Advanced Topics from Scientific Computing
TU Berlin Winter 2022/23
Notebook 08
 Jürgen Fuhrmann

Iterative methods for linear systems

Let $V = \mathbb{R}^n$ be equipped with the inner product $(\cdot,\cdot)$. Let A be an $n \times n$ nonsingular matrix.

Solve $Au = b$ iteratively. For this purpose, two components are needed:

- **Preconditioner**: a matrix $M \approx A$ "approximating" the matrix A but with the property that the system $Mv = f$ is easy to solve
- **Iteration scheme**: algorithmic sequence using M and A which updates the solution step by step

Simple iteration scheme

Assume we know the exact solution $\hat{u}$: $A\hat{u} = b$.

Then it must fulfill the identity

$$\hat{u} = \hat{u} - M^{-1}(A\hat{u} - b)$$

⇒ iterative scheme: put the "old" value on the right hand side and the ""new" value on the left hand side:

$$u_{k+1} = u_k - M^{-1}(Au_k - b) \quad (k = 0, 1 \ldots)$$

Obviously, if $u_k = \hat{u}$, the process would be stationary.

Otherwise it leads to a sequence of approximations

$$u_0, u_1, \ldots, u_k, u_{k+1}, \ldots$$

Implementation: solve $Au = b$ with tolerance ε:

1. Choose initial value u_0, set $k = 0$
2. Calculate *residuum* $r_k = Au_k - b$
3. Test convergence: if $||r_k|| < \varepsilon$ set $u = u_k$, finish
4. Calculate *update*: solve $Mv_k = r_k$
5. Update solution: $u_{k+1} = u_k - v_k$, set $k = k + 1$, repeat with step 2.

General convergence theorem

Let $\hat{u}$ be the solution of $Au = b$.

Let $e_k = u_k - \hat{u}$ be the error of the k-th iteration step. Then:

$$\begin{aligned} u_{k+1} &= u_k - M^{-1}(Au_k - b) \\ &= (I - M^{-1}A)u_k + M^{-1}b \\ u_{k+1} - \hat{u} &= u_k - \hat{u} - M^{-1}(Au_k - A\hat{u}) \\ &= (I - M^{-1}A)(u_k - \hat{u}) \\ &= (I - M^{-1}A)^k(u_0 - \hat{u}) \end{aligned}$$

resulting in

$$e_{k+1} = (I - M^{-1}A)^k e_0$$

- So when does $(I - M^{-1}A)^k$ converge to zero for $k \to \infty$?
- Denote $B = I - M^{-1}A$

Definition The spectral radius $\rho(B)$ is the largest absolute value of any eigenvalue of B: $\rho(B) = \max_{\lambda \in \sigma(B)} |\lambda|$.

Sufficient condition for iterative method convergence:

$$\rho(I - M^{-1}A) < 1$$

Asymptotic convergence factor ρ_{it} can be estimated via the spectral radius:

$$\begin{aligned}\rho_{it} &= \lim_{k\to\infty}\left(\max_{u_0}\frac{||(I-M^{-1}A)^k(u_0-\hat{u})||}{||u_0-\hat{u}||}\right)^{\frac{1}{k}}\\ &= \lim_{k\to\infty}||(I-M^{-1}A)^k||^{\frac{1}{k}}\\ &= \rho(I-M^{-1}A)\end{aligned}$$

Depending on u_0, the rate may be faster, though

Convergence estimate for symmetric positive definite A,M

Matrix preconditioned Richardson iteration: M, A spd.

Scaled Richardson iteration with preconditioner M

$$u_{k+1} = u_k - \alpha M^{-1}(Au_k - b)$$

Spectral equivalence estimate

$$0 < \gamma_{min}(Mu,u) \leq (Au,u) \leq \gamma_{max}(Mu,u)$$

$\Rightarrow \gamma_{min} \leq \lambda_i \leq \gamma_{max}$

$\Rightarrow$ optimal parameter $\alpha = \frac{2}{\gamma_{max}+\gamma_{min}}$

$\Rightarrow$ convergence rate with optimal parameter: $\rho_{opt} \leq \frac{\kappa(M^{-1}A)-1}{\kappa(M^{-1}A)+1}$ where $\kappa(M^{-1}A) \leq \frac{\gamma_{max}}{\gamma_{min}}$

Regular splittings

Definiton

- $A = M - N$ is a regular splitting if
  - M is nonsingular
  - $M^{-1} \geq 0$, $N \geq 0$ are element-wise nonnegative

Just remark that in this case $M^{-1}N = I - M^{-1}A$, and that we don't assume symmetry.

