Iterative methods for linear systems

Advanced Topics from Scientific Computing
TU Berlin Winter 2024/25
Notebook 08
Jürgen Fuhrmann

```
begin
    using LinearAlgebra
    using SparseArrays
    using LaTeXStrings,Colors
    using CairoMakie
    CairoMakie.activate!(type="svg")
    using BenchmarkTools
    BenchmarkTools.DEFAULT_PARAMETERS.seconds=1

end;
```

Reading: Y.Saad: Iterative Methods for Sparse 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

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

$\Rightarrow$ iterative scheme: put the "old" value on the right hand side and the ""new" value on the left hand side:

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

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

Otherwise it leads to a sequence of approximations

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

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: Assume M, A are symmetrix and positive definite (spd).

Scaled Richardson iteration with preconditioner M

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

Spectral equivalence estimate

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

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

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

$\Rightarrow$ convergence rate with optimal parameter:

$$\rho_{opt} \le \frac{\kappa(M^{-1}A)-1}{\kappa(M^{-1}A)+1}$$

where

$$\kappa(M^{-1}A) \le \frac{\gamma_{max}}{\gamma_{min}}$$

Convergence estimate for regular splittings

Definiton

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

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

Theorem: Assume A is nonsingular, $A^{-1} \ge 0$, 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 $A^{-1} \ge 0$, $A = M_1 - N_1$ and $A = M_2 - N_2$ be regular splittings.

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

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) $a_{ij} \le 0$ for $i \ne j$
- (ii) A is nonsingular
- (iii) $A^{-1} \ge 0$

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

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

Theorem : A is irreducible ⇔ 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.

Sparse matrix generation for solver benchmarking

Create a sparse matrix with approximately N unknowns on a d-dimensional lattice grid. After a discourse contribution by A. Braunstein:

```
begin
    A ⊕ B = kron(I(size(B, 1)), A) + kron(B, I(size(A, 1)))
    function lattice(n; Tv = Float64)
        d = fill(2 * one(Tv), n)
        d[1] = one(Tv)
        d[end] = one(Tv)
        spdiagm(1 => -ones(Tv, n - 1), 0 => d, -1 => -ones(Tv, n - 1))
    end
    lattice(L...; Tv = Float64) = lattice(L[1]; Tv) ⊕ lattice(L[2:end]...; Tv)
end;
```

```
function mysparse(N; dim=1, Tv = Float64, dd = 1.0e-2)
    n = N^(1 / dim) |> ceil |> Int
    lattice([n for i in 1:dim]...; Tv) + Tv(dd) * I
end;
```

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. Plot the graph.

```
using Graphs,GraphPlot
```

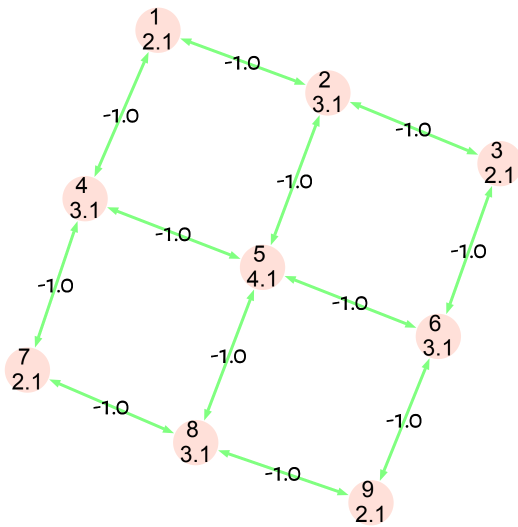
```
plot_graph (generic function with 1 method)
```

```
function plot_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

    GraphPlot.gplot(g;
                nodelabel=nlabel,
                edgelabel=elabel,
                nodefillc=RGBA(1.0,0.6,0.5,0.3),
                EDGELABELSIZE=4,
                NODESIZE=0.1,
                    edgestrokec=RGB(0.5,1.0,0.5),
                EDGELINEWIDTH=0.5,
                    arrowlengthfrac=0.05
                    )
end
```

```
A3 = 9×9 Matrix{Float64}:
  2.1  -1.0   0.0  -1.0   0.0   0.0   0.0   0.0   0.0
 -1.0   3.1  -1.0   0.0  -1.0   0.0   0.0   0.0   0.0
  0.0  -1.0   2.1   0.0   0.0  -1.0   0.0   0.0   0.0
 -1.0   0.0   0.0   3.1  -1.0   0.0  -1.0   0.0   0.0
  0.0  -1.0   0.0  -1.0   4.1  -1.0   0.0  -1.0   0.0
  0.0   0.0  -1.0   0.0  -1.0   3.1   0.0   0.0  -1.0
  0.0   0.0   0.0  -1.0   0.0   0.0   2.1  -1.0   0.0
  0.0   0.0   0.0   0.0  -1.0   0.0  -1.0   3.1  -1.0
  0.0   0.0   0.0   0.0   0.0  -1.0   0.0  -1.0   2.1
```

```
A3=mysparse(5,dim=2,dd=0.1)|>Matrix
```




```
1 plot_graph(A3)
```

