# Linear System and Linear Solver

**Hrvoje Jasak**

**hrvoje.jasak@fsb.hr**

**FSB, University of Zagreb, Croatia**

# Matrix Assembly

From Discretisation to Linear System

- Assembling the terms from the discretisation method
  - Time derivative: $x$ depends on old value
  - Convection: $\mathbf{u}$ provided; $x_f$ depends on $x_P$ and $x_N$
  - Diffusion: $\mathbf{s}_f \bullet (\nabla x)_f$ depends on $x_P$ and $x_N$
- Thus, the value of the solution in a point depends on the values around it: this is always the case. For each computational point, we will create an equation

$$a_P x_P + \sum_N a_N x_N = R$$

where N denotes the neighbourhood of a computational point

  - Every time $x_P$ depends on itself, add contribution into $a_P$
  - Every time $x_N$ depends on itself, add contribution into $a_N$
  - Other contributions into $R$

# Implicit and Explicit Methods

Solution Advancement Method

- **Explicit method**: $x_P^n$ depends on the **old** neighbour values $x_N^o$
  - Visit each cell, and using available $x^o$ calculate

$$x_P^n = \frac{R - \sum_N a_N x_N^o}{a_P}$$

  - No additional information needed
  - Fast and efficient; however, poses the **Courant number limitation**: the information about boundary conditions is propagated very slowly and poses a limitation on the time-step size
- **Implicit method**: $x_P^n$ depends on the **new** neighbour values $x_N^n$

$$x_P^n = \frac{R - \sum_N a_N x_N^n}{a_P}$$

  - Each cell value of $x$ for the "new" level depends on others: all equations need to be solved simultaneously

# Linear System of Equations

Nomenclature

- Equations form a **linear system** or a matrix

$$[A][x] = [b]$$

  where $[A]$ contain matrix coefficients, $[x]$ is the value of $x_P$ in all cells and $[b]$ is the right-hand-side

- $[A]$ is potentially very big: N cells $\times$ N cells
- This is a **square matrix**: the number of equations equals the number of unknowns
- . . . but very few coefficients are non-zero. The matrix connectivity is always local, potentially leading to storage savings if a good format can be found
- What about non-linearity? Typically, we wish to avoid handling non-linearity at this level due to high cost of non-linear matrix solvers

# Matrix Storage Formats

Matrix Format

- **Dense matrix format**. All matrix coefficients have are stored, typically in a two-dimensional array
  - Diagonal coefficients: $a_{ii}$, off-diagonal coefficients: $a_{ij}$
  - Convenient for small matrices and direct solver use
  - Matrix coefficients represent a large chunk of memory: efficient operations imply memory management optimisation
  - It is impossible to say if the matrix is symmetric or not without floating point comparisons
- **Sparse matrix format**. Only non-zero coefficients will be stored
  - Considerable savings in memory
  - Need a mechanism to indicate the position of non-zero coefficients
  - This is **static format**, which imposes limitations on the operations: if a coefficient is originally zero, it is very expensive to set its value: recalculating the format. This is usually termed a **zero fill-in** condition
  - Searching for coefficients is out of the question: need to formulate sparse matrix algorithms

# Matrix Storage Formats

Sparse Matrix: Compressed Row Format

- Operate on a row-by-row basis. Diagonal coefficients may be separate
- Coefficients stored in a single 1-D array. Coefficients ordered row-by-row
- Addressing in two arrays: “row start” and “column”
- The column array records the column index for each coefficients. Size of column array equal to the number of off-diagonal coefficients
- The row array records the start and end of each row in the column array. Thus, row $i$ has got coefficients from `row[i]` to `row[i + 1]`. Size of row arrays equal to number of rows + 1

```
vectorProduct(b, x) //    [b] = [A] [x]
{
    for (int n = 0; n < count; n++)
    {
        for (int ip = row[n]; ip < row[n+1]; ip++)
        {
            b[n] = coeffs[ip]*x[col[ip]];
        }
    }
}
```

