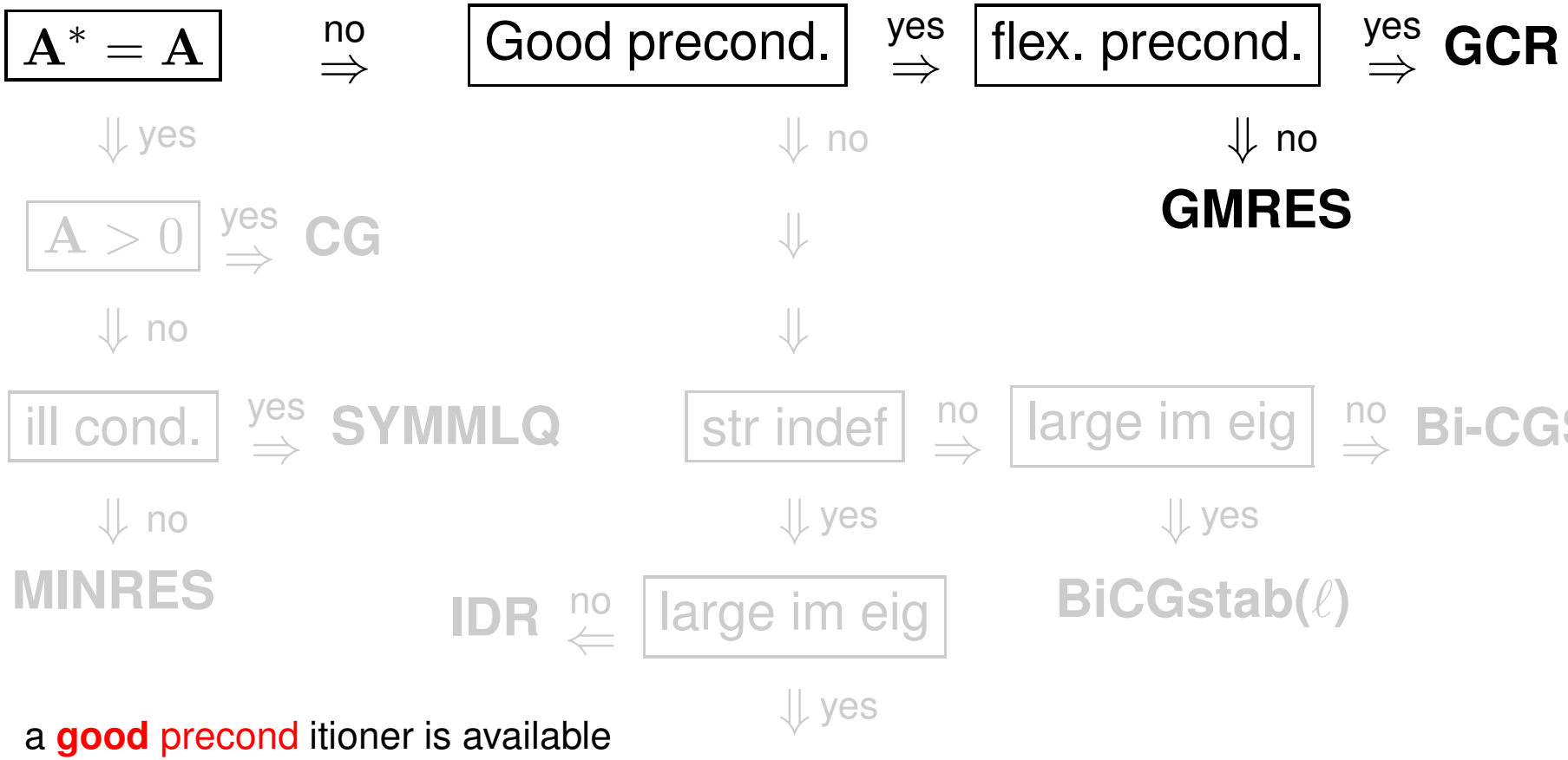
Numerical Linear Algebra

Improving iterative solvers: preconditioning, deflation, numerical software and parallelisation

Gerard Sleijpen and Martin van Gijzen

November 29, 2017

Solving $\mathrm{Ax} = \mathrm{b}$, an overview



a **good** precond itioner is available

the precond itioner is flex ible

$\mathbf{A} + \mathbf{A}^*$ is strongly indefinite

$\mathbf{A}$ has large imaginary eigenvalues

Introduction

We already saw that the performance of iterative methods can be improved by applying a preconditioner. Preconditioners (and deflation techniques) are a key to successful iterative methods. In general they are very problem dependent.

Today we will discuss some standard preconditioners and we will explain the idea behind deflation.

We will also discuss some efforts to standardise numerical software.

Finally we will discuss how to perform scientific computations on a parallel computer.

Program

- Preconditioning
  - Diagonal scaling, Gauss-Seidel, SOR and SSOR
  - Incomplete Choleski and Incomplete LU
- Deflation
- Numerical software
- Parallelisation
  - Shared memory versus distributed memory
  - Domain decomposition

Preconditioning

A preconditioned iterative solver solves the system

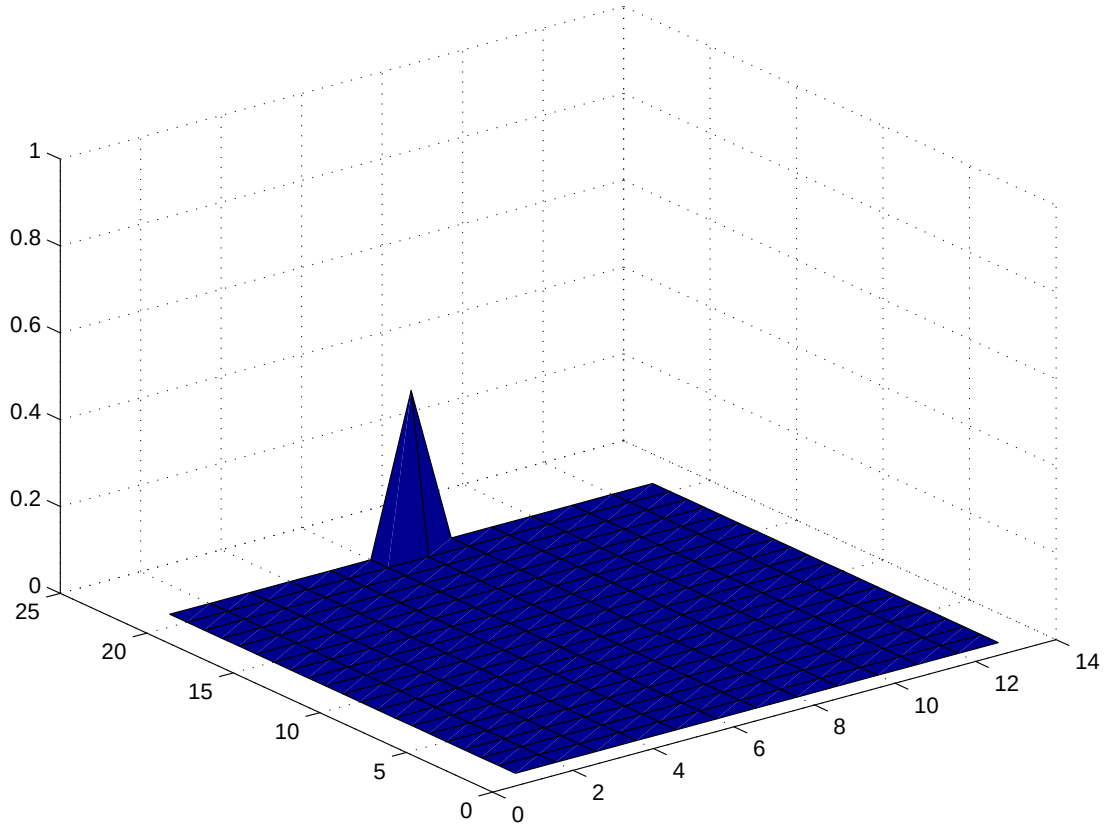
$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}.$$

The matrix $\mathbf{M}$ is called the **preconditioner**.