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

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

The matrices created with mysparse are strictly diagonally dominant:

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

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

```
▸(0.1, 0.1)
```

```
1 extrema(rowdiff(mysparse(20;dim=2,dd=0.1)))
```

heatmatrix2d! (generic function with 1 method)

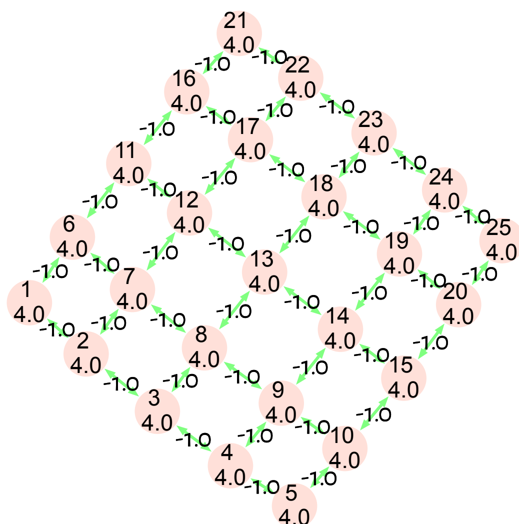
```
function heatmatrix2d!(A,n;α=1)
    function update_pair(A,v,i,j)
        A[i,j]+=-v
        A[j,i]+=-v
        A[i,i]+=v
        A[j,j]+=v
    end
    N=n^2
    h=1.0/(n-1)
    l=1
        for j=1:n
            for i=1:n
                if i<n
                    update_pair(A,1.0,l,l+1)
                end
                if i==1|| i==n
                    A[l,l]+=α
                end
                if j<n
                    update_pair(A,1,l,l+n)
                end
                if j==1|| j==n
                    A[l,l]+=α
                end
                l=l+1
            end
        end
    A
end

```

The heat conduction matrix from previous lecture is irreducibly diagonally dominant:

A4 = 25×25 SparseMatrixCSC{Float64, Int64} with 105 stored entries:

```
A4=heatmatrix2d!(spzeros(25,25),5)
```




```
plot_graph(A4)
```

▸ (0.0, 2.0)

```
extrema(rowdiff(A4))
```

Let A be sdd or idd. Assume that $a_{ii} > 0$ and $a_{ij} \leq 0$ for $i \neq j$. Then A is an M-Matrix.

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.

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
    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
```

```
JacobiPreconditioner([0.497512, 0.332226, 0.497512, 0.332226, 0.249377, 0.332226, 0.497512
```

```
JacobiPreconditioner(mysparse(5,dim=2))
```

Incomplete LU factorization

Idea (Varga, Buleev, ≈ 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 L, U, remainder R: $A = LU - R$
- What about zero pivots which prevent such an algoritm from being computable ?

Theorem (Saad, Th. 10.2): If 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, $A = LU - R = M - N$ where $M = LU$ and $N = R$ 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 ILU0 as the default initial guess for a sensible preconditioner

Julia has a very good ILU0 package: ILUZero.jl by Matt Covalt:

```
1 using ILUZero
```

```
▸ILU0Precon(16, 16, [1, 3, 5, 7, 8, 10, 12, 14, 15, … more ,25], [2, 5, 3, 6, 4, 7, 8, 6, 9, …
```

```
1 ILUZero.ilu0(mysparse(10,dim=2))
```

Further preconditioners

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

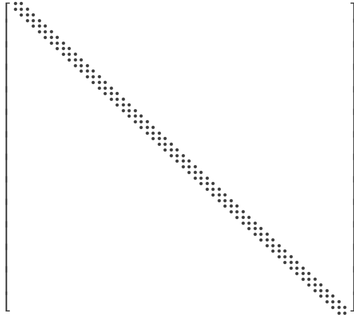
Implementation of the simple iteration scheme

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

```
begin
    function simple!(u,A,b;rtol=1.0e-10,atol=1.0e-10, itmax=5_000,M=I)
    res=A*u-b # initial residual
    r0=norm(res) # residual norm
    history=[r0]  # intialize history recording
    for i=1:itmax
        u=u-ldiv!(M,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)<rtol || r<atol      # check for relative tolerance
            return u,(residuals=history,niter=i,solved=true)
        end
    end
    return u,(residuals=history,niter=itmax,solved=false)
    end

    simple(A,b;atol=1.0e-10,rtol=1.0e-10,itmax=5_000,M=I)=simple!
(zeros(length(b)),A,b;          atol,itmax,M)
end
```