# Matrix Storage Formats

Sparse Matrix: Arrow Format

- Arbitrary sparse format. Diagonal coefficients typically stored separately
- Coefficients in 2-3 arrays: diagonal, upper and lower triangle
- Diagonal addressing implied
- Off-diagonal addressing in 2 arrays: “owner” (row index) “neighbour” (column index) array. Size of addressing equal to the number of coefficients
- The matrix structure (fill-in) is assumed to be symmetric: presence of $a_{ij}$ implies the presence of $a_{ji}$. Symmetric matrix easily recognised
- If the matrix coefficients are symmetric, only the upper triangle is stored – a symmetric matrix is easily recognised and stored only half of coefficients

```
vectorProduct(b, x) //   [b] = [A] [x]
{
    for (int n = 0; n < coeffs.size(); n++)
    {
        int c0 = owner(n);
        int c1 = neighbour(n);
        b[c0] = upperCoeffs[n]*x[c1];
        b[c1] = lowerCoeffs[n]*x[c0];
    }
}
```

# Matrix Format and Discretisation

FVM and Matrix Structure

- Relationship between the FV mesh and a matrix:
  - A cell value depends on other cell values only if the two cells share a face. Therefore, a correspondence exists between the off-diagonal matrix coefficients and the mesh structure
  - In practice, the matrix is assembled by looping through the mesh
- Finite Element matrix assembly
  - Connectivity depends on the shape function and point-to-cell connectivity in the mesh
  - In assembly, a local matrix is assembled and then inserted into the global matrix
  - Clever FEM implementations talk about the kinds of assembly without the need for searching: a critical part of the algorithm

# Matrix Properties

Matrix Characterisation

- We shall introduce a set of terms to describe a matrix in general terms
- A matrix is **sparse** if it contains only a few non-zero elements
- A sparse matrix is **banded** if its non-zero coefficients are grouped in a stripe around the diagonal
- A sparse matrix has a **multi-diagonal structure**if its non-zero off-diagonal coefficients form a regular diagonal pattern
- A **symmetric** matrix is equal to its transpose

$$[A] = [A]^T$$

- A matrix is **positive definite** if for every $[x] \neq [0]$

$$[x]^T [A][x] > 0$$

# Matrix Properties

Matrix Characterisation

- A matrix is **diagonally dominant** if in each row the sum of off-diagonal coefficient magnitudes is equal or smaller than the diagonal coefficient

$$a_{ii} \geq \sum_{j=1}^{N} |a_{ij}| \;\; ; \;\; j \neq i$$

and for at least one $i$

$$a_{ii} > \sum_{j=1}^{N} |a_{ij}| \;\; ; \;\; j \neq i$$

# Matrix Properties

Definition of a Residual

- Matrix form of the system we are trying to solve is

$$[A][x] = [b]$$

- The exact solution can be obtained by inverting the matrix $[A]$:

$$[x] = [A]^{-1}\,[b]$$

  This is how direct solvers operate: number of operations required for the inversion of $[A]$ is fixed and until the inverse is constructed we cannot get $[x]$
- Iterative solvers start from an approximate solution $[x]_0$ and generates a set of solution estimates $[x]_k$, where $k$ is the iteration counter
- Quality of the solution estimate is measured through a **residual**:

$$[r] = [b] - [A][x]_k$$

  Residual is a vector showing how far is the current estimate $[x]_k$ from the exact solution $[x]$. Note that for $[x]$, $[r]$ will be zero

# Matrix Properties

Definition of a Residual

- $[r]$ defines a value for every equation (row) in $[A]$: we need a better way to measure it. A residual norm $||r||$ can be assembled in many ways, but usually

$$||r|| = \sum_{j=1}^{N} |r_j|$$

  In CFD software, the residual norm is normalised further for easier comparison between the equations etc.