The preconditioner should satisfy certain requirements:

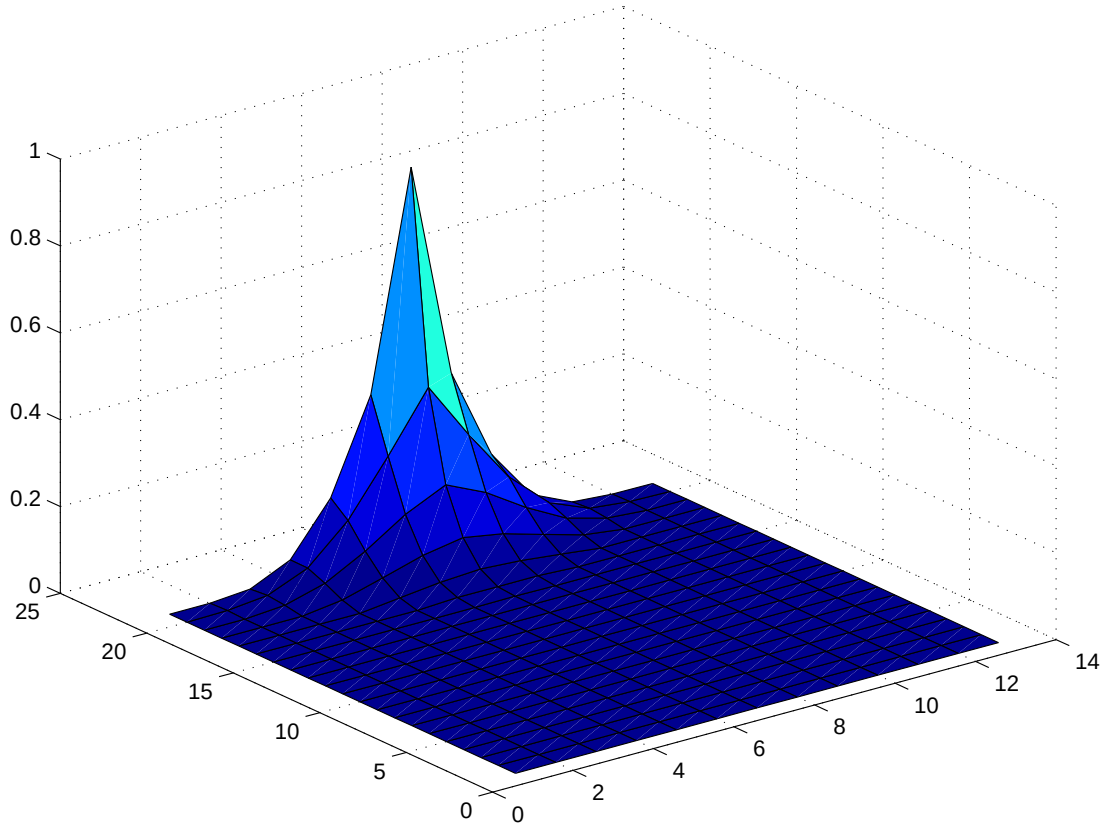
- Convergence should be (much) faster (in time) for the preconditioned system than for the original system. Normally this means that $\mathbf{M}$ is constructed as an "easily invertible" approximation to $\mathbf{A}$. Note that if $\mathbf{M} = \mathbf{A}$ any iterative method converges in one iteration.
- Operations with $\mathbf{M}^{-1}$ should be easy to perform ("cheap").
- $\mathbf{M}$ should be relatively easy to construct.

Why preconditioners?



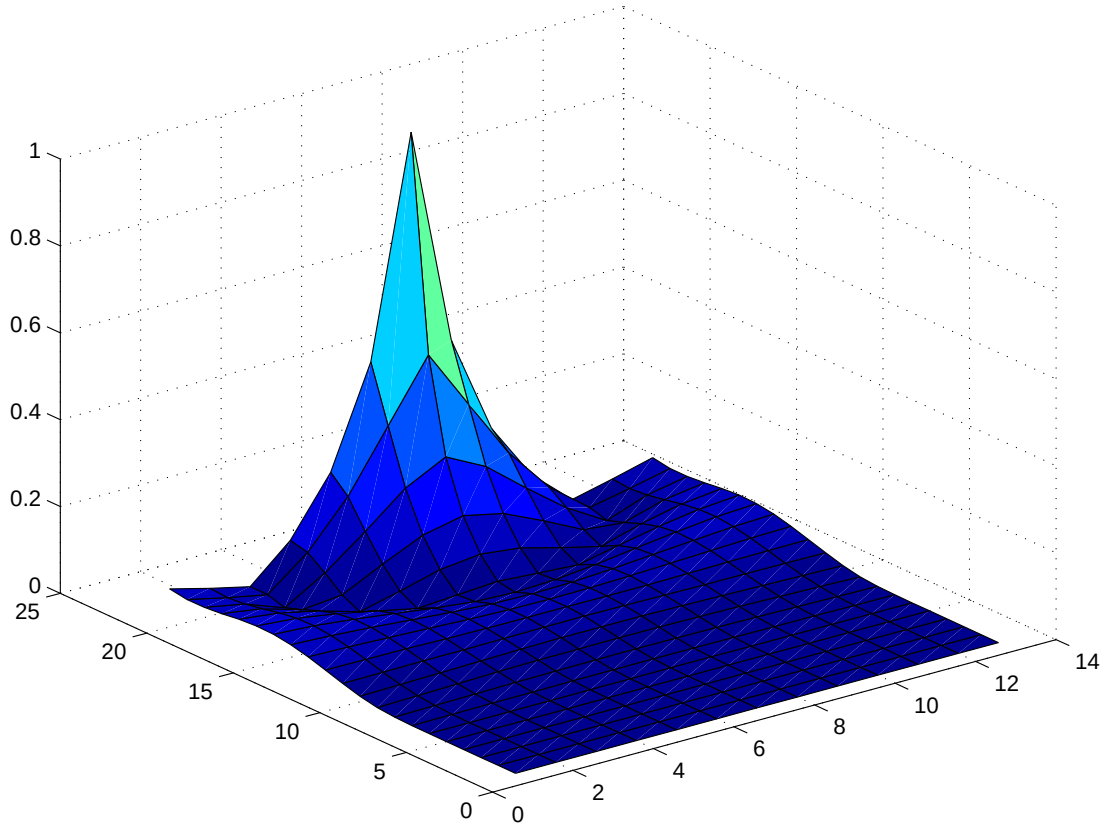
Information after 1 iteration

Why preconditioners?



Information after 7 iterations

Why preconditioners?



Information after 21 iterations

Why preconditioning? (2)

From the previous pictures it is clear that, in 2-d, we need $\mathcal{O}(1/h) = \mathcal{O}(\sqrt{n})$ iterations to move information from one end to the other end of the grid.

