Advanced Programming for Scientific Computing (PACS)
Lecture title: Solution of large linear systems

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2020/2021

The issue

Numerical methods for the solution of partial differential equations eventually lead to the solution of linear systems

$$\mathrm{A}\boldsymbol{x} = \boldsymbol{b}$$

where A is usually large and sparse

Storing a matrix

In a computer matrix values are generally stored as a linear structure in contiguous memory (i.e. like a vector). Since matrix values are stored consecutively, it is important to distinguish *rowmajor* and *columnmajor* ordering:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Rowmajor	[1,2,3,4,5,6,7,8,9]
Columnmajor	[1,4,7,2,5,8,3,6,9]

an example

An example of full matrix where the user can choose among different storage ordering is in MyMat0_templates/MyMat0.hpp. It is a template matrix and argument tag technique is used to select the appropriate operations according to the storage ordering type. However, in this course we prefer to use the Eigen library.

Sparse matrices

A sparse matrix is a matrix whose number of non-zero elements N_{nz} is $O(N)$, N being the size of the matrix. In other words is a matrix where the average number of non-zero elements per row is constant w.r.t. the matrix size. We will indicate this constant as m.

It is then convenient to store only the non-zero elements.

Storing sparse matrices

Details on sparse matrix storage may be found in the book of Y. Saad.

We recall the general ideas. We need to distinguish among

- dynamic storage system: it is possible to add new non-zero elements at any time with low cost. This type of storage system is normally used only in the process of assembling the matrix. For efficiency reason is then converted to a static (compressed) storage system.
- semi-dynamic storage system. It uses a the same data structure as a static storage system, yet it leaves some "empty spaces" to make the insertion of new elements faster (until there is no empty space left). It's the strategy used by Eigen.
- static storage system. The pattern of non-zero elements is fixed. It is not possible (or at least not easy) to add a new non-zero element. Normally operations like matrix-vector product are more efficient with this format.

An example of dynamic storage: a map

A possibility is to use a `map` $(i,j) \to \mathrm{A}_{ij}$. In practice the map is build either by using a hash table or a binary tree. In C++ the first is a unordered_map<>, the second a map<> Example:

```
class SimpleDynMatrix
{
  ....
  private:
  std::map<tuple<int,int>,double> m_A;
};
```

If traversed elements are given in rowmajor ordering.... why?

Static storage

There are several formats, the most common ones are

- CSR Compressed sparse rows. Matrix is stored row major. As a consequence accessing an entire row is an $O(1)$ operation, while random access to an element is $O(m)$. Used by `Eigen`.
- CSC. Compressed sparse columns. Matrix is stored column major. Access to an entire row is $O(1)$. Used by `Eigen`
- MSR. Modified sparse row. As CSR but more compact. Easy access to diagonal elements. Only square matrices. Used by the suite `Aztec` of Trilinos.
- COO. Basically a `vector<tuple<int,int>,double>` storing (i,j) and the corresponding A_{ij}.

Variants that store only half matrix are available for symmetric matrices.

CSR and CSC formats

It is the only we describe in more detail since are the ones used in Eigen sparse matrices (in *compressed*, i.e. non-dynamic state). Different variants are possible. I present that of *Eigen*. The difference is rowmajor (CSR) or colummajor (CSC) ordering.
We have three vectors

- `Values` The non-zero values in columnmajor/rowmajor ordering (of size `nonZeroes()`);
- `InnerIndex` The index of the row (columnmajor) or column (rowmajor) of the value (of size `nonZeroes()`);
- `InnerNNZs` Number of non-zeros elements in columns/rows (of size `row()`/`col()`).

An example

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & 1 \\ 2 & 0 & 5 \end{bmatrix}$$

Column Major:

`Values:`1,2,2,3,1,5 `InnerIndex:`0,2,1,0,1,2 `InnerNNZs:`2,1,3

Row Major:

`Values:`1,3,2,1,2,5 `InnerIndex:`0,2,1,2,0,2 `InnerNNZs:`2,2,2

Sparse Matrices and Graphs

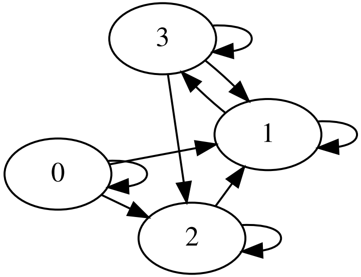
A sparse matrix $A \in \mathbb{R}^{m \times n}$ is associated to a directed graph, called adjacency graph or adjacency matrix $G = G(V, E)$, where $V = v_0, \ldots, v_{n-1}$ are the vertices representing the "unknowns", of indexes $(0, \ldots n-1)$, while $E = \{(i_0, j_0), (i_1, j_1), \ldots\}$ is the set of edges with cardinality N_{nz} and such that

$$(i, j) \in E \leftrightarrow A_{ij} \text{ may be different from } 0$$

The graph of the matrix is usually determined by the underlying problem. For square matrices, an adjacency graph is said to be symmetric if $(i, j) \in E \leftrightarrow (j, i) \in E$. In this case the graph may be considered un-directed.

Adjacency graph, a simple example

$$\begin{bmatrix} 2.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 4.0 & 0.0 & 3.0 \\ 0.0 & 1.0 & 5.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 2.0 \end{bmatrix}$$



Note that the graph presents loops with edges having origin and target on the same graph node, to represent non-empty diagonal entries. Sometimes the diagonal entries are always stored, so in the adjacency graph we may omit them.

Bandwidth of a (sparse) matrix

If $v_j \in G$ is a vertex of the adjacency graph (of index j) we indicate with $\mathrm{ad}(v_j)$ or simply $\mathrm{ad}(j)$ the set of vertices adjacent to v_j (connected by an outgoing edge). The cardinality of $\mathrm{ad}(v_j)$ is the degree of v_j (the number of non-zeros entries nnz_j in row j). The quantity $b_j = 1 + \max(|k - j|, \, k \in \mathrm{ad}(j))$ is the width of node v_j (or row bandwidth).

The bandwidth b of a matrix A is

$$b = \max(b_j, \, j = 1, \ldots n).$$

Example: for a tridiagonal matrix $b = 3$. For a diagonal matrix $b = 1$. For a full matrix $b = n$, the size of the matrix.

Direct methods

Direct methods for general matrices are mostly based on Gaussian elimination which is strictly related to the $LU(\mathcal{P}, \mathcal{Q})$ decomposition (LU with full pivoting): for any matrix A there exist two permutation matrices $\mathcal{P}$, $\mathcal{Q}$, a lower triangular matrix L and a upper triangular matrix U such that

$$\mathcal{P}\mathrm{A}\mathcal{Q} = \mathrm{LU}$$

The solution of the linear system implies the sequence of operation

$$\boldsymbol{c} = \mathcal{P}\boldsymbol{b} \quad \mathrm{L}\boldsymbol{y} = \boldsymbol{c} \quad \mathrm{U}\boldsymbol{z} = \boldsymbol{y} \quad \boldsymbol{x} = \mathcal{Q}\boldsymbol{z}$$

The second and third systems can be solved with a $O(n^2)$ method, called forward and back-substitution. If $\mathcal{Q}$ is the identity matrix, we have just *row pivoting*, also called *left pivoting*. If $\mathcal{Q} = \mathcal{P}^T$ we have a *symmetric reordering*.

spd matrices

For a sdp matrix it is convenient to perform the Cholesky factorization (which is a particular case of LU):

$$\mathrm{A} = \mathrm{H}^{\mathrm{T}}\mathrm{H}$$

where H is upper-triangular (of course you may also do $\mathrm{A} = \mathrm{H}\mathrm{H}^{\mathrm{T}}$). A note: the stability of the algorithm depends on the

2-norm of the matrix. If the matrix is badly conditioned round-off errors may produce null or negative pivots. In that case the algorithm fails. A possible remedy is to replace A by $\mathrm{A} + \alpha\mathrm{I}$ where α is a small number and I the identity matrix.