```
A1 = 10000×10000 SparseMatrixCSC{Float64, Int64} with 49600 stored entries:
```



```
A1=mysparse(10_000,dim=2,dd=1.0e-2)
```

```
x1 =
▸[0.477921, 0.388802, 0.290118, 0.312901, 0.281233, 0.711237, 0.68907, 0.384397, 0.868857
```

```
x1=rand(size(A1,2))
```

```
b1 =
▸[0.00024953, -0.311695, -0.370068, -0.399769, -0.827744, 0.692448, 0.350582, -1.30427, 1
```

```
b1=A1*x1
```

```
jac1 =
▸JacobiPreconditioner([0.497512, 0.332226, 0.332226, 0.332226, 0.332226, 0.332226, 0.332226
```

```
jac1=JacobiPreconditioner(A1)
```

```
▸([0.47792, 0.388801, 0.290117, 0.3129, 0.281232, 0.711236, 0.689068, 0.384396, 0.868855,
```

```
u1_jac,stats1_jac=simple(A1,b1;M=jac1)
```

```
1.74988670718218e-6
```

```
norm(u1_jac-x1,Inf)
```

```
ilu1 =
▸ILU0Precon(10000, 10000, [1, 3, 5, 7, 9, 11, 13, 15, 17, ⋯ more ,19801], [2, 101, 3, 102, 4,
```

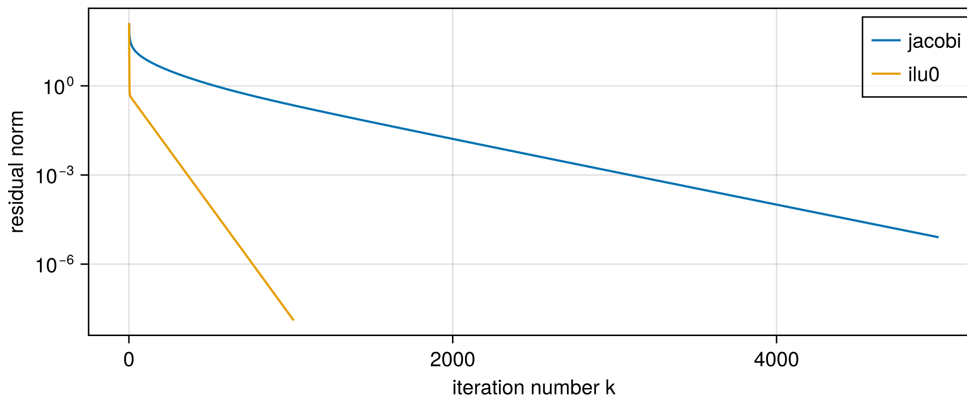
```
ilu1=ilu0(A1)
```

```
▸([0.477921, 0.388802, 0.290118, 0.312901, 0.281233, 0.711237, 0.68907, 0.384397, 0.86885
```

```
u1_ilu,stats1_ilu=simple(A1,b1;M=ilu1)
```

```
1.360486734380828e-8
```

```
norm(u1_ilu-x1,Inf)
```



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 $\mathcal{K}_k(A, y) = \text{span}\{y, Ay, \dots, A^{k-1}y\}$.

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. In this case, the orthogonalization leads to short recursions, and there is no need to store all the Krylov basis vectors.

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

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.

Krylov methods in Julia

Julia has a very good package for Krylov subspace methods: Krylov.jl by Alexis Montoison and Dominique Orban:

```
1 using Krylov
```

▸ ([0.477921, 0.388802, 0.290118, 0.312901, 0.281233, 0.711237, 0.68907, 0.384397, 0.86885

```
1 u1_cgjac,stats1_cgjac=cg(A1,b1;M=jac1,ldiv=true,history=true,rtol=1.0e-10)
```

1.8151410152711378e-8