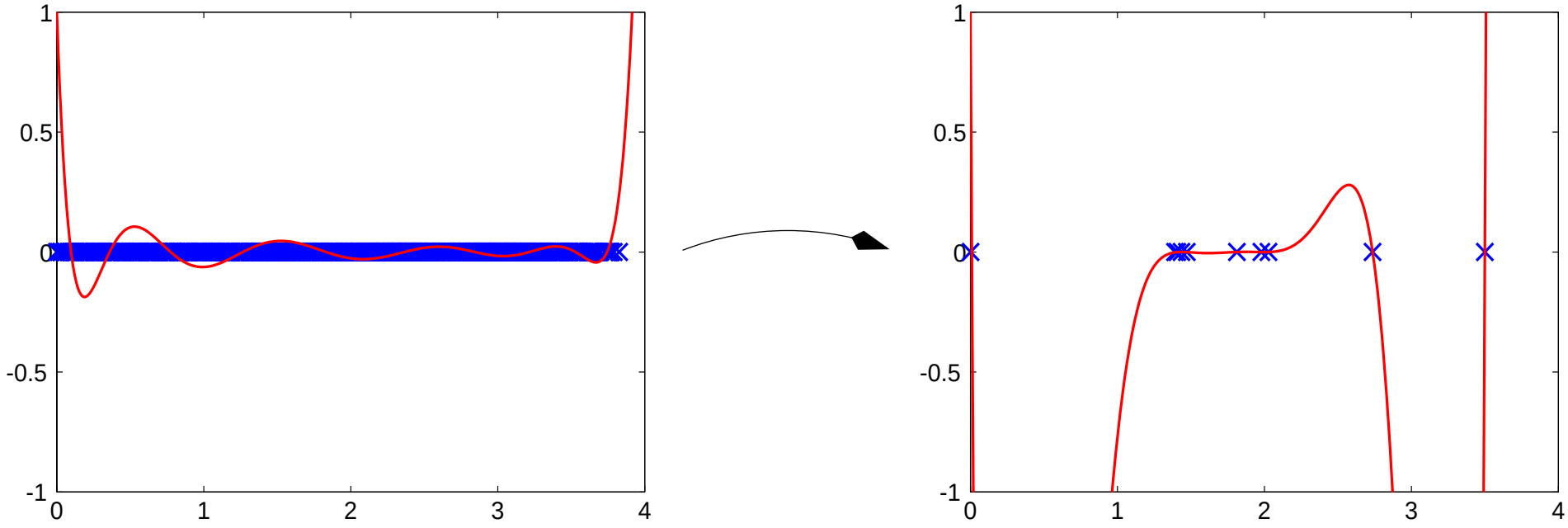
So, *at best* it takes $\mathcal{O}(n^{3/2})$ operations to compute the solution with an iterative method.

In order to improve this we need a preconditioner that enables fast propagation of information through the mesh.

Clustering the spectrum

In lecture 7 we saw that CG performs better when the spectrum of $\mathbf{A}$ is clustered.

Therefore, a good preconditioner clusters the spectrum.



Incorporating a preconditioner

Normally the matrix $\mathbf{M}^{-1}\mathbf{A}$ is not explicitly formed.

The multiplication

$$\mathbf{u} = \mathbf{M}^{-1}\mathbf{A}\mathbf{v}$$

can simply be carried out by the two operations

$\mathbf{t} = \mathbf{A}\mathbf{v}$ %% *MatVec*

solve $\mathbf{M}\mathbf{u} = \mathbf{t}$ for $\mathbf{u}$ %% *MSolve*.

Incorporating a preconditioner (2)

Preconditioners can be applied in different ways:

- from the **left**

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

- **central**ly

$$\mathbf{M} = \mathbf{L}\mathbf{U}; \qquad \mathbf{L}^{-1}\mathbf{A}\mathbf{U}^{-1}\mathbf{y} = \mathbf{L}^{-1}\mathbf{b}; \qquad \mathbf{x} = \mathbf{U}^{-1}\mathbf{y},$$

- or from the **right**

$$\mathbf{A}\mathbf{M}^{-1}\mathbf{y} = \mathbf{b}; \qquad \mathbf{x} = \mathbf{M}^{-1}\mathbf{y}.$$

- Some methods allow **implicit** preconditioning.

Left, central and right preconditioning is **explicit**.

Incorporating a preconditioner (3)

All forms of preconditioning lead to the same spectrum.

Yet there are differences:

- Left preconditioning is most natural: no extra step is required to compute $\mathbf{x}$;
- Central preconditioning allows to preserve symmetry (if $\mathbf{U} = \mathbf{L}^*$);
- Right preconditioning does not affect the residual norm.
- Implicit preconditioning does not affect the residual norm, no extra steps are required to compute $\mathbf{x}$. Unfortunately, not all methods allow implicit preconditioning. If possible, adaption of the code of the iterative solver is required.
- Explicit preconitioning allows optimisation of the combination of MSolve and MatVec as the Eisentat trick.

CG with central preconditioning

$\mathbf{x} = \mathbf{U}\mathbf{x}_0,\ \mathbf{r} = \mathbf{L}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}),\ \mathbf{u} = \mathbf{0},\quad \rho = 1$ %% *Initialization*

while $\|\mathbf{r}\| >$ tol do

$\sigma = -\rho,\ \rho = \mathbf{r}^*\mathbf{r},\ \beta = \rho/\sigma$

$\mathbf{u} \leftarrow \mathbf{r} - \beta\,\mathbf{u},\ \mathbf{c} = (\mathbf{L}^{-1}\mathbf{A}\mathbf{U}^{-1})\mathbf{u}$

$\sigma = \mathbf{u}^*\mathbf{c},\ \alpha = \rho/\sigma$

$\mathbf{r} \leftarrow \mathbf{r} - \alpha\,\mathbf{c}$ %% *update residual*

$\mathbf{x} \leftarrow \mathbf{x} + \alpha\,\mathbf{u}$ %% *update iterate*

end while

$\mathbf{x} \leftarrow \mathbf{U}^{-1}\mathbf{x}$

Here, $\mathbf{M} = \mathbf{L}\mathbf{U}$ with $\mathbf{U} = \mathbf{L}^*$. Note that $\mathbf{M}$ is positive definite.
If $\mathbf{M} = (\widetilde{\mathbf{L}} + \mathbf{D})\mathbf{D}^{-1}(\widetilde{\mathbf{L}} + \mathbf{D})^*$, take $\mathbf{L} \equiv (\widetilde{\mathbf{L}} + \mathbf{D})\mathbf{D}^{-\frac{1}{2}}$.

CG with implicit preconditioning

$\mathbf{x} = \mathbf{x}_0,\ \mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x},\ \mathbf{u} = \mathbf{0},\ \rho = 1$ %% *Initialization*

while $\|\mathbf{r}\| >$ tol do

$\mathbf{c} = \mathbf{M}^{-1}\mathbf{r}$

$\sigma = -\rho,\ \rho = \mathbf{c}^*\mathbf{r},\ \beta = \rho/\sigma$

$\mathbf{u} \leftarrow \mathbf{c} - \beta\mathbf{u},\ \mathbf{c} = \mathbf{A}\mathbf{u}$

$\sigma = \mathbf{u}^*\mathbf{c},\ \alpha = \rho/\sigma$

$\mathbf{r} \leftarrow \mathbf{r} - \alpha\,\mathbf{c}$ %% *update residual*

$\mathbf{x} \leftarrow \mathbf{x} + \alpha\,\mathbf{u}$ %% *update iterate*

end while

$\mathbf{M}$ should be Hermitian. Matlab takes $\mathbf{M}_1, \mathbf{M}_2$ (i.e., $\mathbf{M} = \mathbf{M}_1\mathbf{M}_2$).

For instance, $\mathbf{M}_1 = \widetilde{\mathbf{L}}\mathbf{D}^{-1} + \mathbf{I}$ and $\mathbf{M}_2 = (\widetilde{\mathbf{L}} + \mathbf{D})^*$.

Incorporating a preconditioner (4)

Explicit preconditioning requires

- **preprocessing** (to solve $\mathbf{M}\mathbf{b}' = \mathbf{b}$ for $\mathbf{b}'$ in left preconditioning, and $\mathbf{x}_0' = \mathbf{M}\mathbf{x}_0$ in right preconditioning),
- **adjust MatVec**: adjustment of the code for the operation that does the matrix-vector multiplication (to produce, say, $\mathbf{c} = \mathbf{M}^{-1}\mathbf{A}\mathbf{u}$),
- **postprocessing** (to solve $\mathbf{M}\mathbf{x} = \mathbf{x}'$ for $\mathbf{x}$ in right precond.).

Implicit preconditioning requires

- adaption of the code of the iterative solver.

Matlab's `PCG` uses implicit preconditioning.