Disadvantages of standard Gauss elimination for sparse matrices

Gauss elimination (LU factorization) is the most efficient direct method for general full matrices ($O(n^3)$) and if pivoting is performed its stability does not depend on the condition number and is, in general, good.

But, unfortunately, the L and U factors of a sparse matrix are not, in general, sparse!. We have the so-called fill-in.

For large sparse matrices standard Gauss elimination is thus unfeasible for memory limitation.

Fill-in

The fill-in of a matrix are those entries which change from an initial zero to a non-zero value during the execution of an algorithm. So we need extra memory to store the new non-zero entries.
For what concerns the solution of $A\mathrm{x} = \mathrm{b}$ via a Gauss elimination (LU factorization) type procedure the basic idea is to find two permutation matrices $\mathcal{P}$ and $\mathcal{Q}$ so that the system

$$\tilde{\mathrm{A}}\mathrm{z} = \mathcal{P}\mathrm{b}$$

with $\tilde{\mathrm{A}} = \mathcal{P}\mathrm{A}\mathcal{Q}$, can be factored with small fill-in. In general, what we want is that $\tilde{A}$ has a small bandwidth (elements clustered around the diagonal).

This is done by analyzing the adjacency graph of A.

Permutation matrices

A permutation matrix P is an orthonormal matrix ($P^{-1} = P^T$) obtained by exchanging rows (or columns) of the identity. You do not need to build the matrix P, but have a vector π of integers, which defines the map $i \to \pi(i)$ of the indices:

$$P_{ij} = \delta_{i,\pi(j)}$$

Pre-multiplying by P exchanges row i with row $\pi(i)$,
post-multiplying by P^T exchanges column j with πj

$$\hat{A} = PA \Rightarrow \hat{A}_{ij} = A_{\pi(i)j} \quad \tilde{A} = AP^T \Rightarrow \hat{A}_{ij} = A_{i\pi(j)}$$

Of particular relevance is the symmetric reordering

$$\tilde{A} = PAP^T \Rightarrow \tilde{A}_{ij} = A_{\pi(i)\pi(j)}$$

Note that column-reordering corresponds in changing indexes of the vertices of the adjacency graph, while row-reordering leaves the graph unchanged.

Permutation in Eigen

In the Eigen library you can construct a special matrix called `PermutationMatrix<>` to store a permutation. Or you can interpret any Eigen vector containing integers that represent a permutation as a permutation matrix. A special command is available for symmetric permutations.

```
using namespace Eigen;
Matrix<int,Dynamic,1> indices;
//  fill the indices
//  apply permutation to matrix M and store it in R
R = M.twistedBy(indices.asPermutation());
```

Reordering techniques for sparse matrices

The ideas of reordering is to construct a permutation matrix P so that PAP^T guarantees lowest fill-in in a direct method, or the best "cache friendliness" in a matrix-times-vector operation. In practice you want to reduce the bandwidth of the matrix.

One of the most celebrated algorithm is the Reverse Cuthill-McKee (RCMK):

RCMK

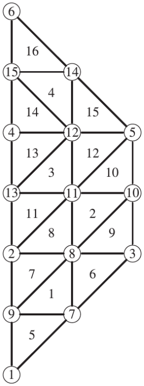
There are different implementations of the RCMK algorithm, and the algorithm is implemented in the boost graph library. I give here the general idea. R is the vector that will store the permuted indexes, initially empty.
Choose a vertex v_j in the graph with a low degree. Set $R = \{j\}$, and $i = 0$

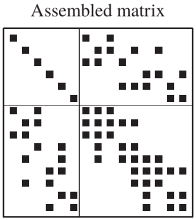
```
1. While card($R$) $< n$;
    1.1 Let $k = R_i$, $i \leftarrow i + 1$
    1.2 Set $A_k = \mathrm{ad}(k) \cap R$, ordered from the lowest to the highest
        degree;
    1.3 Set $R \leftarrow \{R, A_k\}$ (append $A_k$ to $R$);
2. Reverse the order in $R$ (unless you want standard CMK).
```

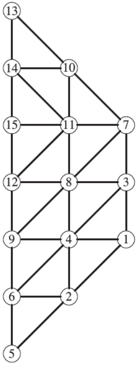
The permutation matrix is $P_{ij} = \delta_{R(i),j}$.

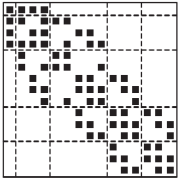
A note

The previous algorithm may be implemented using `sets` or a `priority_queue`, where the order relation is the degree of the node. It is a particular *breadth first search* trasversal of the graph. It is called "reversed" since in the original version in step 1.2 we have $Q \leftarrow \{Q, k\}$, i.e. index k is appended to Q.









Before

After RCMK reordering

Direct methods for sparse matrices

A direct algorithm for sparse linear systems is the multifrontal algorithm, implemented in the open-source libraries UMFPACK (part of the Sparse Suite package) and MUMPS, and SuperLU which executes a LU factorization after reordering to reduce fill-in. The last two are available also for parallel architectures. They employ a two step procedure

- A symbolic analysis of the adjacency graph of the matrix to identify the order of elimination that minimizes fill-in.
- The elimination phase. The actual Gauss-type elimination is performed.

The algorithm does not make useless operations (like multiplication or addition of zeros), so it is rather efficient, yet some fill-in is inevitable.

Note: If the memory of your computer allows you to use direct methods, use them! In general, they are more efficient than any iterative method. The `Eigen` lib. has interfaces for `UMFPACK` and `SuperLU`.

The Eigen interface

The Eigen interface to direct solvers employs a common paradigm. In the following SolverClassName<Matrix> is the name of one of the interfaces offered by the library, for instance SparseLU<>. You have to include the corresponding header file.

```
#include <Eigen/RequiredModuleName>
using namespace Eigen;
// ...
SparseMatrix<double> A;
// fill A
A.makeCompressed(); // Make sure it is compressed
VectorXd b, x;
// fill b
// solve Ax = b
SolverClassName<SparseMatrix<double> > solver;
solver.compute(A); // Preform analysis for factorization
if(solver.info()!=Success) { // decomposition failed
  return;
}
x = solver.solve(b);
if(solver.info()!=Success) {
  // solving failed
  return;
}
// solve for another right hand side:
x1 = solver.solve(b1);
```

If you have more matrices with the same adjacency graph, you can analyze the graph only once, saving time:

```
SolverClassName<SparseMatrix<double> > solver;
// for this step the numerical values of A are not used
solver.analyzePattern(A);
solver.factorize(A);
x1 = solver.solve(b1);
x2 = solver.solve(b2);
...
// modify the values of the nonzeros of A,
// the nonzeros pattern must stay unchanged
A = ...;
solver.factorize(A);
x1 = solver.solve(b1);
x2 = solver.solve(b2);
...
```

Iterative methods: the basic

We present the most basic iterative method, Richardson iteration, which encompasses some classical ones (Jacobi, Gauss-Siedel) because we can infer some basic properties that are relevant also for more advanced method.

