- J.R. Shewchuk: An introduction to the Conjugate Gradient Method without the agonizing pain, 1994
- U. Trottenberg, C. Oosterlee, A. Schüller: Multigrid, 2001