Preconditioning in MATLAB (1)

Example. $\mathbf{L} \equiv \widetilde{\mathbf{L}} + \mathbf{D}$, where $\mathbf{A} = \widetilde{\mathbf{L}} + \mathbf{D} + \widetilde{\mathbf{L}}^*$ is p.d..

$$\mathbf{Ax} = \mathbf{b}, \quad \mathbf{M} \equiv \mathbf{LD}^{-1}\mathbf{L}^* \quad \rightsquigarrow \quad \mathbf{D}^{\frac{1}{2}}\mathbf{L}^{-1}\mathbf{AL}^{-*}\mathbf{D}^{\frac{1}{2}}\mathbf{y} = \mathbf{D}^{\frac{1}{2}}\mathbf{L}^{-1}\mathbf{b}$$

```
function [PreMV,pre_b,PostProcess,pre_x0] = preprocess(A,b,x0)
  L = tril(A);  D = diag(sqrt(diag(A)));  L = L/D;  U = L';
  function c = MyPreMV(u);
    c = L\(A*(U\u));
  end
  PreMV = @MyPreMV;   pre_b = L\b;   pre_x0 = U*x0;
  function x = MyPostProcess(y)
    x = U\y;
  end
  PostProcess = @MyPostProcess;
end
```

Here, `A` is a sparse matrix. Note that `L, U, D` are also sparse

Preconditioning in MATLAB (2)

Example. $\mathbf{L} \equiv \widetilde{\mathbf{L}} + \mathbf{D}$, where $\mathbf{A} = \widetilde{\mathbf{L}} + \mathbf{D} + \widetilde{\mathbf{L}}^*$ is p.d..

$$\mathbf{Ax} = \mathbf{b}, \ \mathbf{M} \equiv \mathbf{LD}^{-1}\mathbf{L}^* \quad \rightsquigarrow \quad \mathbf{D}^{\frac{1}{2}}\mathbf{L}^{-1}\mathbf{AL}^{-*}\mathbf{D}^{\frac{1}{2}}\mathbf{y} = \mathbf{D}^{\frac{1}{2}}\mathbf{L}^{-1}\mathbf{b}$$

```
function x = CG(A,b,tol,kmax,x0)
  ...
    c=A*u;
  ...
end
```

In Matlab's command window, replace

```
> x = CG(A,b,100,10^(-8),x0);
```

by

```
> [pMV,pb,postP,px0] = preprocess(A,b,x0);
> y = CG(pMV,pb,100,10^(-8),px0);
> x = postP(y);
```

Preconditioning in MATLAB (3)

Example. $\mathbf{L} \equiv \widetilde{\mathbf{L}} + \mathbf{D}$, where $\mathbf{A} = \widetilde{\mathbf{L}} + \mathbf{D} + \widetilde{\mathbf{L}}^*$ is p.d..

$$\mathbf{Ax} = \mathbf{b}, \quad \mathbf{M} \equiv \mathbf{L}\mathbf{D}^{-1}\mathbf{L}^* \quad \rightsquigarrow \quad \mathbf{D}^{\frac{1}{2}}\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-*}\mathbf{D}^{\frac{1}{2}}\mathbf{y} = \mathbf{D}^{\frac{1}{2}}\mathbf{L}^{-1}\mathbf{b}$$

The CG code should take both functions and matrices:

```
function x = CG(A,b,tol,kmax,x0)
  A_is_matrix = isnumeric(A);
  function c=MV(u)
    if A_is_matrix, c=A*u; else, c=A(u); end
  end

  ...
    c=MV(u);
  ...

end
```

Preconditioning in MATLAB (4)

Example. $\mathbf{L} \equiv \widetilde{\mathbf{L}} + \mathbf{D}$, where $\mathbf{A} = \widetilde{\mathbf{L}} + \mathbf{D} + \widetilde{\mathbf{L}}^*$ is p.d..

$\mathbf{Ax} = \mathbf{b}$, $\mathbf{M} \equiv \mathbf{LD}^{-1}\mathbf{L}^*$ $\rightsquigarrow$ implicit preconditioning

```
function MSolve = SetPrecond(A)
   L = tril(A);   U = L';   D = diag(diag(A));
   function c = MyPreMV(u);
      c = U\(D*(L\u));
   end
   MSolve = @MyMSolve;
end
function x = PCG(A,b,tol,kmax,MSolve,x0)
    ...
    c=MSolve(r);
    ...
    c=A(u);
    ...
end
```

Diagonal scaling

Diagonal scaling or **Jacobi preconditioning** uses

$$\mathbf{M} = \mathit{diag}(\mathbf{A})$$

as preconditioner. Clearly, this preconditioner does not enable fast propagation through a grid. On the other hand, operations with $\mathit{diag}(\mathbf{A})$ are very easy to perform and diagonal scaling can be useful as a first step, in combination with other techniques.

If the choice for $\mathbf{M}$ is restricted to diagonals,
then $\mathbf{M} = \mathit{diag}(\mathbf{A})$ minimize $\mathcal{C}(\mathbf{M}^{-1}\mathbf{A})$ is some sense.

Gauss-Seidel, SOR and SSOR

The **Gauss-Seidel preconditioner** is defined by

$$\mathbf{M} = \mathbf{L} + \mathbf{D}$$

with $\mathbf{L}$ the strictly lower-triangular part of $\mathbf{A}$ and $\mathbf{D} = \mathit{diag}(\mathbf{A})$. By introducing a **relaxation parameter** ω (using $\mathbf{D}/\omega$ instead of $\mathbf{D}$), we get the **SOR-preconditioner**.

For symmetric problems it is wise to take a symmetric preconditioner. A symmetric variant of Gauss-Seidel is

$$\mathbf{M} = (\mathbf{L} + \mathbf{D})\mathbf{D}^{-1}(\mathbf{L} + \mathbf{D})^*$$

By introducing a relaxation parameter ω (i.e., by using $\mathbf{D}/\omega$ instead of $\mathbf{D}$) we get the so called **SSOR-preconditioner**.

ILU-preconditioners

ILU-preconditioners are the most popular 'black box' preconditioners. They are constructed by making a standard LU-decomposition