Let M be an easily invertible matrix, called preconditioner (we will give details later). The preconditioned Richardson iteration reads:

Set $\boldsymbol{r}_0 = \boldsymbol{b}$, $\boldsymbol{x}_0 = 0$ and for $i = 0, \dots$

- Compute $\boldsymbol{z}_i$ solution of $\mathrm{M}\boldsymbol{z}_i = \boldsymbol{r}_i$.
- $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i \boldsymbol{z}_i$

Where α_i are positive coefficients. If $\alpha_i = \alpha$ the method is stationary, in the simplest case $\alpha_i = 1$. If $\mathrm{M} = \mathrm{I}$ we have no preconditioning.

Preconditioner

Let's note that the preconditioned Richardson iteration is in-fact equal to the non preconditioned version on the modified system

$$\widehat{\mathrm{A}}\boldsymbol{x} = \widetilde{\boldsymbol{b}}$$

with $\widehat{\mathrm{A}} = \mathrm{M}^{-1}\mathrm{A}$ and $\widetilde{\boldsymbol{b}} = \mathrm{M}^{-1}\boldsymbol{b}$. $\widehat{\mathrm{A}}$ will be called preconditioned matrix.

What we are presenting here is the left preconditioning. In general you have two matrices M^{L} and M^{R} and $\widehat{\mathrm{A}} = (\mathrm{M}^{L})^{-1}\mathrm{A}(\mathrm{M}^{R})^{-1}$

M is usually not inverted: the system $\mathrm{Mz} = \mathrm{r}$ is solved instead.

If A is s.p.d. and M is non-singular and symmetric, then $\mathrm{M}{-}1\mathrm{AM}^{-1}$ is s.p.d.

The role of the preconditioner

To understand the role of the preconditioner let's take $\alpha_i = 1$ for simplicity and write the Richardson iteration in terms of the preconditioned system.

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + (\widetilde{\boldsymbol{b}} - \widehat{\mathrm{A}}\boldsymbol{x}_i) = \boldsymbol{x}_i + \widetilde{\boldsymbol{r}}_i$$

By simple manipulation we note that $\widetilde{\boldsymbol{r}}_{i+1} = (\mathrm{I} - \widehat{\mathrm{A}})^{\mathrm{i+1}}\widetilde{\boldsymbol{r}}_0$ So

$$\widetilde{\boldsymbol{r}}_{i+1} = \mathcal{Z}_{i+1}(\widehat{\mathrm{A}})\widetilde{\boldsymbol{r}}_0 = \mathcal{Z}_{i+1}(\widehat{\mathrm{A}})\widetilde{\boldsymbol{b}}$$

where $\mathcal{Z}_i(y) = \sum_{k=0}^{i}(1-y)^k$ is a polynomial of degree i with $\mathcal{Z}_i(0) = 1$ (monic polynomial). And

$$\boldsymbol{x} - \boldsymbol{x}_{i+1} = \mathcal{Z}_{i+1}(\widehat{\mathrm{A}})\boldsymbol{x}$$

Important Note: Here and in the following we often take for simplicity $\boldsymbol{x}_0 = 0$. This does not cause any loss of generality since the case $\boldsymbol{x}_0 \neq 0$ is easily recovered by a simple shift.

What do we learn from this simple example?

Let's assume that $\widehat{\mathrm{A}}$ has N eigenvectors $\boldsymbol{w}_i$ with corresponding eigenvalues λ_i. We can expand $\widetilde{\boldsymbol{r}}_0 = \widetilde{\boldsymbol{b}} = \sum_{i=1}^{N} \gamma_i \boldsymbol{w}_i$. Using the well known fact that $\mathcal{Z}(\widehat{\mathrm{A}})\boldsymbol{w}_i = \mathcal{Z}(\lambda_i)\boldsymbol{w}_i$ we have

$$\widetilde{\boldsymbol{r}}_i = \mathcal{Z}(\widehat{\mathrm{A}})\widetilde{\boldsymbol{r}}_0 = \sum_{j=1}^{N} \gamma_j \mathcal{Z}_i(\lambda_j)\boldsymbol{w}_j$$

Thus $\|\widetilde{\boldsymbol{r}}_i\| \leq \max_{j=1}^{N} |\mathcal{Z}_i(\lambda_J)| \sum_{j=1}^{N} |\gamma_j| \|\boldsymbol{w}_j\|$.

Small error if $\mathcal{Z}_i(x) = \sum_{k=1}^{i} (1-x)^i$ is small in correspondence to the eigenvalues of the preconditioned matrix.

In conclusion

Even if the previous example refers to a very simple (and often ineffective) iterative procedure, it allows us to identify some desirable property of a preconditioner.

- The preconditioned matrix should have eigenvalues clustered around 1 (in more sophisticated iterative schemes it may be sufficient that they are clustered around a small set of points in the complex plane).
- We should be able to solve the preconditioned system $\mathrm{M}\boldsymbol{z} = \boldsymbol{r}$ with much less effort than the original system $\mathrm{Ax} = \mathrm{b}$
- The preconditioner should be simple to build.

Some classic preconditioners

- Jacobi or diagonal preconditioner: $\mathrm{M} = \mathrm{diag}(A)$
- Block Jacobi. If we are able to partition the matrix as

$$\mathrm{A} = \begin{bmatrix} A_{11} & A_{12} & \dots \\ A_{21} & A_{22} & \dots \\ \dots & \dots & \dots \end{bmatrix}$$

  with A_{ij} "small" matrices (that can be inverted by a direct method), we can choose $\mathrm{M} = \mathrm{diag}(A_{11}, A_{22}, \dots)$. Often used in a parallel setting.
- Gauss Siedel. If $\mathrm{A} = D - E - F$ with D $-E$ and $-F$ the diagonal, strict lower and strict upper part, respectively. Then the GS preconditioner is $P = D - E$.

Some classic preconditioners

- Symmetric Gauss Siedel. Since the traditional Gauss Siedel gives a non-symmetric M for a symmetric A, a variant is $P = (D - E)D^{-1}(D - F)$, which is symmetric if A is symmetric.
- SOR. Successive over relaxation preconditioner. $\mathrm{M} = \frac{1}{\omega}D - E$, where ω is a relaxation parameter that has to be chosen carefully. If A is spd, if we have an estimate of $\xi = \rho(\mathrm{I} - \mathrm{D}^{-1}\mathrm{A}) = \max_{\mathrm{i}} |\lambda_{\mathrm{i}}((\mathrm{I} - \mathrm{D}^{-1}\mathrm{A}))|$ the optimal value for ω is $2/(1 + \sqrt{1 - \xi^2})$

We have also a symmetric variant of SOR. Symmetric SOR is particularly effective for SPD matrices when one has a good estimate ω.

Some less classic preconditioners

- Incomplete LU factorization, also called ILU(n). A very popular preconditioner. An approximate factorization $\mathrm{A} \simeq \hat{\mathrm{A}} = \hat{\mathrm{L}}\hat{\mathrm{U}}$ is constructed with the property that $\hat{\mathrm{A}}$ has at most $N_{nz} + nN$ non-zero entries (n is called fill-in factor). Then, $\mathrm{M} = \hat{\mathrm{L}}\hat{\mathrm{U}}$. It is based on LU factorization where elements are dropped. There are different strategies to do it. When it works can be very effective. Unfortunately it is not guaranteed to work always. If $n = 0$ the factored matrix has the same non-zero entries than the original one.

  A variant, called ILUT(n,τ) allows to prescribe a threshold tolerance for the selection of entries to be dropped.
  For spd matrices one can perform an Incomplete Cholesky Factorization.