Theorem: Assume A is nonsingular, $A^{-1} \geq 0$, and $A = M - N$ is a regular splitting. Then $\rho(M^{-1}N) < 1$.

With this theory we cannot say much about the value of the convergence rate, but we have a comparison theorem:

Theorem: Let $A^{-1} \geq 0$, $A = M_1 - N_1$ and $A = M_2 - N_2$ be regular splittings.

If $N_2 \geq N_1$, then $1 > \rho(M_2^{-1}N_2) \geq \rho(M_1^{-1}N_1)$.

What can we say about inverse nonnegative matrices ?

Definition Let A be an $n \times n$ real matrix. A is called M-Matrix if

- (i) $a_{ij} \leq 0$ for $i \neq j$
- (ii) A is nonsingular
- (iii) $A^{-1} \geq 0$

Definition A square matrix A is *reducible* if there exists a permutation matrix P (re-ordering of equations) such that

$$PAP^T = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix}$$

A is *irreducible* if it is not reducible.

An M-Matrix A is inverse positive, i.e. $A^{-1} > 0$ if and only if it is irreducible

Irreducibility is easy to check.

Define a directed graph from the nonzero entries of a $n \times n$ matrix $A = (a_{ik})$:

- Nodes: $\mathcal{N} = \{N_i\}_{i=1\ldots n}$
- Directed edges: $\mathcal{E} = \{\overrightarrow{N_k N_l} | a_{kl} \neq 0\}$
- Matrix entries $\equiv$ weights of directed edges

$\Rightarrow$ 1:1 equivalence between matrices and weighted directed graphs

Theorem : A is irreducible $\Leftrightarrow$ the matrix graph is strongly connected, i.e. for each *ordered* pair (N_i, N_j) there is a path consisting of directed edges, connecting them.

Create a bidirectional graph (digraph) from a matrix in Julia. Create edge labels from off-diagonal entries and node labels combined from diagonal entries and node indices.

```
function create_graph(matrix)
    @assert size(matrix,1)==size(matrix,2)
    n=size(matrix,1)
    g=Graphs.SimpleDiGraph(n)
    elabel=[]
    nlabel=Any[]
    for i in 1:n
    push!(nlabel,"""$(i) \n $(round(matrix[i,i],sigdigits=3))""")
        for j in 1:n
        if i!=j && matrix[i,j]!=0
        add_edge!(g,i,j)
        push!(elabel,round(matrix[i,j],sigdigits=3))
        end
    end
    end
    g,nlabel,elabel
end;
```

Use `ExtendableSparse.fdrand` to create test matrices like the heatmatrix in the previous lecture:

```
fdrand(, nx)
fdrand(, nx, ny)
fdrand(, nx, ny, nz; matrixtype, update, rand, symmetric)
fdrand(nx)
```

Create matrix for a mock finite difference operator for a diffusion problem with random coefficients on a unit hypercube $\Omega \subset \mathbb{R}^d$. with $d=1$ if `nx>1 && ny==1 && nz==1`, $d=2$ if `nx>1 && ny>1 && nz==1` and $d=3$ if `nx>1 && ny>1 && nz>1` . In the symmetric case it corresponds to

$$\begin{aligned} -\nabla a \nabla u &= f \quad && \text{in } \Omega \\ a \nabla u \cdot \vec{n} + bu &= g \quad && \text{on } \delta\Omega \end{aligned}$$

The matrix is irreducibly diagonally dominant, has positive main diagonal entries and nonpositive off-diagonal entries, hence it has the M-Property. Therefore, its inverse will be a dense matrix with positive entries, and the spectral radius of the Jacobi iteration matrix $ho(I - D(A)^{-1}A) < 1$.

Moreover, in the symmetric case, it is positive definite.