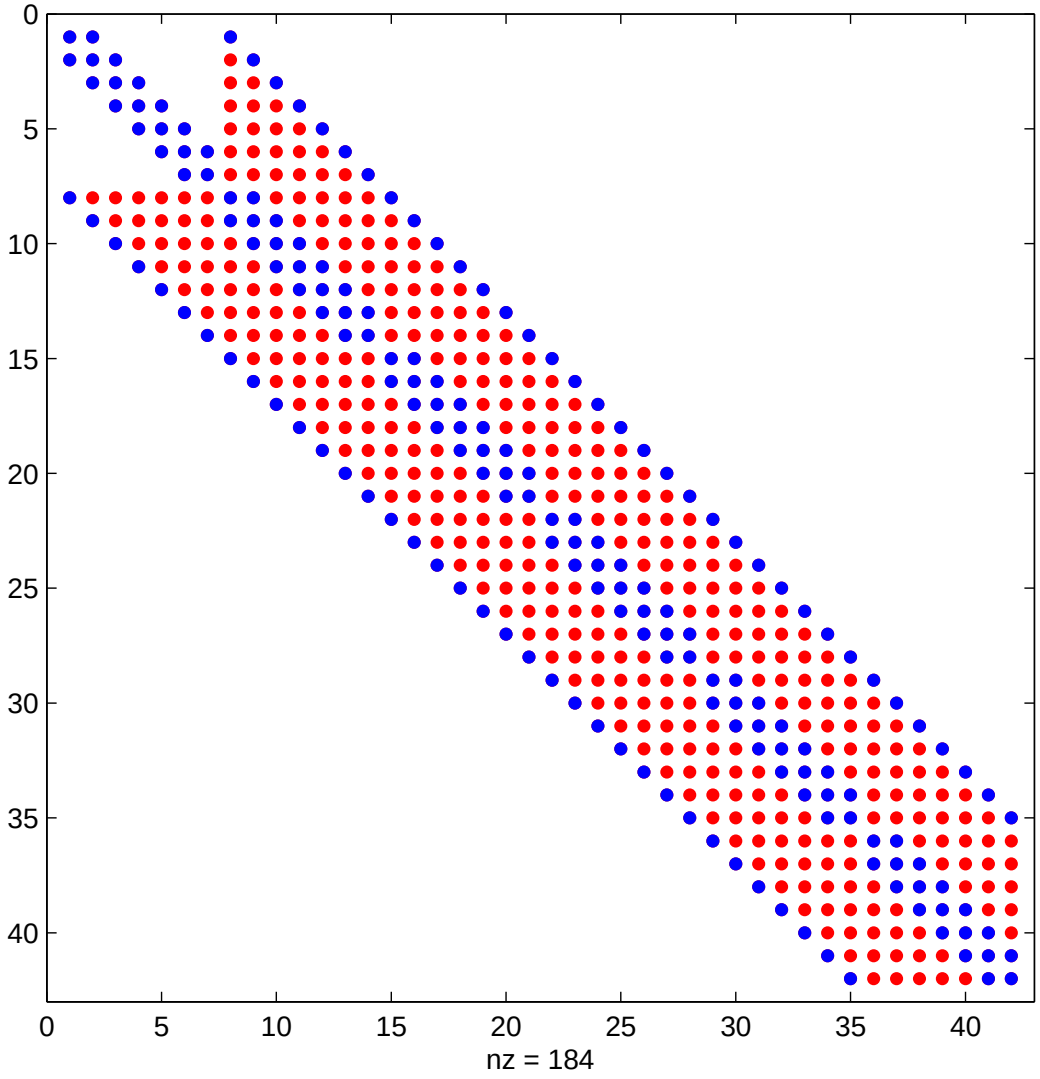
$$\mathbf{A} = \mathbf{LU}.$$

However, during the elimination process some nonzero entries in the factors are discarded (i.e., replaced by 0). This leads to $\mathbf{M} = \mathbf{LU}$ with $\mathbf{L}$ and $\mathbf{U}$ the "incomplete" L- and U-factors.

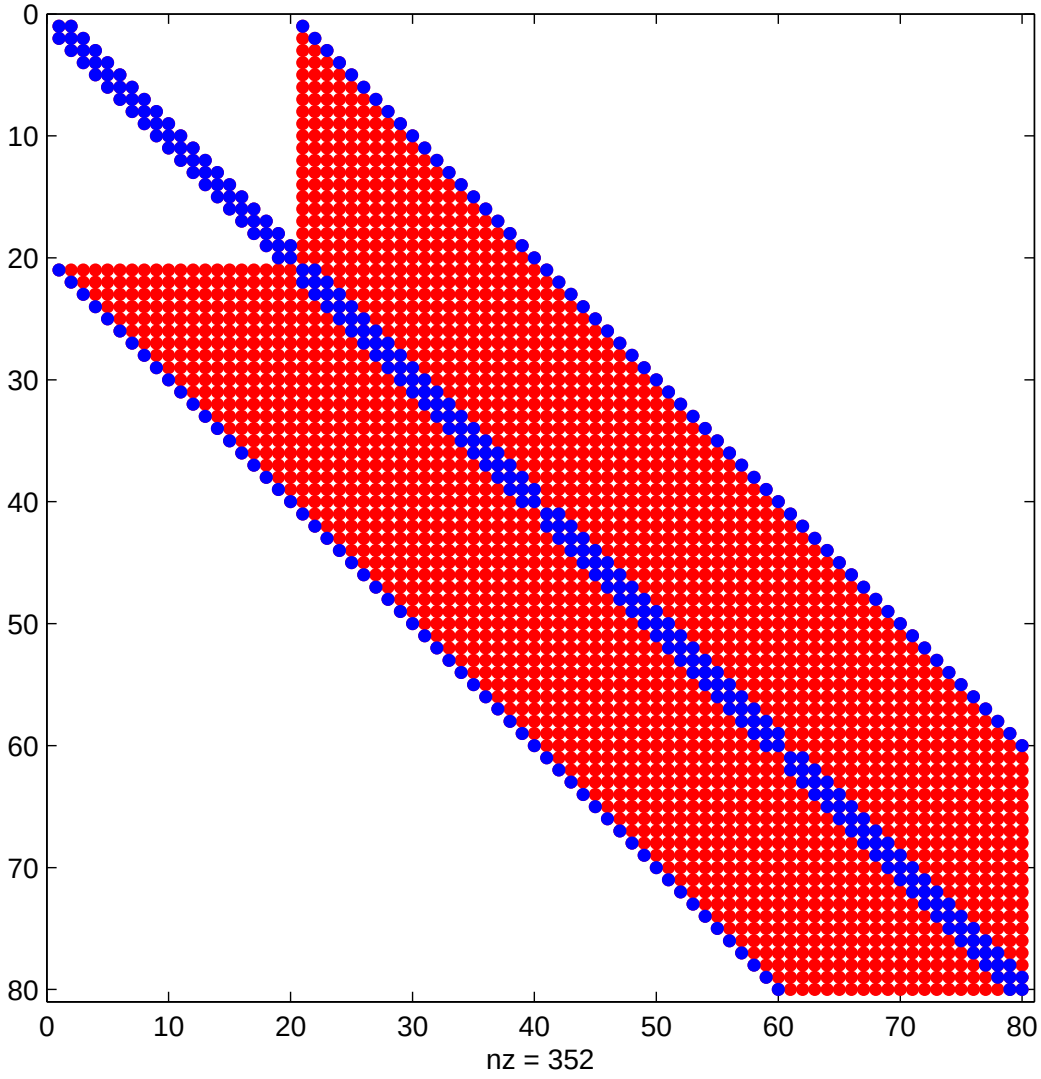
Discarding nonzero entries can be done on basis of two criteria:

- Sparsity pattern: e.g., an entry in a factor is only kept if it corresponds to a nonzero entry in $\mathbf{A}$;
- Size: small entries in the decomposition are dropped.