Some less classic preconditioners

- Polynomial preconditioners. $\mathrm{M}^{-1} = p(\mathrm{A})$ where p is a polynomial (of low degree). If we set $\mathrm{A} = D - C$ if $\rho(CD^{-1}) < 1$ a possible polynomial preconditioner is given by

  $$\mathrm{M}^{-1} = D^{-1}(\mathrm{I} + \mathrm{CD}^{-1} + (\mathrm{CD}^{-1})^2 2 + \dots)$$

  Note that we can avoid computing the inverse of the preconditioner explicitly.

- Multilevel preconditioners. Let $I_N^M : \mathbb{R}^N \to \mathbb{R}^M$ be a restriction operator such that for a $\boldsymbol{v} \in \mathbb{R}^N$ we have $I_N^M \boldsymbol{v} = \boldsymbol{w} \in \mathbb{R}^M$, with $M << N$. Let $I_M^N = (I_N^M)^T$ Then we may choose $\mathrm{M}^{-1} = I_M^N [I_M^N \mathrm{A} I_N^M]^{-1} I_N^M$. Matrix $\mathrm{A}_M = I_M^N \mathrm{A} I_N^M$ is of size M, if it is sufficiently small we can use a direct method to solve the system in A_M, otherwise we apply the same technique recursively. There are several ways of building the restriction operator, the simplest one is agglomeration.

Some less classic preconditioners

- Domain decomposition preconditioners. They are derived from the differential problem by partitioning the domain into subdomains Ω_i and transforming the original differential problems in problems on each subdomain plus interface condition. The preconditioner is then obtained by solving the local problems independently (in a sort of Jacobi or Gauss-Siedel fashion).

  Their detailed explanation goes beyond this notes, if you are interested you may look at the books of Quarteroni and Valli (1999), Toselli Widlund (2005) and Smith, Bjorstad, Gropp (2004).

A general idea to construct iterative solvers

Let $\mathcal{K}_m$ and $\mathcal{L}_m$ be two subspaces of $\mathbb{R}^n$ of dimension $m \leq n$. We may construct an approximate solution $\mathrm{x}_m \in \mathrm{x}_0 + \mathcal{K}_m$ through a (oblique in general) projection procedure on the residual: *Find* $\mathrm{x}_m \in \mathrm{x}_0 + \mathcal{K}_m$ *such that*

$$\boldsymbol{r}_m = \boldsymbol{b} - \mathrm{A}\boldsymbol{x}_m \perp \mathcal{L}_m$$

That is,

$$(\boldsymbol{r}_m, \boldsymbol{w}) = 0 \quad \forall \boldsymbol{w} \in \mathcal{L}_m.$$

Matrix-wise, if $\mathrm{V}_m = \left[\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m\right]$, $\mathrm{W}_m = \left[\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m\right]$, contain a basis of $\mathcal{K}_m$ and $\mathcal{L}_m$, then $\boldsymbol{x}_m = \boldsymbol{x}_0 + \mathrm{V}_m\boldsymbol{y}_m$ with

$$(\mathrm{W}_m\mathrm{A}\mathrm{V}_m)\boldsymbol{y}_m = \mathrm{W}_m^T\boldsymbol{r}_0$$

We could construct a succession of approximating spaces of increasing dimension until we are satisfied. We need find a good approximating spaces (and make the computation less costly!).

The Krylov space

We may note that $\mathcal{Z}_i(\mathrm{A})\boldsymbol{r}_0 = \sum_{k=1}^{i}(\mathrm{I}-\mathrm{A})^{\mathrm{i}}\boldsymbol{r}_0$ is a vector that may be written as

$$\mathcal{Z}_i(\mathrm{A})\boldsymbol{r}_0 = \boldsymbol{r}_0 + a_1\mathrm{A}\boldsymbol{r}_0 + a_2\mathrm{A}^2\boldsymbol{r}_0 + \dots$$

In other words $\mathcal{Z}_i(\mathrm{A})\boldsymbol{r}_0 \in \mathrm{span}(\boldsymbol{r}_0, \mathrm{A}\boldsymbol{r}_0, \dots, \mathrm{A}^i\boldsymbol{r}_0)$

The space $\mathcal{K}_i = \mathcal{K}_i(\mathrm{A}; \boldsymbol{r}_0) = \mathrm{span}(\boldsymbol{r}_0, \mathrm{A}\boldsymbol{r}_0, \dots, \mathrm{A}^{i-1}\boldsymbol{r}_0)$ is the Krylov space of dimension i generated by A and $\boldsymbol{r}_0$. It plays a fundamental role in iterative methods for linear system!

An interesting property. If $\boldsymbol{x}_i - \boldsymbol{x}_0 \in \mathcal{K}_i$ then $\boldsymbol{r}_i \in \mathcal{K}_{i+1}$. Indeed $\boldsymbol{r}_i = \boldsymbol{b} - \mathrm{A}\boldsymbol{x}_i = \boldsymbol{r}_0 + \mathrm{A}(\boldsymbol{x}_i - \boldsymbol{x}_0)$.

Krylov based methods

We can have different strategies:

- Ritz-Galerkin. Orthogonal projection method: $\mathcal{L}_i = \mathcal{K}_i$. For a spd matrix A it is **equivalent** to look for the approximation

  $$\boldsymbol{x}_i = \text{argmin}_{\boldsymbol{y} \in \boldsymbol{x}_0 + \mathcal{K}_i} \|\boldsymbol{y} - \boldsymbol{x}\|_A = \text{argmin}_{\boldsymbol{y} \in \boldsymbol{x}_0 + \mathcal{K}_i} (\boldsymbol{y} - \boldsymbol{x})^T \mathrm{A} (\boldsymbol{y} - \boldsymbol{x})$$

  Methods of this class are: Conjugate Gradient and the Lanczos method.

- *minimum norm residual*. Here $\mathcal{L}_i = A\mathcal{K}_i$. It is **equivalent** to look for the approximation

  $$\boldsymbol{x}_i = \text{argmin}_{\boldsymbol{y} \in \boldsymbol{x}_0 + \mathcal{K}_i} \|\boldsymbol{b} - \mathrm{A}\boldsymbol{y}\|_2$$

  Of this class we have GMRES (generalized minimal residual), MINRES, ORTHODIR.

Krylov based methods

- *Petrov-Galerkin*. Here $\mathcal{L}_i \neq \mathcal{K}_i$ (so minimum norm residual methods are of this class) but the more special case is when $\mathcal{L}_i = A^T \mathcal{K}_i$. Of this class we have QMR and Bi-CG.

Hybrid methods that do not fall in those categories are CGS (conjugate gradient squared), BiCG-STAB (Bi-conjugate gradient stabilized) and FGMRES (flexible gmres).

It's clear that it may be convenient to have an orthogonal base for $\mathcal{K}_i$. However, we will skip this important issue, interested students may look, for instance, to the book of Y. Saad.

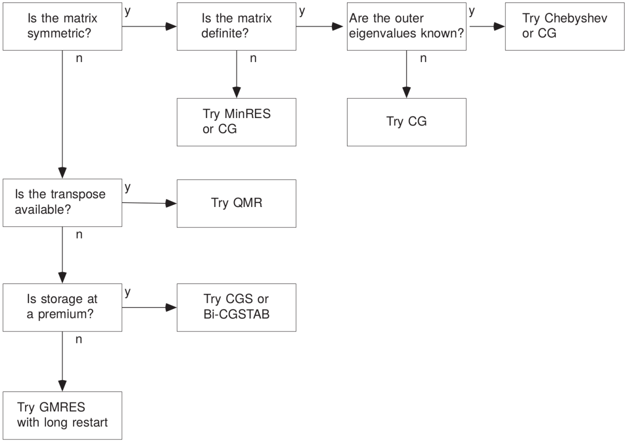
A note on preconditioning

We have already mentioned that preconditioning is essential for good convergence of iterative methods when the system is badly conditioned, and badly conditioned systems happen frequently unfortunately.