Parameters+ default values:

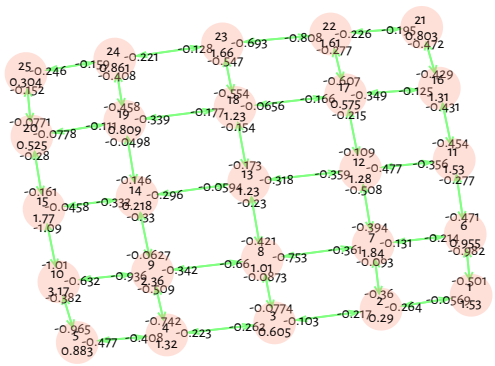
Parameter + default vale	Description
nx	Number of unknowns in x direction
ny	Number of unknowns in y direction
nz	Number of unknowns in z direction
matrixtype = SparseMatrixCSC	Matrix type
update = (A,v,i,j)-> A[i,j]+=v	Element update function
rand =()-> rand()	Random number generator
symmetric=true	Whether to create symmetric matrix or not

The sparsity structure is fixed to an orthogonal grid, resulting in a 3, 5 or 7 diagonal matrix depending on dimension. The entries are random unless e.g. `rand=()->1` is passed as random number generator. Tested for Matrix, SparseMatrixCSC, ExtendableSparseMatrix, Tridiagonal, SparseMatrixLNK and `:COO`

```
A2 = 25×25 SparseMatrixCSC{Float64, Int64} with 105 stored entries:
```

```
A2=fdrand(5,5,symmetric=false)
```

```
graph2,nlabel2,elabel2=create_graph(A2);
```



Let $A = (a_{ij})$ be an $n \times n$ matrix.

- A is *diagonally dominant* if for $i = 1 \ldots n$, $|a_{ii}| \geq \sum_{\substack{j=1\ldots n \\ j \neq i}} |a_{ij}|$
- A is *strictly diagonally dominant* (sdd) if for $i = 1 \ldots n$, $|a_{ii}| > \sum_{\substack{j=1\ldots n \\ j \neq i}} |a_{ij}|$
- A is *irreducibly diagonally dominant* (idd) if
  1. A is irreducible
  2. A is diagonally dominant: for $i = 1 \ldots n$, $|a_{ii}| \geq \sum_{\substack{j=1\ldots n \\ j \neq i}} |a_{ij}|$
  3. for at least one r, $1 \leq r \leq n$, $|a_{rr}| > \sum_{\substack{j=1\ldots n \\ j \neq r}} |a_{rj}|$

```
rowdiff (generic function with 1 method)
```

```
function rowdiff(A)
    [abs(A[i,i])-sum(abs,A[i,1:i-1])-sum(abs,A[i,i+1:end]) for i=1:size(A,1) ]
end
```

```
(-2.22045e-16, 0.280427)
```

```
extrema(rowdiff(A2))
```

```
using Tables
```

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Co
1	1.52685	-0.264271	0.0	0.0	0.0	-0.982148	0.0	0.0
2	-0.0569214	0.289605	-0.103476	0.0	0.0	0.0	-0.0929528	0.0
3	0.0	-0.216704	0.605017	-0.223252	0.0	0.0	0.0	-0.
4	0.0	0.0	-0.262253	1.31654	-0.477198	0.0	0.0	0.0
5	0.0	0.0	0.0	-0.407609	0.883214	0.0	0.0	0.0
6	-0.501379	0.0	0.0	0.0	0.0	0.954898	-0.130856	0.0
7	0.0	-0.360187	0.0	0.0	0.0	-0.213942	1.83561	-0.
8	0.0	0.0	-0.0774368	0.0	0.0	0.0	-0.360917	1.0
9	0.0	0.0	0.0	-0.742497	0.0	0.0	0.0	-0.
10	0.0	0.0	0.0	0.0	-0.964817	0.0	0.0	0.0

more

```
Tables.table(A2)
```

Given some matrix, we now have some nice recipies to establish nonsingularity and iterative method convergence:

- **Check if the matrix is irreducible.**
  - This is mostly the case for elliptic and parabolic PDEs and can be done by checking the graph of the matrix
- **Check if the matrix is strictly or irreducibly diagonally dominant**.
  - If yes, it is in addition nonsingular.
- **Check if main diagonal entries are positive and off-diagonal entries are nonpositive.**
  - If yes, in addition, the matrix is an M-Matrix, its inverse is nonnegative, and elementary iterative methods based on regular splittings converge.

These critera do not depend on the symmetry of the matrix!

Preconditioners

Jacobi preconditioner

Jacobi method: M=D, the diagonal of A

Theorem: If A is an M-Matrix, then the Jacobi preconditioner leads to a regular splitting.