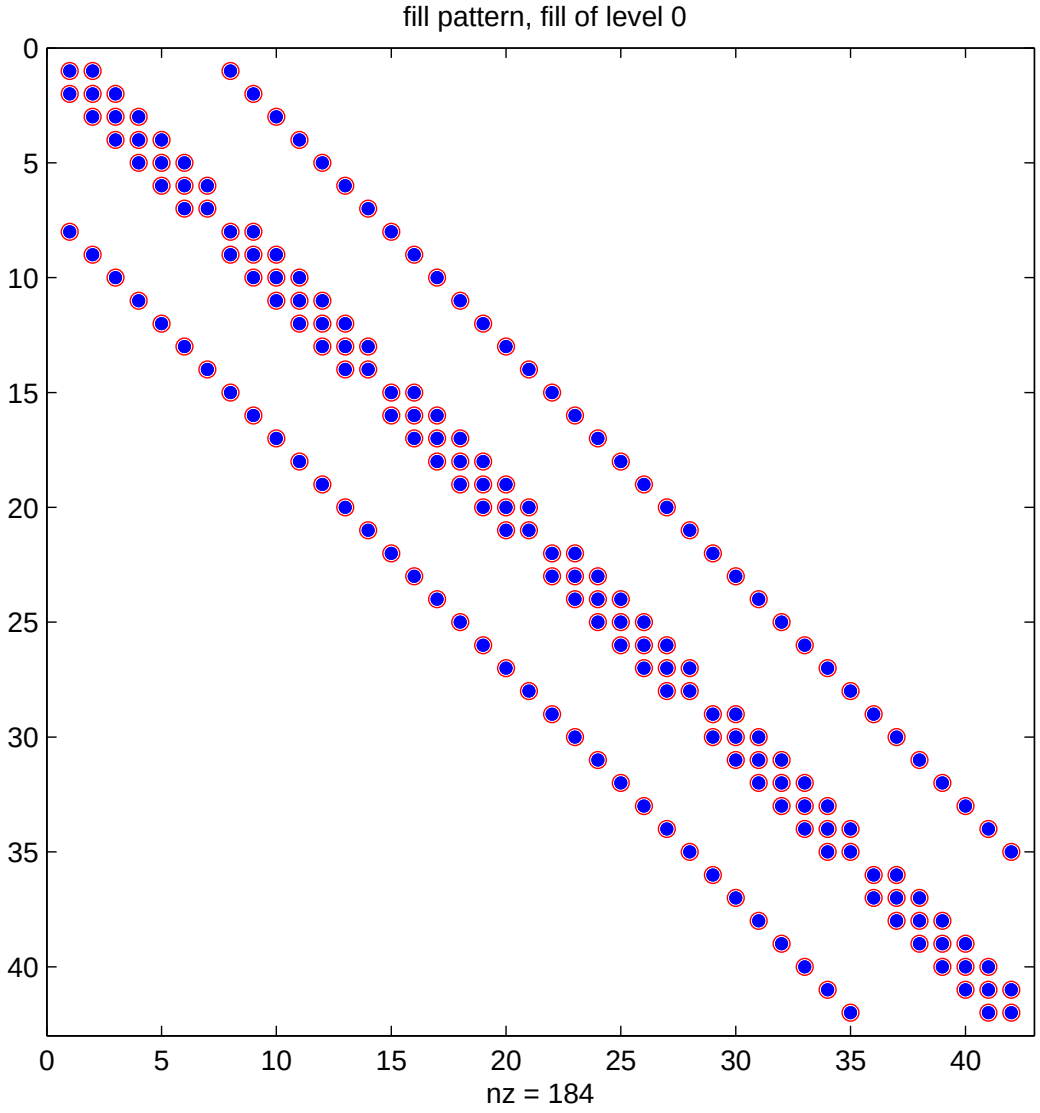
Sparsity patterns, fill



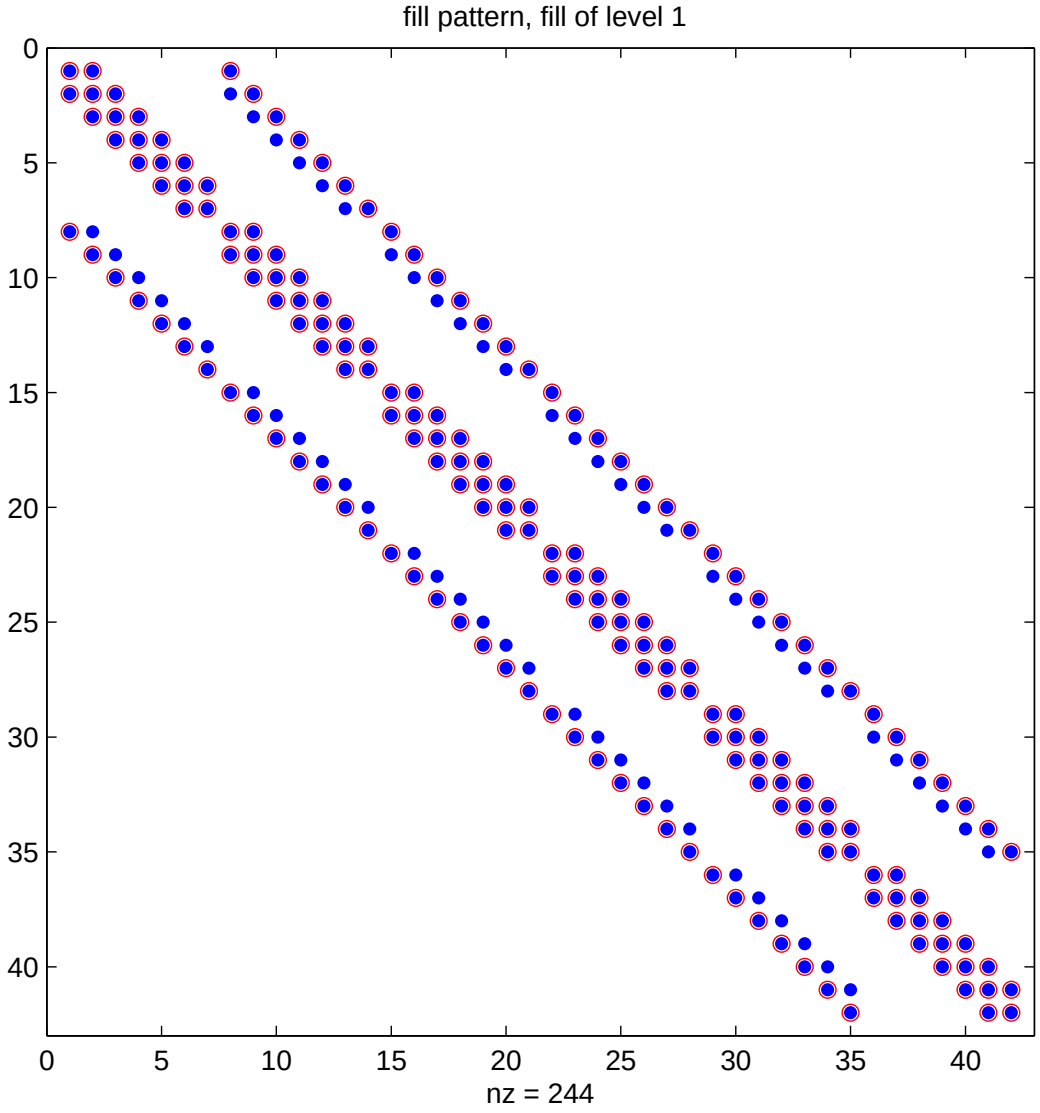
Sparsity patterns, fill



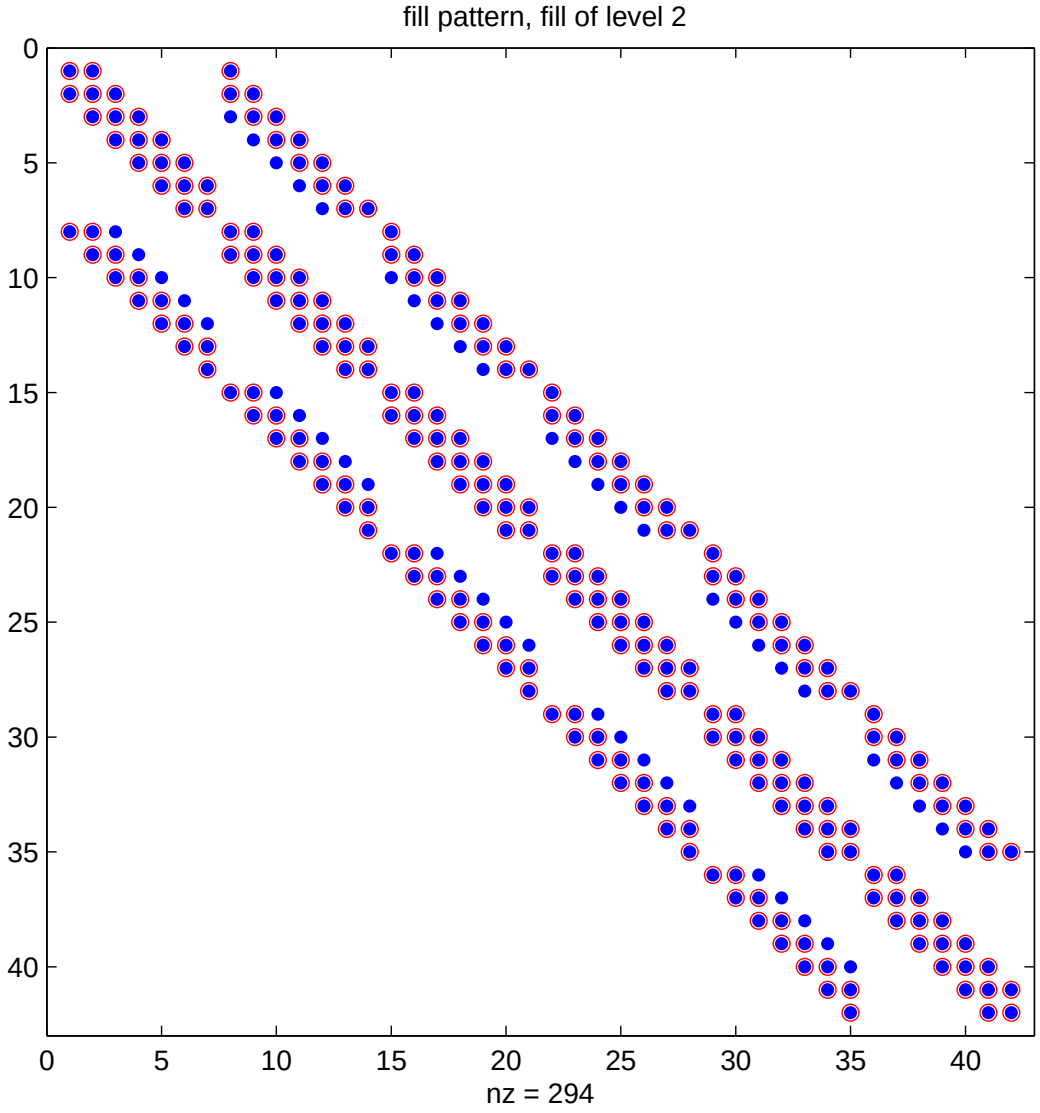
Sparsity patterns, ILU(0)