In a preconditioned method the Krylov space that of the preconditioned matrix, and the residual that of the preconditioned system. However how to apply preconditioning depends on whether we are dealing with symmetric matrices or not. With symmetric matrices not to loose symmetry we need symmetric preconditioning, where the preconditioned system is $\mathrm{M}^{-1}\mathrm{A}\mathrm{M}^{-T}\boldsymbol{y} = \mathrm{M}^{-T}\boldsymbol{b}$ (remember that we will never build the preconditioned matrix! What we need is to compute solutions of systems of the form $\mathrm{M}\boldsymbol{z} = \boldsymbol{r}$).

An overview af the main iterative methods

There are many other methods. We give an overview with hints:



Preconditioned conjugate gradient method

The (preconditioned) conjugate gradient method is the method of choice for spd systems.

Given an sdp preconditioning matrix M and its Chowlesky factorization $\mathrm{M} = \mathrm{L}\mathrm{L}^T$ the preconditioned iterates are equivalent to solve the preconditioned system

$$\widehat{\mathrm{A}}\boldsymbol{x} = \mathrm{L}^{-1}\mathrm{A}\mathrm{L}^{-T}\mathsf{y} = \mathrm{L}^{-1}\mathsf{b}, \quad \mathsf{y} = \mathrm{L}^T\mathsf{x}$$

Conjugate Gradient method

For simplicity we set $\mathrm{x}^{(0)} = 0$. The conjugate gradient method is a Ritz-Galerkin method that at iteration k computes

$$\boldsymbol{x}^{(k)} = \mathrm{argmin}_{\mathrm{z} \in \mathcal{K}_k} \|\mathrm{x} - \mathrm{z}\|_{\widehat{\mathrm{A}}}.$$

which is equivalent to say that $\boldsymbol{b} - \mathrm{A}\boldsymbol{x}^{(k)} = \boldsymbol{r}_k \perp \mathcal{K}_k(\widehat{\mathrm{A}}, \boldsymbol{r}_0)$.
But, luckily we do not have to store a basis for the whole Krylov subspace, but thanks to a special recurrence relation of its coefficients the last three are enough (two in fact, since one may be overwritten). So, the CG scheme is particularly efficient from the computational point of view.

The algorithm

```
Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
**for** $i = 1, 2, \ldots$
    **solve** $Mz^{(i-1)} = r^{(i-1)}$
    $\rho_{i-1} = {r^{(i-1)}}^T z^{(i-1)}$
    **if** $i = 1$
      $p^{(1)} = z^{(0)}$
    **else**
      $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
      $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
    **endif**
    $q^{(i)} = Ap^{(i)}$
    $\alpha_i = \rho_{i-1}/{p^{(i)}}^T q^{(i)}$
    $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
    $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
    check convergence; continue if necessary
**end**
```

Preconditioned CG

The fact that we do not have to construct L is due to the fact that, if M is spd matrix, even if the left preconditioned matrix $\widehat{\mathrm{A}} = \mathrm{M}^{-1}\mathrm{A}$ is symmetric with respect to the scalar product $(\boldsymbol{x}, \boldsymbol{y})_{\mathrm{M}} = \boldsymbol{x}^T \mathrm{M} \boldsymbol{y}$. Indeed

$$(\boldsymbol{x}, \widehat{\mathrm{A}}\boldsymbol{y})_{\mathrm{M}} = \boldsymbol{x}^T \mathrm{M}\mathrm{M}^{-1}\mathrm{A}\boldsymbol{y} = (\mathrm{M}^{-1}\mathrm{A}\boldsymbol{x})^T \mathrm{M}\boldsymbol{y} = (\widehat{\mathrm{A}}\boldsymbol{x}, \boldsymbol{y})_{\mathrm{M}}$$

Thus, the basic algorithm has been modified for the preconditioned case by just changing the scalar products!

Convergence properties of preconditioned CG

The i-th iteration of the conjugate gradient satisfies

$$\boldsymbol{x}_i - \boldsymbol{x} = \mathcal{Q}_i(\widehat{\mathrm{A}})(\boldsymbol{x}_0 - \boldsymbol{x})$$

where $\mathcal{Q}_i$ minimizes $\|\boldsymbol{x}_i - \boldsymbol{x}\|_A$ among all polynomials $\mathcal{W}_i$ of degree i and such that $\mathcal{W}_i(0) = 1$.
As a consequence we have this interesting property: let $\boldsymbol{w}_i$ be the i-th eigenvector of $\widehat{\mathrm{A}}$ and λ_i the corresponding eigenvalue. We can always write $\boldsymbol{b} = \sum_i \gamma_i \boldsymbol{w}_i$ (some γ_i may be zero). Then

$$\|\boldsymbol{x}_i - \boldsymbol{x}\|^2_{\mathrm{A}} = \sum_{j=1}^{N} \frac{\gamma_j^2}{\lambda_j} \mathcal{Q}_i^2(\lambda_j) \leq \max_{\substack{j=1\ldots,N \\ \gamma_j \neq 0}} \mathcal{W}_i^2(\lambda_j) \sum_{j=1}^{N} \frac{\gamma_j^2}{\lambda_j}$$

where $\mathcal{W}_i$ is any polynomial of degree i that takes the value 1 at 0.

The consequences

If m is the number of $\gamma_i \neq 0$ we have that In exact arithmetic the CG algorithm converges to the exact solution in m steps.

But more importantly: If l is the number of different eigenvalues of $\widehat{\mathrm{A}}$ then in exact arithmetic the CG algorithm converges to the exact solution in l steps.

Thus the role of the preconditioner in then to cluster the eigenvalues of the preconditioned matrix.

By exploiting some properties of polynomials we have also the classical result

$$\|\boldsymbol{x}_i - \boldsymbol{x}\|_{\mathrm{A}}^2 = 2\left(\frac{\sqrt{K(\widehat{\mathrm{A}})} - 1}{\sqrt{K(\widehat{\mathrm{A}})} + 1}\right)^i \|\boldsymbol{x}_0 - \boldsymbol{x}\|_{\mathrm{A}}^2$$

Stopping criteria

Normally the stopping criteria is given in terms of the relative residual:

$$\|\boldsymbol{r}^{(k)}\|/\|\boldsymbol{r}^{(0)}\| \leq \epsilon$$

Since

$$\|\mathrm{e}^{(k)}\|_{\mathrm{A}}/\|\mathrm{e}^{(0)}\|_{\mathrm{A}} = \sqrt{\mathrm{cond}(\mathrm{A})}\|\boldsymbol{r}^{(k)}\|/\|\boldsymbol{r}^{(0)}\|$$

the relative residual is linked to the error in the A-norm, which for system arising from the solution of elliptic problems by finite elements is related to the error in the energy norm.

Practical rule: the error should be of the same order of that introduced by the discretization. For linear finite elements we should stop when $\|\mathrm{e}^{(k)}\|_{\mathrm{A}} \leq \tau h$, where $\tau > 0$ is a constant (related to $\|D^2 u\|/\|\nabla u\|$ and the interpolation constant). Using the previous relation and $\mathrm{cond}(\mathrm{A}) = Ch^{-2}$ for linear finite elements

$$\|\boldsymbol{r}^{(k)}\|/\|\boldsymbol{r}^{(0)}\| \leq \epsilon h.$$

where ϵ is a user selected tolerance (for instance 10^{-4}).