Incomplete LU factorization

Idea (Varga, Buleev, $\approx$ **1960** : derive a preconditioner not from an additive decomposition but from the LU factorization.

- LU factorization has large fill-in. For a preconditioner, just limit the fill-in to a fixed pattern.
- Apply the standard LU factorization method, but calculate only a part of the entries, e.g. only those which are larger than a certain threshold value, or only those which correspond to certain predefined pattern.
- Result: incomplete LU factors $\boldsymbol{L}, \boldsymbol{U}$, remainder $\boldsymbol{R}$: $\boldsymbol{A} = \boldsymbol{LU} - \boldsymbol{R}$
- What about zero pivots which prevent such an algoritm from being computable ?

Theorem (Saad, Th. 10.2): If $\boldsymbol{A}$ is an M-Matrix, then the algorithm to compute the incomplete LU factorization with a given pattern is stable, i.e. does not detriorate due to zero pivots (main diagonal elements) Moreover, $\boldsymbol{A} = \boldsymbol{LU} - \boldsymbol{R} = \boldsymbol{M} - \boldsymbol{N}$ where $\boldsymbol{M} = \boldsymbol{LU}$ and $\boldsymbol{N} = \boldsymbol{R}$ is a regular splitting.

- Generally better convergence properties than Jacobi, though we cannot apply the comparison theorem for regular splittings to cpmpare between them
- Block variants are possible
- ILU Variants:
  - ILUM: ("modified"): add ignored off-diagonal entries to main diagonal
  - ILUT: ("threshold"): zero pattern calculated dynamically based on drop tolerance
  - ILU0: Drop all fill-in
  - Incomplete Cholesky: symmetric variant of ILU
- Dependence on ordering
- Can be parallelized using graph coloring
- Not much theory: experiment for particular systems and see if it works well
- I recommend it as the default initial guess for a sensible preconditioner

Further preconditioners

- Multigrid methods
- Domain decomposition
- Block variants of Jacobi, ILU...

Krylov subspace methods

- So far we considered simple iterative schemes, perhaps with preconditioners
- Krylov subspace methods are more sophisticateand and in many cases yield faster convergence than simple iterative schemes
- Reading material:
  - M. Gutknecht A Brief Introduction to Krylov Space Methods for Solving Linear Systems
  - J. Shewchuk Introduction to the Conjugate Gradient Method Without the Agonizing Pain
  - E.Carson, J.Liesen, Z. Strakoš: 70 years of Krylov subspace methods: The journey continues

Definition: Let $A \in \mathbb{R}^{N\times N}$ be nonsingular, let $0 \neq y \in \mathbb{R}^n$. The k-th *Krylov subspace* generated from A by y is defined as $\mathcal{K}_k(A, y) = \mathrm{span}\{y, Ay, \ldots, A^{k-1}y\}$.

Definition: Let $A \in \mathbb{R}^{N\times N}$ be nonsingular, let $0 \neq y \in \mathbb{R}^N$. An iterative method such that

$$u_k = u_0 + q_{k-1}(A)r_0 \in \mathcal{K}_k(A, r_0)$$

where q_{k-1} is a polynomial of degree k is called *Krylov subspace method*.

The idea of the GMRES method

Search the new iterate

$$u_k = u_0 + q_{k-1}(A)r_0 \in \mathcal{K}_k(A, r_0)$$

such that $r_k = ||Au_k - b||$ is minimized. This results in the *Generalized Minimum Residual* (GMRES) method.

- In order to find a good solution of this problem, we need to find an orthogonal basis of $\mathcal{K}_k \Rightarrow$ run an orthogonalization algorithm at each step
- One needs to store at least k vectors simultaneously $\Rightarrow$ usually, the iteration is restarted after a fixed number of iteration steps to keep the dimension of $\mathcal{K}_k$ limited
- There are preconditioned variants
- For symmetric matrices, one gets short three-term recursions, and there is no need to store a full Krylov basis. This results in the MINRES method
- Choosing q_k such that we get short recursions always will sacrifice some of the convergence estimates for GMRES. Nevertheless, this appraoch is tried quite often, resuling in particular in the BiCGstab and CGS methods.