- Convergence of the iterative solver is usually measured in terms of residual reduction. When

$$\frac{||r_k||}{||r_0||} < \epsilon$$

  the linear system of equations is considered to be solved

# Linear Solvers

The Role of a Linear Solver

- Good (implicit) numerical simulation software will spend $50 - 90\%$ percent of CPU time inverting matrices: performance of linear solvers is absolutely critical for the performance of the solver
- Like in the case of mesh generation, we will couple the characteristics of a discretisation method and the solution algorithm with the linear solver
- Only a combination of a discretisation method and a linear solver will result in a useful solver. Typically, properties of discretisation will be set up in a way that allows the choice of an efficient solver

Solution Approach

- **Direct solver**. The solver algorithm will perform a given number of operations, after which a solution will be obtained
- **Iterative solver**. The algorithm will start from an initial solution and perform a number of operations which will result in an improved solution. Iterative solvers may be variants of the direct solution algorithm with special characteristics
- **Explicit method**. New solution depends on currently available values of the variables. The matrix itself is not required or assembled; in reality, the algorithm reduces to point-Jacobi or Gauss-Seidel sweeps

# Choice of Linear Solver

Direct or Iterative Solver

- Direct solvers: expensive in storage and CPU time but can handle any sort of matrix – no need to worry about matrix properties during discretisation
- Iterative solvers: work by starting from an initial guess and improving the solution. However, require matrices with “special” properties
- For large problems, iterative solvers are the only option
- Fortunately, the FVM matrices are ideally suited (read: carefully constructed) for use with iterative solvers
- Direct solver is typically used for cases where it is difficult to control matrix properties through discretisation: high-order FEM methods, Hermitian elements, Discontinuous Galerkin etc.

# Iterative Solution Algorithm

Partial Convergence

- When we are working on linear problems with linear discretisation in steady-state, the solution algorithm will only use a single solver call. This is very quick and very rare: linear systems are easy to simulate
- Example: linear stress analysis. In some FEM implementations, for matrices under a certain size the direct solver will be used exclusively for matrices under a given size
- In cases of coupled or non-linear partial differential equations, the solution algorithm will iterate over the non-linearity. Therefore, intermediate solution will only be used to update the non-linear parameters.
- With this in mind, we can choose to use partial convergence, update the non-linearity and solve again: capability of obtaining an intermediate solution at a fraction of the cost becomes beneficial
- Moreover, in iterative procedures or time-marching simulations, it is quite easy to provide a good initial guess for the new solution: solution form the previous iteration or time-step. This further improves the efficiency of the algorithm
- Historically, in partial convergence cases, FEM solvers use tighter tolerances that FVM: 6 orders of magnitude for FEM *vs.* 1-2 orders of magnitude for the FVM

# Direct Solver

Properties of Direct Solvers

- The most important property from the numerical point of view is that the number of operations required for the solution is known and intermediate solutions are of no interest
- **Matrix fill-in**. When operating on a large sparse matrix like the one from discretisation methods, the direct solver will create entries for coefficients that were not previously present. As a consequence, formal matrix storage requirement for a direct solver is a full matrix for a complete system: huge! This is something that needs to be handled in a special way
- Advantage of direct solvers is that they can handle any sort of well-posed linear system
- In reality, we additionally have to worry about pollution by the round-off error. This is partially taken into account through the details of the solution algorithm, but for really bad matrices this cannot be helped

# Direct Solver

Gaussian Elimination

- Gaussian elimination is the easiest direct solver: standard mathematics. Elimination is performed by combining row coefficients until a matrix becomes triangular. The elimination step is followed by backwards substitution to obtain the solution.
- **Pivoting**: in order to control the discretisation error, equations are chosen for elimination based on the central coefficient
- Combination of matrix rows leads to fill in
- Gaussian elimination is one of the cases of I-L-U decomposition solvers and is rarely used in practices
- The number of operations in direct solvers scales with the number of equations cubed: very expensive!