MINRES

An algorithm (called Lanczos) has been originally devised to construct approximations of the eigenvalues of general symmetric A as the eigenvalues of T_k yet it is the basis for the MINRES method. It's a Petrov-Galerkin projection methods, with $\mathcal{L}_k = \mathrm{A}\mathcal{K}_k$, so, it is equivalent to minimize $\|\boldsymbol{r}^{(k)}\|_2$. In its preconditioned version we minimize the preconditioned residual. Also here, the existence of a recurring relation for the basis of the Krylov subspaces T_k makes it possible to build a computationally efficient method. It is often the method of choice for symmetric indefinite systems, like those arising from saddle point problems. The preconditioner must be symmetric and positive definite.

Algorithm 2.4: The Minres Method

```
$\mathbf{v}^{(0)} = \mathbf{0}, \mathbf{w}^{(0)} = \mathbf{0}, \mathbf{w}^{(1)} = \mathbf{0}$
Choose $\mathbf{u}^{(0)}$, compute $\mathbf{v}^{(1)} = \mathbf{f} - A\mathbf{u}^{(0)}$, set $\gamma_1 = \|\mathbf{v}^{(1)}\|$
Set $\eta = \gamma_1, s_0 = s_1 = 0, c_0 = c_1 = 1$
for $j = 1$ until convergence do
    $\mathbf{v}^{(j)} = \mathbf{v}^{(j)}/\gamma_j$
    $\delta_j = \langle A\mathbf{v}^{(j)}, \mathbf{v}^{(j)}\rangle$
    $\mathbf{v}^{(j+1)} = A\mathbf{v}^{(j)} - \delta_j\mathbf{v}^{(j)} - \gamma_j\mathbf{v}^{(j-1)}$ (Lanczos process)
    $\gamma_{j+1} = \|\mathbf{v}^{(j+1)}\|$
    $\alpha_0 = c_j\delta_j - c_{j-1}s_j\gamma_j$    (update QR factorization)
    $\alpha_1 = \sqrt{\alpha_0^2 + \gamma_{j+1}^2}$
    $\alpha_2 = s_j\delta_j + c_{j-1}c_j\gamma_j$
    $\alpha_3 = s_{j-1}\,\gamma_j$
    $c_{j+1} = \alpha_0/\alpha_1;\ s_{j+1} = \gamma_{j+1}/\alpha_1$    (Givens rotation)
    $\mathbf{w}^{(j+1)} = (\mathbf{v}^{(j)} - \alpha_3\mathbf{w}^{(j-1)} - \alpha_2\mathbf{w}^{(j)})/\alpha_1$
    $\mathbf{u}^{(j)} = \mathbf{u}^{(j-1)} + c_{j+1}\eta\mathbf{w}^{(j+1)}$
    $\eta = -s_{j+1}\eta$
    <Test for convergence>
enddo
```

GMRES

The Generalized Minimal Residual Method has been developed by Y. Saad and M.H. Schultz in 1986 as an attempt to find a good iterative method for general matrices. We take $\mathcal{L}_k = \mathrm{A}\mathcal{K}_k$, thus at each iteration GMRES minimizes $\|\boldsymbol{r}_k\|$ over $\mathcal{K}_k(\mathrm{A}; \boldsymbol{r}_0)$. To do so it has to solve at every step k

$$\min_{\boldsymbol{y}\in\mathbb{R}^k} \|\boldsymbol{r}_k\| = \min \|\|\boldsymbol{r}_0\|\boldsymbol{e}_1 - \hat{\mathrm{H}}_k \boldsymbol{y}_k\|$$

Since H_k is an Hessemberg matrix, its (thin) $\mathrm{Q}_k\mathrm{R}_k$ is computed by Givens rotation to give and we have to solve

$$\mathrm{R}_k \boldsymbol{y}_k = \mathrm{V}_k^T \boldsymbol{b}, \quad \boldsymbol{x}_k = \mathrm{V}_k \boldsymbol{y}_k$$

easily computable by back-substitution.

We will illustrate the not preconditioned version. With simple modifications one has the preconditioned version.

Algorithm 4.1: The Gmres Method

```
Choose $\mathbf{u}^{(0)}$, compute $\mathbf{r}^{(0)} = \mathbf{f} - F\mathbf{u}^{(0)}$, $\beta_0 = \|\mathbf{r}^{(0)}\|$, $\mathbf{v}^{(1)} = \mathbf{r}^{(0)}/\beta_0$
for $k = 1, 2, \ldots$ until $\beta_k < \tau\beta_0$ do
    $\mathbf{w}_0^{(k+1)} = F\mathbf{v}^{(k)}$
    for $l = 1$ to $k$ do
        $h_{lk} = \langle \mathbf{w}_l^{(k+1)}, \mathbf{v}^{(l)} \rangle$
        $\mathbf{w}_{l+1}^{(k+1)} = \mathbf{w}_l^{(k+1)} - h_{lk}\mathbf{v}^{(l)}$
    enddo
    $h_{k+1,k} = \|\mathbf{w}_{k+1}^{(k+1)}\|$
    $\mathbf{v}^{(k+1)} = \mathbf{w}_{k+1}^{(k+1)}/h_{k+1,k}$
    Compute $\mathbf{y}^{(k)}$ such that $\beta_k = \left\|\beta_0\mathbf{e}_1 - \widehat{H}_k\mathbf{y}^{(k)}\right\|$ is minimized,
        where $\widehat{H}_k = [h_{ij}]_{1\le i\le k+1, 1\le j\le k}$
enddo
$\mathbf{u}^{(k)} = \mathbf{u}^{(0)} + V_k\mathbf{y}^{(k)}$
```

Convergence of GMRES and GMRES(m)

Let $\widehat{\mathrm{A}}$ be diagonalizable, i.e. we may write $\widehat{\mathrm{A}} = \mathrm{X}^{-1}\mathrm{D}\mathrm{X}$, with $\mathrm{D} = \mathrm{diag}(\lambda_1, \ldots, \lambda_{\mathrm{N}})$. We define $e_k = \min_{\substack{\mathcal{W}\in\mathbb{P}^k \\ \mathcal{W}(0)=1}} \max_{k=1,\ldots,N} |\mathcal{W}(\lambda_k)|$

and we have that

$$\|\widetilde{\boldsymbol{r}}_k\| \leq e_k K(\mathrm{X})\|\widetilde{\boldsymbol{r}}_0\|.$$

Therefore, if $\widehat{\mathrm{A}}$ is diagonalizable and has l distinct eigenvectors, in exact arithmetic GMRES converges in l iterations. Moreover, if $h_{k+1,k} = 0$ the algorithm breaks down, but it means that we have reached the exact solution!.

For a theory of convergence of GMRES see V. Simoncini et al., 1996.

GMRES with restart and FGMRES

The major disadvantage of GMRES is memory. You need to store the vectors $\boldsymbol{v}^{(k)}$, which form the basis of $\mathcal{K}_k$, so memory requirement increases at each iteration.

A technique is to restart the iterations every l. This method is called GMRES(l). The convergence property is not clear, but if l is chosen sufficiently large and you use a good preconditioner it is often a good technique.

Sometimes you need to change the preconditioner at every iteration (dynamic preconditioning). In this case you need to use Flexible GMRES, we omit here to give details. FGMRES needs approx. as double the memory as GMRES. If the preconditioner does not change at each iteration the two schemes give the same result.