Conjugated Gradients

This method assumes that the A and M are symmetric, positive definite.

$$\begin{aligned}
r_0 &= b - Au_0 \\
d_0 &= M^{-1}r_0 \\
\alpha_i &= \frac{(M^{-1}r_i, r_i)}{(Ad_i, d_i)} \\
u_{i+1} &= u_i + \alpha_i d_i \\
r_{i+1} &= r_i - \alpha_i A d_i \\
\beta_{i+1} &= \frac{(M^{-1}r_{i+1}, r_{i+1})}{(r_i, r_i)} \\
d_{i+1} &= M^{-1}r_{i+1} + \beta_{i+1} d_i
\end{aligned}$$

The convergence rate (error reduction in a norm defined by M and A) can be estimated via $\rho_{CG} = 2\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$ where $\kappa = \kappa(M^{-1}A)$. In fact, the distribution of the eigenvalues is important for convergence as well.

CG is a Krylov subspace method as well.

Complexity estimates

Solve linear system iteratively, for the error norm, assume $e_k \leq \rho^k e_0$. Iterate until $e_k \leq \epsilon$. Estimate the necessary number of iteration steps:

$$\begin{aligned} \rho^k e_0 &\leq \epsilon \\ k \ln \rho &< \ln \epsilon - \ln e_0 \\ k \geq k_\rho &= \left\lceil \frac{\ln e_0 - \ln \epsilon}{\ln \rho} \right\rceil \end{aligned}$$

⇒ we need at least k_ρ iteration steps to reach accuracy ϵ

The **ideal iterative solver**:

- $\rho < \rho_0 < 1$ independent of h resp. $N \Rightarrow k_\rho$ independent of N.
- A sparse ⇒ matrix-vector multiplication Au has complexity $O(N)$
- Solution of $Mv = r$ has complexity $O(N)$.

⇒ Number of iteration steps k_ρ independent of N Each iteration step has complexity $O(N)$
⇒ Overall complexity $O(N)$

Typical situation with second order PDEs and e.g. **Jacobi or ILU preconditioners:**

$$\begin{aligned} \kappa(M^{-1}A) &= O(h^{-2}) \quad (h \to 0) \\ \rho(I - M^{-1}A) &\leq \frac{\kappa(M^{-1}A) - 1}{\kappa(M^{-1}A) + 1} \approx 1 - O(h^2) \quad (h \to 0) \\ \rho_{CG}(I - M^{-1}A) &\leq \frac{\sqrt{\kappa(M^{-1}A)} - 1}{\sqrt{\kappa(M^{-1}A)} + 1} \approx 1 - O(h) \quad (h \to 0) \end{aligned}$$

- Mean square error of approximation $||u - u_h||_2 < h^\gamma$, in the simplest case $\gamma = 2$.

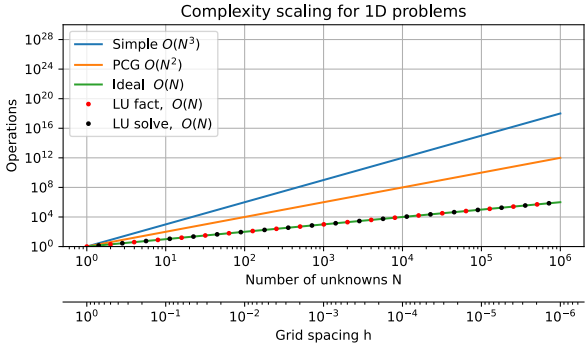
Back of the envelope complexity estimate

Simple iteration ($\delta = 2$) or preconditioned CG ($\delta = 1$):

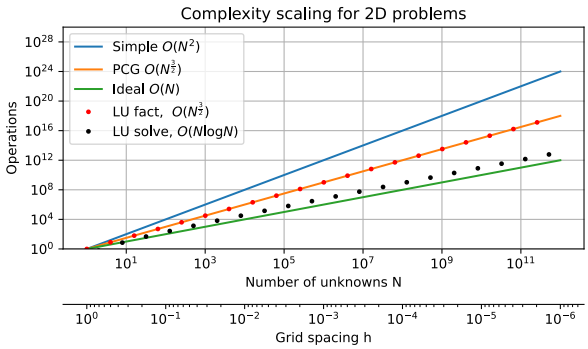
- $\rho = 1 - h^\delta$
  - $\Rightarrow \ln \rho \approx -h^\delta$
  - $\Rightarrow k_\rho = O(h^{-\delta})$
- d: space dimension:
  - $N \approx n^d$
  - $h \approx \frac{1}{n} \approx N^{-\frac{1}{d}}$
  - $\Rightarrow k_\rho = O(N^{\frac{\delta}{d}})$
- $O(N)$ complexity of one iteration step (e.g. Jacobi, ILUo)
- ⇒ Overall complexity $O(N^{1+\frac{\delta}{d}})$=$O(N^{\frac{d+\delta}{d}})$
  - Typical scaling for simple iteration scheme: $\delta = 2$ (Jacobi, ILUo ...)
  - Estimate for preconditioned CG (PCG) gives $\delta = 1$