# Direct Solver

Multi-Frontal Solver

- When handling very sparse systems, the fill-in is very problematic: leads to a large increase in storage size and accounts for the bulk of operations
- **Window approach**: modern implementation of direct solvers
  - Looking at the structure of the sparse system, it can be established that equation for $x_P$ depends only on a small subset of other nodes: in principle, it should be possible to eliminate the equation for P just by looking at a small subset of the complete matrix
  - If all equations under elimination have overlapping regions of zero off-diagonal coefficients, there will be no fill-in in the shared regions of zeros!
  - Idea: Instead of operating on the complete matrix, create an active window for elimination. The window will sweep over the matrix, adding equations one by one and performing elimination immediately
  - The window matrix will be dense, but much smaller than the complete matrix. The triangular matrix (needed for back-substitution) can be stored in a sparse format
- The window approach may reduce the cost of direct solvers by several orders of magnitude: acceptable for medium-sized systems. The number of operations scales roughly with $N\,M^2$, where N is the number of equations and M is the maximum size of the solution window

# Direct Solver

Implementing Direct Solvers

- The first step in the implementation is control of the window size: the window changes its width dynamically and in the worst case may be the size of the complete matrix
- Maximum size of the window depends on the matrix connectivity and ordering of equation. Special optimisation software is used to control the window size: matrix renumbering and ordering heuristics
- Example: ordering of a Cartesian matrix for minimisation of the band
- Most expensive operation in the multi-frontal solver is the calculation of the Schur's complement: the difference between the trivial and optimised operation can be a factor fo 10000! In practice, you will not attempt this (cache hit rate and processor-specific pre-fetch operations)
- **Basic Linear Algebra (BLAs)** library: special assembly code implementation for matrix manipulation. Code is optimised by hand and sometimes written specially for processor architecture. It is unlikely that a hand-written code for the same operation achieves more than 10 % efficiency of BLAs. A good implementation can now be measured in how much the code spends on operations outside of BLAs.

# Simple Iterative Solvers

Simple Iterative Solvers: Fixed-Point Methods

- Performance of iterative solvers depends on the **matrix characteristics**. The solver operates by incrementally improving the solution, which leads to the concept of error propagation: if the error is augmented in the iterative process, the solver diverges
- The easiest way of analysing the error is in terms of **eigen-spectrum** of the matrix
- The general idea of iterative solvers is to replace $[A]$ with a matrix that is easy to invert and approximates $[A]$ and use this to obtain the new solution
  - **Point-Jacobi** solution
  - **Gauss-Seidel** solver
  - Tri-diagonal system and generalisation to 5- or 7-diagonal matrices
- **Propagation of information** in simple iterative solvers. Point Jacobi propagates the “data” one equation at a time: very slow. For Gauss-Seidel, the information propagation depends on the matrix ordering and sweep direction. In practice **forward** and **reverse** sweeps are alternated

# Fixed-Point Methods

Mathematical Formalism for Fixed-Point Methods

- Consider again linear problem $[A][x] = [b]$
- A **stationary iterative method** is obtained by splitting $[A] = [M] - [N]$:

$$[x]^{(\nu+1)} = [R][x]^{(\nu)} + [M]^{-1}[b]$$

- Here $[R]$ is the iteration matrix

$$[R] = [M]^{-1}[N]$$

- Define solution error $[e]$ and error propagation equation:

$$[e]^{(\nu)} = [x]^{(\nu)} - [x]^{*}$$
$$[e]^{(\nu+1)} = [R][e]^{(\nu)}$$

- Iteration matrix possesses the recursive property:

$$[e]^{(\nu+1)} = [R]^{\nu}[e]^{(0)} \tag{1}$$

# Krylov Space Solvers

Krylov Subspace Methods

- Looking at the direct solver, we can imagine that it operates in N-dimensional space, where N is the number of equations and searches for a point which minimises the residual
- In Gaussian elimination, we will be visiting each direction of the N-dimensional space and eliminating it from further consideration
- The idea of Krylov space solvers is that an approximate solution can be found more efficiently if we look for search directions more intelligently. A residual vector $[r]$ at each point contains the "direction" we should search in; additionally, we would like to always search in a direction orthogonal to all previous search directions
- On their own, Krylov space solvers are poor; however, when **matrix preconditioning** is used, we can assemble efficient methods. This is an example of an iterative roughener
- In terms of performance, the number of operations in Krylov space solvers scales with $N\,log(N)$, where N is the number of unknowns