Sparsity patterns, ILU(1)



Sparsity patterns, ILU(2)



ILU-preconditioners (2)

Modified **ILU-preconditioners** are constructed by making an ILU-decomposition

$$\mathbf{M} = \mathbf{LU}.$$

However, during the elimination process the nonzero entries in the factors that are discarded in the ILU construction are now added to the diagonal entry of $\mathbf{U}$. Then $\mathbf{A1} = \mathbf{M1}$.

The idea here is that both $\mathbf{A}$ and $\mathbf{M}$ have the approximately the same effect when acting on “smooth” functions (vectors).

ILU-preconditioners (2)

The number of nonzero entries that is maintained in the LU-factors is normally of the order of the number of nonzeros in $\mathbf{A}$.
This implies that operations with the ILU-preconditioner are approximately as costly as multiplications with $\mathbf{A}$.

For $\mathbf{A}$ Symmetric Positive Definite a special variant of ILU exists, called **Incomplete Choleski**. This preconditioner is based on the Choleski decomposition $\mathbf{A} = \mathbf{C}\mathbf{C}^*$.

Deflation

There are two popular ways to improve the convergence of Krylov solvers.

- **Preconditioning**: improve distribution of the eigenvalues (cluster eigenvalues).
- **Deflation**: remove eigenvectors with eigenvalues that are small in absolute value.
  Actually, small eigenvalues are mapped to zero and approximate solution are constructed in the orthogonal complement of the kernel of the deflated matrix.

Deflation (2)

$\mathbf{U}$ is an $n \times s$ matrix (typically with s small), $\mathbf{V} \equiv \mathbf{A}\mathbf{U}$. Suppose the **interaction matrix** $\mu \equiv \mathbf{U}^*\mathbf{A}\mathbf{U} = \mathbf{U}^*\mathbf{V}$ is non-singular.

Consider the skew projections

$$\Pi_1 \equiv \mathbf{I} - \mathbf{V}\mu^{-1}\mathbf{U}^* \quad \text{and} \quad \Pi_0 \equiv \mathbf{I} - \mathbf{U}\mu^{-1}\mathbf{U}^*\mathbf{A}$$

Note. $\Pi_1\mathbf{y} \perp \mathbf{U}$ for all n-vectors $\mathbf{y}$, $\Pi_1\mathbf{A}\mathbf{U} = \mathbf{0}$, $\Pi_1\mathbf{A} = \mathbf{A}\Pi_0$

Theorem. Put $\mathbf{A}' \equiv \Pi_1\mathbf{A}$ and $\mathbf{b}' \equiv \Pi_1\mathbf{b}$.
If $\mathbf{x}'$ solves $\mathbf{A}'\mathbf{x}' = \mathbf{b}'$ for $\mathbf{x}' \perp \mathbf{U}$,
then $\mathbf{x} \equiv \Pi_0\mathbf{x}' + \mathbf{U}\mu^{-1}\mathbf{U}^*\mathbf{b}$ solves $\mathbf{A}\mathbf{x} = \mathbf{b}$.

Exercise. If $\mathbf{A}$ is Hermitian, then $\mathbf{A}'$ is Hermitian.

Deflation (3)

The **deflated** map $\mathbf{A}'$ maps $\mathbf{U}^{\perp}$ to $\mathbf{U}^{\perp}$. In particular, Krylov subspaces generated by $\mathbf{A}'$ and $\mathbf{b}'$ are orthogonal to $\mathbf{U}$ and Krylov subspace solvers find approximate solutions in $\mathbf{U}^{\perp}$.

If the space spanned by the columns of $\mathbf{U}$ (approximately) contains the eigenvectors associated with the s absolute smallest eigenvalues, then the deflated map $\mathbf{A}'$ has (approximately) the same eigenvalues as $\mathbf{A}$, except for the s absolute smallest ones, which are replaced by 0.
When $\mathbf{A}'$ is considered as a map from $\mathbf{U}^{\perp}$ to $\mathbf{U}^{\perp}$, then these 0 eigenvalues are removed (then $\mathbf{A}'$ has only $n - s$ eigenvalues).

Deflation, examples.

• Multiple right hand side systems or **block systems**.
$\mathbf{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_s]$, where $\mathbf{u}_j$ are the Ritz vectors with absolute smallest Ritz values obtained from solving $\mathbf{A}\mathbf{x} = \mathbf{b}_1$ with GMRES. Then $\mathbf{U}$ can be used to solve $\mathbf{A}\mathbf{x} = \mathbf{b}_2, \ldots$.

• **GCRO**: Local Minimal Residual iteration where the update vector $\mathbf{u}_k$ in step k is obtained by solving

$$\mathbf{A}'\mathbf{u}' = \mathbf{r}_k \quad \text{for} \quad \mathbf{u}' \perp \mathbf{U}_{k-1}$$

with ℓ steps of GMRES. In the next step $\mathbf{U}_k \equiv [\mathbf{U}_{k-1}, \mathbf{u}_k]$.

• **Multi-grid**. $\mathbf{U}$ is the matrix that represents the restriction to a coarse grid, $\mathbf{U}^*$ is the prolongation, μ^{-1} is the coarse grid correction. s is not really small in this application (typically $s = \frac{n}{4}$).

Deflation, examples (2)

• In many application (typically those from CFD), eigenvectors with absolute small eigenvalues correspond to smooth (non- or mildly oscillating) functions on the computational grid. Then the columns of $\mathbf{U}$ are taken to be vectors corresponding to smooth functions, as the function of all 1, plus functions that are 1 on part of the grid and 0 elsewhere. The part of the grid where the function is 0 can be a part where the coefficients in the PDE are (small) constant.
The idea here is that the 'hampering' eigenvectors are not known but they will be close to the space spanned by the $\mathbf{U}$-vectors.

Deflation of this type is typically applied in case of an isolated cluster of absolute small eigenvalues.

Numerical software

To promote the use of good quality software several efforts have been made in the past. We mention:

- **Eispack**: Fortran 66 package for eigenvalue computations.
- **Linpack**: F77 package for dense or banded linear systems.
- **BLAS**: standardisation of basic linear algebra operations. Available in several languages.
- **LAPACK**: F77 package for dense eigenvalue problems and linear systems. Builds on BLAS and has replaced Eispack and Linpack. F90 and C++ versions also exist.

The above software can be downloaded from http://www.netlib.org.

B asic L inear A lgebra S ubroutines

The BLAS libraries provide a standard for basic linear algebra subroutines. The BLAS library is available in optimised form on many (super)computers. Three versions are available:

- **BLAS 1**: vector-vector operations;
- **BLAS 2**: matrix-vector operations;
- **BLAS 3**: matrix-matrix operations.

BLAS 1

Examples of BLAS 1 routines are:

- Vector update: `daxpy`
- Inner product: `ddot`
- Vector scaling: `dscal`

Cache memory

Computers normally have small, fast memory close to the processor, the so called **cache** memory.

For optimal performance data in the cache should be reused as much as possible.