Overview on complexity estimates

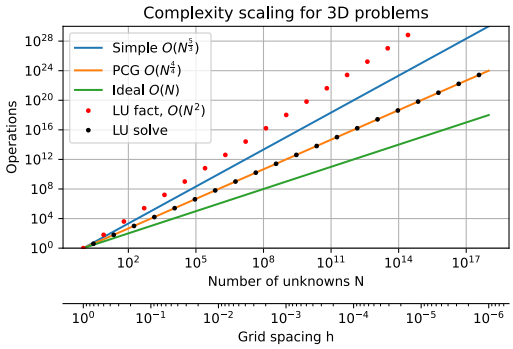
Space dim	*Simple*	*PCG*	LU fact	LU solve
1	$O(N^3)$	$O(N^2)$	$O(N)$	$O(N)$
2	$O(N^2)$	$O(N^{\frac{3}{2}})$	$O(N^{\frac{3}{2}})$	$O(N \log N)$
3	$O(N^{\frac{5}{3}})$	$O(N^{\frac{4}{3}})$	$O(N^2)$	$O(N^{\frac{4}{3}})$
Tendency with d ↑	↓	↓	↑↑	↑



- Sparse direct solvers, tridiagonal solvers are asymptotically optimal
- Non-ideal iterative solvers significantly worse than optimal



- Sparse direct solvers better than simple nonideal iterative solvers
- Sparse direct solvers on par preconditioned CG



- Sparse LU factorization is expensive: going from h to $h/2$ increases N by a factor of 8 and operation count by a factor of 64!
- Sparse LU solve on par preconditioned CG

Examples

Implementation of a Jacobi preconditioner: we need at least a constructor and `ldiv!` methods.

```
begin
    # Data structure: we store the inverse of the main diagonal
    struct JacobiPreconditioner
       invdiag::Vector
    end

    # Constructor:
    function JacobiPreconditioner(A::AbstractMatrix)
        n=size(A,1)
        invdiag=zeros(n)
        for i=1:n
            invdiag[i]=1.0/A[i,i]
        end
        JacobiPreconditioner(invdiag)
    end

    # Solution of preconditioning system   Mu=v
    # Method name and signature are compatible to IterativeSolvers.jl
    function  LinearAlgebra.ldiv!(u,precon::JacobiPreconditioner,v)
        invdiag=precon.invdiag
        n=length(invdiag)
        for i=1:n
            u[i]=invdiag[i]*v[i]
        end
    u
    end

    # In-place solution of preconditioning system
    LinearAlgebra.ldiv!(precon::JacobiPreconditioner,v)=ldiv!(v,precon,v)
end
```