# Krylov Space Solvers

Mathematical Formalism for Krylov Space Solvers

- The Conjugate Gradient (CG) solver is an orthogonal projection technique onto the Krylov space $\mathcal{K}([A], [r]^{(0)})$:

$$\mathcal{K}([A], [r]^{(0)}) = span([r]^{(0)}, [A][r]^{(0)}, \cdots, [A]^{\nu}[r]^{(0)})$$

- Conjugate Gradient algorithm

CG$([A], [x], [b])$:
$[r]^{(0)} = [b] - [A][x]^{(0)}, \ [p]^{(0)} = [r]^{(0)}$
for $j = 0, 1, \ldots$
$\alpha_j = ([r]^{(j)}, [r]^{(j)})/([A][p]^{(j)}, [p]^{(j)})$
$[x]^{(j+1)} = [x]^{(j)} + \alpha_j [p]^{(j)}$
$[r]^{(j+1)} = [r]^{(j)} - \alpha_j [A][p]^{(j)}$
$\beta_j = ([r]^{(j+1)}, [r]^{(j+1)})/([r]^{(j)}, [r]^{(j)})$
$[p]^{(j+1)} = [r]^{(j+1)} + \beta_j [p]^{(j)}$
end

# Krylov Space Solvers

Mathematical Formalism for Krylov Space Solvers

- CG solver seeks the solution in the Krylov space in the following form:

$$[x]^{(\nu+1)} = [x]^{(0)} + \alpha_\nu [p]^{(\nu)}$$

- Auxiliary vectors $[p]^{(\nu)}$ are chosen to have the following property:

$$\left([A][p]^{(\nu)}, [p]^{(\nu)}\right) = 0$$

- Here, symbol $(\cdot, \cdot)$is the scalar product of two vectors

# Krylov Space Solvers

Preconditioner $[M]$ is a matrix which approximates $[A]$ such that equation

$$[M][x] = [b]$$

may be inexpensive to solve. Then, we solve the following:

$$[M]^{-1}[A][x] = [M]^{-1}[b]$$

$$[A][M]^{-1}[u] = [b],\ [x] = [M]^{-1}[u]$$

PCG$([A], [x], [b])$:

$[r]^{(0)} = [b] - [A][x]^{(0)},\ [z]^{(0)} = [M]^{-1}[r]^{(0)},\ [p]^{(0)} = [z]^{(0)}$

for $j = 0, 1, \ldots$

$\alpha_j = ([r]^{(j)}, [z]^{(j)})/([A][p]^{(j)}, [p]^{(j)})$

$[x]^{(j+1)} = [x]^{(j)} + \alpha_j [p]^{(j)}$

$[r]^{(j+1)} = [r]^{(j)} - \alpha_j [A][p]^{(j)}$

$z^{(j+1)} = [M]^{-1}[r]^{(j+1)}$

$\beta_j = ([r]^{(j+1)}, [z]^{(j+1)})/([r]^{(j)}, [z]^{(j)})$

$[p]^{(j+1)} = [z]^{(j+1)} + \beta_j [p]^{(j)}$

end

# Algebraic Multigrid

Basic Idea of Multigrid

- Mathematical analysis of discretisation shown it makes sense to use coarse-mesh solutions to accelerate the solution process on the fine mesh, through initialisation and coarse correction: Multigrid
- In terms of matrices and linear solvers, the same principle should apply: our matrices come from discretisation! However, it would be impractical to build a series of coarse meshes just to solve a system of linear equations
- We can readily recognise that all the information about the coarse mesh (and therefore the coarse matrix) already exists in the fine mesh

Can we do the same with a linear equation solver?: Algebraic Multigrid (AMG)

- Operation of a multigrid solver relies on the fact that a **high-frequency error** is easy to eliminate: consider the operation of the Gauss-Seidel algorithm
- Once the high-frequency error is removed, iterative convergence slows down. At the same time, the error that looks smooth on the current mesh will behave as high-frequency on a coarser mesh
- If the mesh is coarser, the error is both eliminated faster and in fewer iterations.
- Thus, in multigrid the solution is mapped through a series of coarse levels, each of the levels being responsible for a “band” of error