A simple library

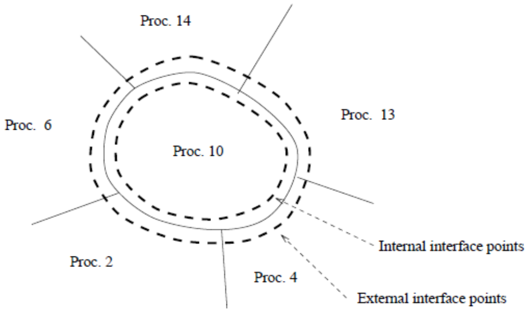
If you want to have a look to the main iterative schemes with codes written “straight out of the book” you may have a look at the Examples in LinearAlgebra/IML_Eigen/include modified from the IML++ library to be compatible with the Eigen library.
In particular in testing/test_all.cpp you have a wrapper for all iterative solvers of the suite, plus the interface with SparseLU and UMFPACK for direct methods.
You need to have al look also at LinearAlgebra/Utilities/MM_readers.hpp for tools to read matrices in Matrix Market format.
In LinearAlgebra/MatrixData you have a set of matrices to try.

Parallelization issues

Iterative methods can be implemented in parallel architecture more easily than direct method (even if parallel version of the multifrontal method are available).
The main ingredient are

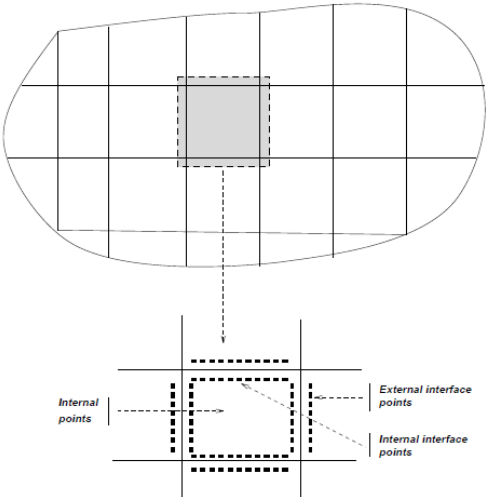
- Preconditioner setup
- Matrix-by vector operation
- Vector updates
- Dot products
- Preconditioning operations

Matrix assembly



Mesh is distributed among processors. The interface points are associated to dofs shared among processors and are used for interprocessor communication.
A strategy is that each processor assemble the rows of the matrix corresponding to the internal dofs. The interface nodes are needed to complete the local assembly.

Layout of dofs



Parallel matrix and vector operation

Since each processor stores the rows corresponding to the internal dofs and the value of the internal+interface dofs (communication needed!) every processor may compute its contribution to $\mathrm{A}\boldsymbol{x} = \boldsymbol{b}$.

The scalar product of a vector is also easily parallelizable: each processor computes its own part and then communicates by a broadcast its contribution to the other processors.

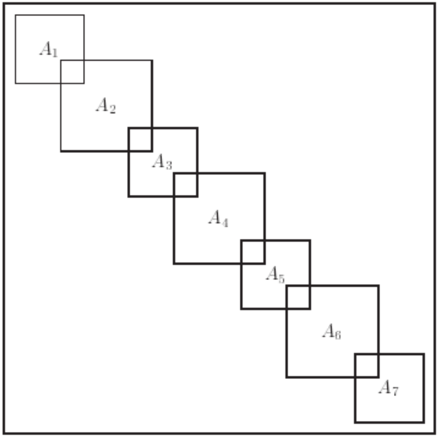
It remains to understand which preconditioners are suitable for parallel computations.

Block Jacobi preconditioners

Let $S_i = \{l_{i,1}, \ldots, l_{i,N_i}\}$ be the set containing the numbering of the internal and external interface dofs of processor i, for $i = 1, \ldots, p$. Let $\mathrm{V_i} = [\boldsymbol{e}_{\mathrm{l_{i,1}}}, \ldots, \boldsymbol{e}_{\mathrm{l_{i,N_i}}}]$, where $\boldsymbol{e}_i \in \mathbb{R}^N$ and

$$[\boldsymbol{e}_i]_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

The matrix $\mathrm{A_i} = \mathrm{V_i^T A V_i}$ is a $N_i \times N_i$ matrix. A block Jacobi preconditioner is of the form $\mathrm{M} = \sum_{k=1}^{p} \mathrm{V_i A_i^{-1} V_i^T}$.



A pictorial view of the block Jacobi matrix

Incomplete LU Block Jacobi preconditioners

The inversion of A_i may be still costly so we can replace A_i by its incomplete LU (or Cholesky) factorization $\hat{\mathrm{A}}_i = \hat{L}_i\hat{U}_i$, whose inversion is simple: $\mathrm{M} = \sum_{k=1}^{p} \mathrm{V_i}\hat{\mathrm{A}}_\mathrm{i}^{-1}\mathrm{V_i^T}$.

To implement the preconditioner in practice we compute $\hat{L}_i$ and $\hat{U}_i$ on each processor and then (if we use a simple Richardson iteration) at each iteration k

- Restrict the residual on each processor $\boldsymbol{r}_{k,i} = \mathrm{V_i^T}\boldsymbol{r}_\mathrm{k}$
- Solve $\hat{L}_i\hat{U}_i\boldsymbol{z}_{k,i} = \boldsymbol{r}_i$
- Sum the contribution (communication required) $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \mathrm{V_i}\boldsymbol{z}_\mathrm{k,i}$

Note: The matrices $\mathrm{V_i}$ are never build explicitly!

Parallel polynomial preconditioners

Polynomial preconditioners involve only multiplications of vectors with the original matrix A, so they can be effectively implemented in parallel.

Multigrid

Multigrid technique are based on a property of several iterative schemes (like Gauss-Siedel), valid for symmetric positive definite matrices arising from the discretization of elliptic problems.
If we decompose the residual $\boldsymbol{r}_0$ into discrete Fourier components one may note that high frequency modes are damped faster than low frequency modes.
From this observation the idea of having a series of grids of different level of coarsening and transfer the residual among them. In this way the coarser grids will damp the lower frequency component.

Classic and algebraic multigrid

Multigrids methods subdivides into two main categories

- Grid based multigrids. A series of nested grids $\{\tau_1, \tau_2, \ldots, \tau_m\}$ (from finer to coarser) are built and the differential operator at hand discretized on each of them using the method of choice. Consequently, a series of matrices $\{\mathrm{A} = \mathrm{A}_1, \ldots, \mathrm{A}_m\}$ and corresponding right hand side $\{\boldsymbol{b} = \boldsymbol{b}_1, \ldots, \boldsymbol{b}_m\}$ are produced.
- Algebraic multigrid (AMG). Here, the matrices $\mathrm{A}_1, \ldots, \mathrm{A}_m$ are built from the original matrix A using algebraic manipulation, without the need of building the coarse grids explicitly. Since they are more practical are currently among the most used multigrid methods, even if sometimes they are less effective than the former.

The general layout

In both cases the general layout of a multigrid method is based on the following ingredients. For each level $\nu = 1, \ldots, m$ we have

- A smoother. i.e. an iterative method with the capability of eliminating rapidly high frequency components of the error. We indicate it as $S(\mathrm{A}_\nu, \boldsymbol{b}_\nu; \boldsymbol{x}_\nu)$. It returns the result of the iterative procedure starting from the initial value $\boldsymbol{x}_\nu \in \mathbb{R}^{N_\nu}$. A_ν is the matrix at level ν
- Restriction (coarsening) operators $I_{N_\nu}^{N_{\nu+1}} : \mathbb{R}^{N_\nu} \to \mathbb{R}^{N_{\nu+1}}$ that transfers vectors from level ν to level $\nu + i$.
- Prolongation operators $I_{N_{\nu+1}}^{N_\nu} : \mathbb{R}^{N_{\nu+1}} \to \mathbb{R}^{N_\nu}$ that transfers vectors from level $\nu + 1$ to level ν. Often $I_{N_{\nu+1}}^{N_\nu} = \alpha[I_{N_\nu}^{N_{\nu+1}}]^T$ (and in AGM usually $\alpha = 1$).
- A strategy of traversing the various grids. We can have V cycles or W cycles.