Implement an LU preconditoner:

```
begin
    # Data structure: we store the inverse of a modfied main diagonal
    # and a pointer to the main diagonal entry of each column
    struct ILU0Preconditioner{Tv,Ti}
      A::SparseMatrixCSC{Tv,Ti}
      xdiag::Vector{Tv}
      idiag::Vector{Ti}
    end

    function ILU0Preconditioner(A)
        n=size(A,1)
        colptr=A.colptr
        rowval=A.rowval
        nzval=A.nzval
        idiag=zeros(Int64,n)
        xdiag=zeros(n)


        # calculate main diagonal indices
        for j=1:n
            for k=colptr[j]:colptr[j+1]-1
                i=rowval[k]
                if i==j
                    idiag[j]=k
                    break
                end
            end
        end

        # calculate modified inverse main diagonal
        for j=1:n
            xdiag[j]=1/nzval[idiag[j]]
            for k=idiag[j]+1:colptr[j+1]-1
                i=rowval[k]
                for l=colptr[i]:colptr[i+1]-1
                    if rowval[l]==j
                        xdiag[i]-=nzval[l]*xdiag[j]*nzval[k]
                        break
                    end
                end
            end
        end

        ILU0Preconditioner(A,xdiag,idiag)
    end

    # Solution of the preconditioning system
    function  LinearAlgebra.ldiv!(u,precon::ILU0Preconditioner, v)
        A=precon.A
        colptr=A.colptr
        rowval=A.rowval
        n=size(A,1)
        nzval=A.nzval
        xdiag=precon.xdiag
        idiag=precon.idiag
        T=eltype(v)

        for j=1:n
           u[j]=xdiag[j]*v[j]
        end

        for j=n:-1:1
          for k=idiag[j]+1:colptr[j+1]-1
            i=rowval[k]
            u[i]-=xdiag[i]*nzval[k]*u[j]
          end
        end

        for j=1:n
            for k=colptr[j]:idiag[j]-1
               i=rowval[k]
               u[i]-=xdiag[i]*nzval[k]*u[j]
            end
        end
    u
    end

    LinearAlgebra.ldiv!(precon::ILU0Preconditioner,v)=ldiv!(v,precon,v)

end
```

Implement a simple iteration scheme

```
simple (generic function with 1 method)
```

```
begin
    function simple!(u,A,b;tol=1.0e-10,log=true,maxiter=100,Pl=nothing)
    res=A*u-b # initial residual
    r0=norm(res) # residual norm
    history=[r0]  # intialize history recording
    for i=1:maxiter
        u=u-ldiv!(Pl,res) # solve preconditioning system and update solution
        res=A*u-b         # calculate residual
        r=norm(res)       # residual norm
        push!(history,r)  # record in history
        if (r/r0)<tol     # check for relative tolerance
        return u,Dict( :resnorm => history)
        end
    end
    return u,Dict( :resnorm =>history )
    end

    simple(A,b;tol=1.0e-10, log=true,maxiter=100,Pl=nothing)=simple!
  (zeros(length(b)),A,b,tol=tol,maxiter=maxiter,log=log,Pl=Pl)
end
```

Test problem

```
N = 10000
```

```
N=10000
```

```
true
```

```
dim=3; symmetric=true
```

```
n = 22
```

```
n=Int(ceil(N^(1/dim)))
```

```
A1=fdrand([n for i=1:dim]...;symmetric);
```

```
b1 =
```

```
[0.00382312, 0.000782596, 0.00288277, 0.00280951, 0.00111274, 0.00377058, 0.00240034, 0.00
```

```
b1=A1*ones(size(A1,1))
```

```
A1Jacobi=JacobiPreconditioner(A1);
```

```
A1ILU0=ILU0Preconditioner(A1);
```

```
tol = 1.0e-10
```

```
tol=1.0e-10
```

Solve the test problem with the `simple` iterative solver:

Convergence simple+CG

```
maxiter = 10010
```

```
maxiter=10010
```

```
(
  1: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,  more ,1.0]
  2: Dict(:resnorm => [0.0805066, 0.0651035, 0.0581066, 0.0543425, 0.0515648,  more
)
```

```
sol_simple_jacobi,hist_simple_jacobi=simple(A1,b1;tol,maxiter,log=true,Pl=A1Jacobi)
```

```
(
  1: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,  more ,1.0]
  2: Dict(:resnorm => [0.0805066, 0.0571623, 0.0506342, 0.0470401, 0.0445629,  more
)
```

```
sol_simple_ilu0,hist_simple_ilu0=simple(A1,b1;tol,maxiter,log=true,Pl=A1ILU0)
```

Solve the test problem with the CG iterative solver from IterativeSolvers.jl:

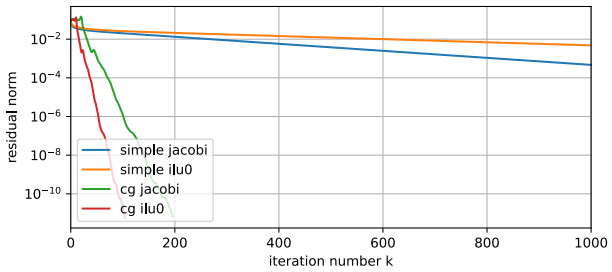
```
([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,  more ,1.0], Converged after 197 iteratior
```

```
sol_cg_jacobi,hist_cg_jacobi=cg(A1,b1; reltol=tol,log=true,maxiter,Pl=A1Jacobi)
```

```
([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,  more ,1.0], Converged after 105 iteratior
```

```
sol_cg_ilu0,hist_cg_ilu0=cg(A1,b1; reltol=tol,log=true,maxiter,Pl=A1ILU0)
```