# Algebraic Multigrid Solver

Algebraic Multigrid Solver: Main Ingredients

- Restriction operator: $[R]_n^{n+1}$
- Prolongation operator: $[P]_{n+1}^n$
- Single-level smoother
- Type of multigrid cycle

Construction of a Coarse Matrix

- Coarse matrix $[A]^{n+1}$ is constructed through projection:

$$[A]^{n+1} = [R]_n^{n+1}[A]^n[P]_{n+1}^n$$

Algebraic Multigrid Solver: $\mu$-Cycle

- Number of pre-sweeps $(\nu_1)$
- Number of post-sweeps $(\nu_2)$
- Smoother type
- Prolongation $([P]_{n+1}^n)$ and restriction operators $([R]_n^{n+1})$
- Type of the cycle $(\mu)$

# Algebraic Multigrid Solver

Algebraic Multigrid Solver: $\mu$-Cycle

$\mu$-Cycle$([x]^n, [r]^n)$:
Create multigrid levels:
$[A]^n$, $[R]_n^{n+1}$, $[P]_{n+1}^n$, $n = 0, 1, 2, ..., N-1$
for $n = 0$ to $N-1$
  $\nu_1$ pre-smoothing sweeps:
  solve $[A]^n[x]^n = [b]^n$, $[r]^n = [b]^n - [A]^n[x]^n$
  if $n \,!= N-1$
    $[b]^{n+1} = [R]_n^{n+1}[r]^n$
    $[x]^{n+1} = 0$
    $[x]^{n+1} = \mu$-Cycle$([x]^{n+1}, [r]^{n+1})$ $\mu$ times
    Correct $[x]_{new}^n = [x]^n + [P]_{n+1}^n[x]^{n+1}$
  end
  $\nu_2$ post-smoothing sweeps:
  solve $[A]^n[x]_{new}^n = [b]^n$
end

# Algebraic Multigrid Solver

Algebraic Multigrid Operations

- **Matrix coarsening**. This is roughly equivalent to creation of coarse mesh cells. Two main approaches are:
  - **Aggregative multigrid (AAMG)**. Equations are grouped into clusters in a manner similar to grouping fine cells to for a coarse cell. The grouping pattern is based on the strength of off-diagonal coefficients
  - **Selective multigrid (SAMG)**. In selective multigrid, the equations are separated into two groups: the **coarse** and **fine** equations. Selection rules specifies that no two coarse points should be connected to each other, creating a maximum possible set. Fine equations form a fine-to-coarse interpolation method (restriction matrix), $[R]$, which is used to form the coarse system.
- **Restriction of residual** handles the transfer of information from fine to coarse levels. A fine residual, containing the smooth error component, is restricted and used as the r.h.s. (right-hand-side) of the coarse system.
- **Prolongation of correction**. Once the coarse system is solved, coarse correction is prolongated to the fine level and added to the solution. Interpolation introduces **aliasing errors**, which can be efficiently removed by smoothing on the fine level.

# Algebraic Multigrid

Algebraic Multigrid Operations

- **Multigrid smoothers**. The bulk of multigrid work is performed by transferring the error and correction through the multigrid levels. Smoothers only act to remove high-frequency error: simple and quick. Smoothing can be applied on each level:
  - Before the restriction of the residual, called **pre-smoothing**
  - After the coarse correction has been added, called **post-smoothing**
- Algorithmically, post-smoothing is more efficient
- **Cycle types**. Based on the above, AMG can be considered a two-level solver. In practice, the "coarse level" solution is also assembled using multigrid, leading to multi-level systems.
- The most important multigrid cycle types are
  - **V-cycle**: residual reduction is performed all the way to the coarsest level, followed by prolongation and post-smoothing. Mathematically, it is possible to show that the V-cycle is optimal and leads to the solution algorithm where the number of operations scales linearly with the number of unknowns
  - **Flex cycle**. Here, the creation of coarse levels is done on demand, when the smoother stops converging efficiently
- Other cycles, *e.g.* W-cycle or F-cycle are a variation on the V-cycle theme