How to build the A_ν

In AMG the coarse matrix and coarse right hand side is usually build by Galerkin projection from the finer level (remind that $\mathrm{A}_1 = \mathrm{A}$)

$$\mathrm{A}_{\nu+1} = I_{N_\nu}^{N_{\nu+1}} \mathrm{A}_\nu I_{N_{\nu+1}}^{N_\nu}$$

$$\boldsymbol{b}_{\nu+1} = I_{N_\nu}^{N_{\nu+1}} \boldsymbol{b}_\nu$$

Note: Usually the last level matrix A_m is small enough so that the linear system can be solved by a direct method.

Two grid cycle

To explain the general idea let's suppose to have just two levels, 1 and 2 We start from a tentative solution $\boldsymbol{x}^{(0)} = \boldsymbol{x}_1^{(0)}$ at level 1 (finer level). The algorithm reads

- Pre-smooth $\boldsymbol{x}_1^{(1)} = S(\mathrm{A}_1, \boldsymbol{b}_1; \boldsymbol{x}_1^{(0)})$
- Get residual $\boldsymbol{r}_1 = \boldsymbol{b}_1 - \mathrm{A}_1 \boldsymbol{x}_1^{(1)}$
- Coarsen $\boldsymbol{r}_2 = I_{N_1}^{N_2} \boldsymbol{r}_1$
- Solve for the correction $\mathrm{A}_2 \boldsymbol{w}_2 = \boldsymbol{r}_2$
- Correct $\boldsymbol{x}_1^{(2)} = \boldsymbol{x}_1^{(1)} + I_{N_2}^{N_1} \boldsymbol{w}_2$
- Post-smooth $\boldsymbol{x}_1^{(3)} = S(\mathrm{A}_1, \boldsymbol{b}_1; \boldsymbol{x}_1^{(2)})$
- End Return $\boldsymbol{x}_1^{(3)}$ as approximate solution.

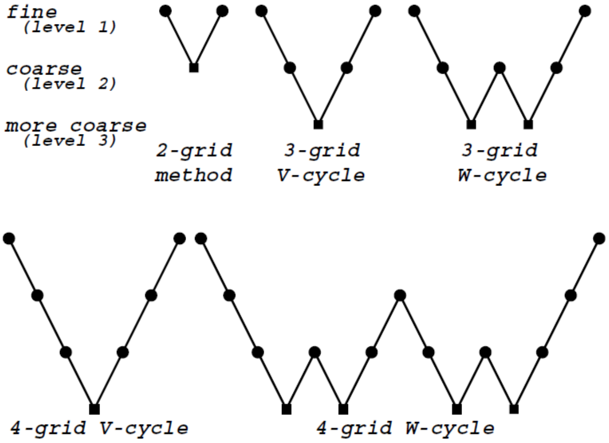
If $\boldsymbol{x}_1^{(3)}$ is not a satisfactory approximation the scheme is repeated setting $\boldsymbol{x}_1^{(0)} = \boldsymbol{x}_1^{(3)}$.

V and W cycles

If we have more levels we repeat the previous algorithm recursively. The step Solve for the correction is however replaced by a smoothing step, a part the coarsest level, where we solve (using a direct method).

The W cycle is a variation where we a V cycle from level 1 and 2 is followed by a V cycle from level 1 and 3 and so on, up to the coarsest level (many variations are in fact possible). Again the direct solution is performed only at the coarsest level.

V and W cycles



Restriction and prolongation operators

The construction of the restriction operator is clearly one of the key points of the method. in AMG setting one is the transpose of the other and a minimal requirement is that the prolongation operator be full rank.
Coarsening is often obtained by agglomeration. In practice the rows of $I_{N_\nu}^{N_{\nu+1}}$ contains either 0 or 1 (or another constant value) so that $I_{N_\nu}^{N_{\nu+1}}\boldsymbol{x}_\nu$ effectively sum-up a subset of the components of $\boldsymbol{x}_\nu$. It is called agglomeration because normally you sum up "nearby components", i.e. components i and j for which $[\mathrm{A}_{nu}]_{ij} \neq 0$.

The way to choose the agglomeration strategy may depend on the problem at hand. For the Darcy problem we mention the works of K. Stüben.

Available software

There are plenty of good software tools for direct and iterative methods, both for scalar and parallel architectures. We give a list of the main ones.

- UMFPACK. A set of routines for solving unsymmetric sparse linear systems using the Unsymmetric MultiFrontal method. Written in ANSI/ISO C. Probably the most used package of this task. Adopted also by MATLAB. I do not know of any parallel implementation yet.
- MUMPS: a MUltifrontal Massively Parallel sparse direct solver. As the name says MUMPS can be effectively run on distributed memory parallel architectures.
- LAPACK. Linear Algebra PACKage. An old but well established Fortan library with several solvers. No parallelism and no support for sparse matrices.
- ARPACK. The ARnoldi PACKage. A set of Fortran routines for large scale eigenvalue problems. Support for sparse matrices.

Available software

- SuperLU: it is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations on high performance machines. The library is written in C and is callable from either C or Fortran. Uses a modified version of LU decomposition to reduce fill-in. Scalar and parallel version (both for SIMD and MIMD architectures).
- SPARSEKIT. Sparse Matrix Utility Package. Written by Y. Saad, is a Fortran90 package that implements sparse matrix storage schemes and several iterative methods.
- Eigen. A C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. Tailored for SSEx architectures (Intel). Supports sparse and dense matrices and it contains several solvers (wrappers for LAPACK, UMFPACK, SuperLU).

Available software

- PETSC. Portable, Extensible Toolkit for Scientific Computation. A vast collection of tools for parallel linear algebra, non-linear equations, numerical solution of PDE... Originally written in Fortran 77, has now wrappers for Fortran90, C, C++.
- TRILINOS. The Trilinos Project is an effort to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems. Written in C++ has a wrapper for Fortran. It is a collection of libraries (too many to recall them) among which EPETRA/TPETRA for storing distributed matrices, BELOS for iterative solvers, ML for multilevel preconditioners, IFPACK algebraic preconditioners.... and many many others.

Domain partitioning

To partition a mesh for parallel computation there are rather well established open-source tools. For instance, the METIS-PARMETIS suite and the ZOLTAN software by Sandia National Labs.

They operate on the graph of the matrix/mesh and they allow to weight nodes in case of not equilibrated workload among nodes.

They also allow repartitioning, i.e. the dynamic transfer of nodes across processors. Very useful in the case of mesh adaption or when the workload changes in time and you need to redistribute the unknowns among processors to recalibrate it.

Bibliography

- Y. Saad. *Iterative methods for large linear systems*, 2nd Edition, SIAM, 2003.
- H. Elman, D. Silvester and A. Wathen. *Finite elements and fast iterative solvers, with applications in incompressible fluid dynamics*, Oxford Science Publications, 2005
- A Quarteroni and A Valli, *Domain decomposition methods for partial differential equations*, Oxford University Press, 1999.
- B Smith, P Bjorstad and W Gropp, *Domain decomposition*, Cambridge University Press, 1996.
- H. A. van der Vorst, *Iterative Krylov methods for large linear systems*, Cambridge University Press, 2003.
- V. Simoncini and E. Gallopoulos, *Convergence properties of block GMRES and matrix polynomials*, Linear Algebra and Applic., 1996