- As we see, all CG variants converge within the given number of iterations steps.
- Precoditioning helps
- The better the preconditioner, the faster the iteration (though this also depends on the initial value)
- The behaviour of the CG residual is not monotone



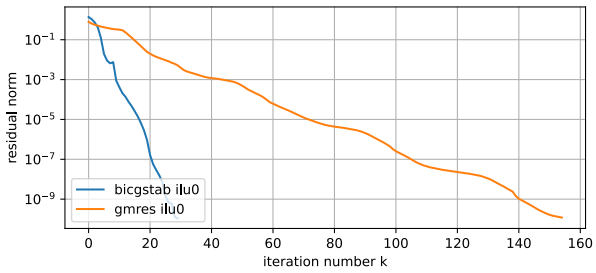
Convergence: ILU + bicgstab

```
([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,  more ,1.0], Converged after 30 iterations
```

```
sol_bicgstab_ilu0,hist_bicgstab_ilu0=bicgstabl(A1,b1,reltol=tol,log=true,max_mv_produc
ts=2*maxiter,Pl=A1ILU0)
```

```
([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,  more ,1.0], Converged after 155 iteratior
```

```
sol_gmres_ilu0,hist_gmres_ilu0=gmres(A1,b1;Pl=A1ILU0,reltol=tol,log=true,maxiter)
```



Solution times

Compare Sparse direct solver, PCG and bicgstab:

```
BenchmarkTools.Trial: 140 samples with 1 evaluation.
 Range (min … max):  34.411 ms … 41.491 ms  ┊ GC (min … max): 0.00% … 3.50%
 Time  (median):     35.298 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   35.716 ms ±  1.157 ms  ┊ GC (mean ± σ):  0.88% ± 1.64%
```


```
Memory estimate: 24.05 MiB, allocs estimate: 66.
```

```
@benchmark A1\b1
```

```
BenchmarkTools.Trial: 182 samples with 1 evaluation.
 Range (min … max):  26.992 ms …  32.119 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     27.550 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   27.602 ms ± 412.222 μs  ┊ GC (mean ± σ):  0.04% ± 0.52%
```


```
Memory estimate: 417.19 KiB, allocs estimate: 338.
```

```
if hist_cg_ilu0.isconverged
    @benchmark cg(A1,b1; reltol=tol,log=true,maxiter,Pl=A1ILU0)
end
```

```
BenchmarkTools.Trial: 119 samples with 1 evaluation.
 Range (min … max):  37.073 ms … 46.741 ms  ┊ GC (min … max): 0.00% … 3.56%
 Time  (median):     42.495 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   42.278 ms ±  2.115 ms  ┊ GC (mean ± σ):  1.91% ± 1.95%
```


```
Memory estimate: 28.11 MiB, allocs estimate: 586.
```

```
if hist_bicgstab_ilu0.isconverged
    @benchmark
bicgstabl(A1,b1,reltol=tol,log=true,max_mv_products=2*maxiter,Pl=A1ILU0)
end
```

Final remarks

- Iterative solvers are a combination of preconditioning and iteration scheme. Krylov method based iteration schemes (CG, BiCGstab, GMRES...) provide significant advantages.
- Iterative solvers can beat direct solvers for problems stemming from the discretization of PDEs in 3D
- Convergence of iterative solvers needs more matrix properties than just nonsingularity
- Parallelization is easier for iterative solvers than for sparse direct solvers

Julia packages

- Iteration schemes
  - Krylov.jl (closer to current research)
  - IterativeSolvers.jl (used in this notebook)
- Preconditioners
  - ILUZero.jl for zero fill-in ILU decomposition
  - IncompleteLU.jl - ILU with drop tolerance
  - AlgebraicMultigrid.jl - Multigrid methods with automatic coarsening
- LinearSolve.jl - Attempt on a "on-stop shop" for linear system solution
- ExtendableSparse.jl - Simple+ efficient sparse matrix building + integration with preconditioners and various sparse direct solvers

```
pyplot (generic function with 1 method)
```

Table of Contents