Level 1 BLAS routines are for this reason not very efficient: for every number that is loaded into the cache only one calculation is made.

BLAS 2 and 3

An example of a BLAS 2 routine is the matrix-vector multiplication routine `dgemv`.

An example of a BLAS 3 routine is the matrix-matrix multiplication routine `dgemm`.

BLAS 2 and in particular BLAS 3 routines make much better use of the cache than BLAS 1 routines.

Parallel computing

Modern supercomputers may contain many thousands of processors. Another popular type of parallel computer is the cluster of workstations connected via a communication network.

Parallel computing poses special restrictions on the numerical algorithms. In particular, algorithms that rely on recursions are difficult to parallelise.

Shared versus distributed memory

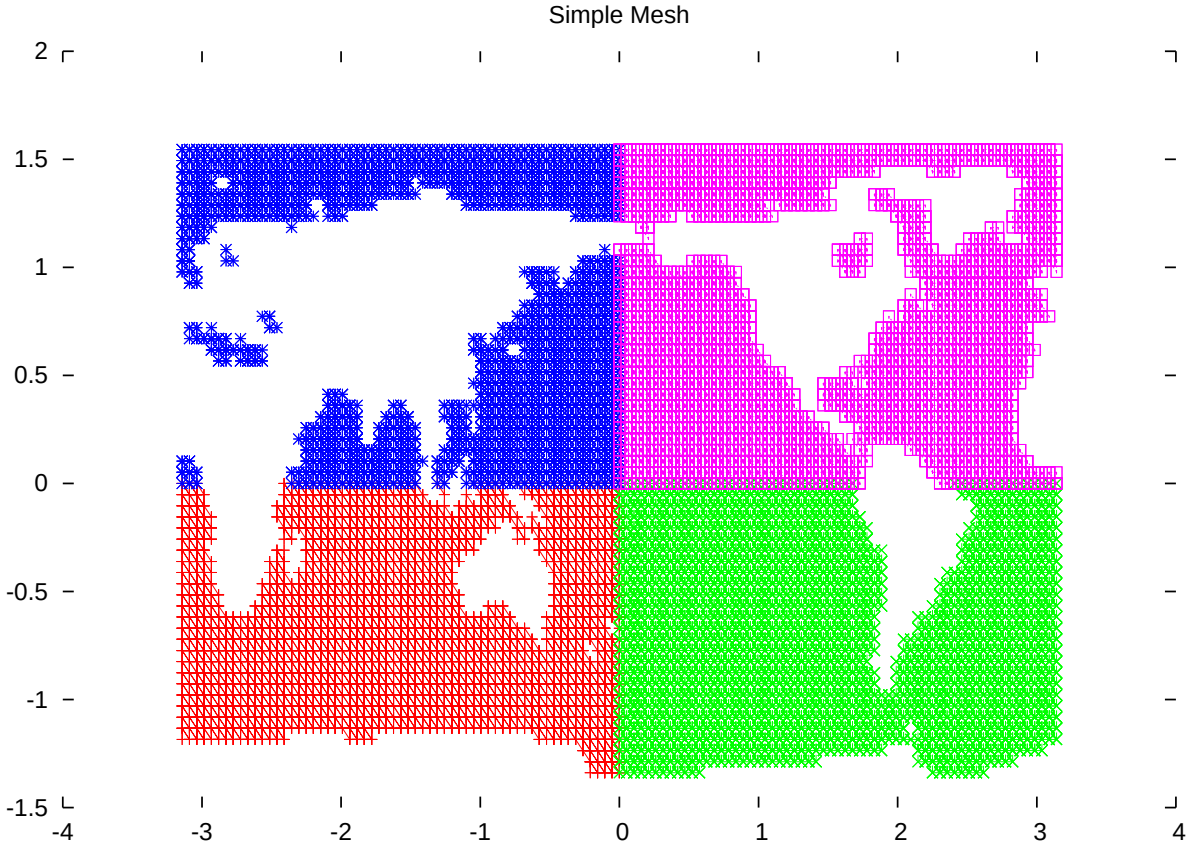
An important distinction in computer architecture is the memory organisation.

Shared memory machines have a single address space: all processors read from and write to the same memory. This type of machines can be parallelised using fine grain, loop level parallelisation techniques.

On **distributed memory** computers every processors has its own local memory. Data is exchanged via a message passing mechanism. Parallelisation should be done using a coarse grain approach. This is often achieved by making a domain decomposition.

Domain decomposition

A standard way to parallelise a grid-based computation is to split the domain into p subdomains, and to map each subdomain on a processor.



Domain decomposition (2)

The domain decomposition decomposes the system $\mathbf{Ax} = \mathbf{b}$ into blocks. For two subdomains one obtains

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix},$$

$\mathbf{x}_1$ and $\mathbf{x}_2$ represent the subdomain unknowns, $\mathbf{A}_{11}$ and $\mathbf{A}_{22}$ the subdomain discretization matrices and $\mathbf{A}_{12}$ and $\mathbf{A}_{21}$ the coupling matrices between subdomains.

Parallel matrix-vector multiplication

For the *matrix-vector multiplication* (MV, MatVec)

$$\mathbf{u} = \mathbf{A}\mathbf{v}$$

the operations

$$\mathbf{u}_1 = \mathbf{A}_{11}\mathbf{v}_1 + \mathbf{A}_{12}\mathbf{v}_2 \quad \text{and} \quad \mathbf{u}_2 = \mathbf{A}_{22}\mathbf{v}_2 + \mathbf{A}_{21}\mathbf{v}_1$$

can be performed in parallel.

However, processor 1 has to send (part of) $\mathbf{v}_1$ to processor 2 before the computation, and processor 2 (part of) $\mathbf{v}_2$ to processor 1.

This kind of communication is **local**, only informations from neighbouring subdomains is needed.

Parallel vector operations

Inner products (DOT)

$$\mathbf{u}^*\mathbf{v}$$

are calculated by computing the local inner products

$$\mathbf{u}_1^*\mathbf{v}_1 \quad \text{and} \quad \mathbf{u}_2^*\mathbf{v}_2.$$

The final result is obtained by adding all local inner products.

This requires **global** communication.

Vector updates (AXPY)

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{y}$$

can be performed locally, without any communication.

Domain decomposition preconditioners

It is a natural idea to solve a linear system

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

by solving the subdomain problems independently and to iterate to correct for the error.

This idea has given rise to the family of domain decomposition preconditioners.
The theory for domain decomposition preconditioners is vast, here we only discuss some important ideas.

Block-Jacobi preconditioner

A simple domain decomposition preconditioner is defined by

$$\mathbf{M} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{22} \end{bmatrix}$$

(**Block-Jacobi** preconditioner) or, in general, by

$$\mathbf{M} = \begin{bmatrix} \mathbf{A}_{11} & & \\ & \ddots & \\ & & \mathbf{A}_{MM} \end{bmatrix}$$

The subdomain systems can be solved in parallel.

Solution of subdomain problems

There are several ways to solve the subdomain problems:

- Exact solves. This is in general (too) expensive.
- Inexact solves using an incomplete decomposition (block-ILU).
- Inexact solves using an iterative method to solve the subproblems. Since in this case the preconditioner is variable, the outer iteration should be flexible, for example GCR.

On the scalability of block-Jacobi

Without special techniques, the number of iterations increases with the number of subdomains. The algorithm is not scalable.

To overcome this problems techniques can be applied to enable the exchange of information between subdomains.

Improving the scalability

Two popular techniques improve the flow of information are:

- Use an overlap between subdomains. Of course one has to ensure that the value of unknowns in gridpoints that belong to multiple domains is unique.
- Use a coarse grid correction. The solution of the coarse grid problem is added to the subdomain solutions.
  This idea is closely related to multigrid, since the coarse-grid solution is the non-local, smooth part of the solution that cannot be represented on a single subdomain.

Concluding remarks

Preconditioners are the key to a successful iterative method.

Today we saw some of the most important preconditioners. The 'best' preconditioner, however, depends completely on the problem.