```
1 norm(u1_cgjac-x1,Inf)
```

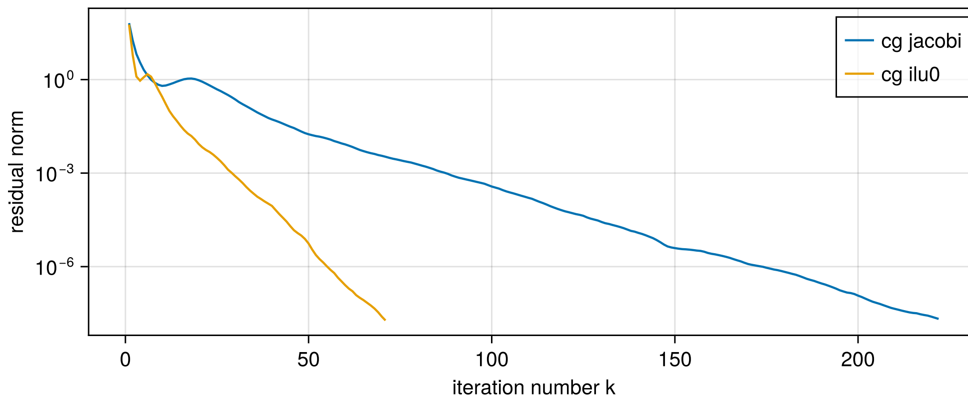
▸ ([0.477921, 0.388802, 0.290118, 0.312901, 0.281233, 0.711237, 0.68907, 0.384397, 0.86885

```
1 u1_cgilu,stats1_cgilu=cg(A1,b1;M=ilu1,ldiv=true,history=true,rtol=1.0e-10)
```

9.502263531580013e-9

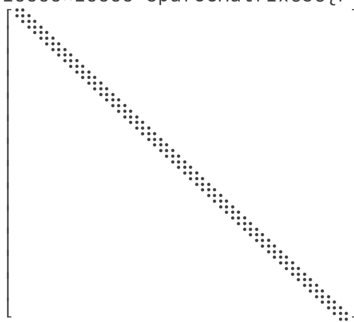
```
1 norm(u1_cgilu-x1,Inf)
```

Add plots of simple results ? ☐



Nonsymmmetric problems

A1N = 10000×10000 SparseMatrixCSC{Float64, Int64} with 49600 stored entries:



```
A1N=A1*Diagonal(rand(0.01:0.01:1,size(A1,2)))
```

b1n =
▸[-0.141849, -0.552453, -0.25015, -0.0853089, -0.172765, -0.215805, -0.928566, 0.431573,

```
b1n=A1N*x1
```

jac1n =
▸JacobiPreconditioner([1.10558, 1.00675, 0.604047, 0.455104, 0.651423, 3.6914, 33.2226, 0.3

```
jac1n=JacobiPreconditioner(A1N)
```

ilu1n =
▸ILU0Precon(10000, 10000, [1, 3, 5, 7, 9, 11, 13, 15, 17, ⋯ more ,19801], [2, 101, 3, 102, 4,

```
ilu1n=ilu0(A1N)
```

▸([0.477919, 0.3888, 0.290117, 0.3129, 0.281232, 0.711229, 0.68899, 0.384396, 0.868856, ⋯

```
u1n_jac,stats1n_jac=simple(A1N,b1n;M=jac1n)
```

9.004396049597241e-5

```
norm(u1n_jac-x1,Inf)
```

▸([0.477921, 0.388802, 0.290118, 0.312901, 0.281233, 0.711237, 0.689069, 0.384397, 0.8688

```
u1n_ilu,stats1n_ilu=simple(A1N,b1n;M=ilu1n)
```

1.0426034557919905e-6

```
norm(u1n_ilu-x1,Inf)
```

▸([0.477921, 0.388802, 0.290118, 0.312901, 0.281233, 0.711237, 0.68907, 0.384397, 0.86885

```
u1n_cgjac,stats1n_cgjac=cg(A1N,b1n;M=jac1n,ldiv=true,history=true,rtol=1.0e-
10,itmax=5_000)
```

6.131244714713269e-7

```
norm(u1n_cgjac-x1,Inf)
```

▸ ([0.477921, 0.388802, 0.290118, 0.312901, 0.281233, 0.711237, 0.68907, 0.384397, 0.86885

```julia
u1n_cgilu,stats1n_cgilu=cg(A1N,b1n;M=ilu1n,ldiv=true,history=true,rtol=1.0e-10,itmax=5_000)
```

3.3938112131703946e-7

```julia
norm(u1n_cgilu-x1,Inf)
```

▸ ([0.477921, 0.388802, 0.290118, 0.312901, 0.281233, 0.711237, 0.689067, 0.384397, 0.8688

```julia
u1n_gmresjac,stats1n_gmresjac=gmres(A1N,b1n;M=jac1n,ldiv=true,history=true,rtol=1.0e-10,itmax=5_000)
```

3.1639490543078352e-6

```julia
norm(u1n_gmresjac-x1,Inf)
```

▸ ([0.477921, 0.388802, 0.290118, 0.312901, 0.281233, 0.711237, 0.68907, 0.384397, 0.86885

```julia
u1n_gmresilu,stats1n_gmresilu=gmres(A1N,b1n;M=ilu1n,ldiv=true,history=true,rtol=1.0e-10,itmax=5_000)
```

2.9716759669673465e-7

```julia
norm(u1n_gmresilu-x1,Inf)
```

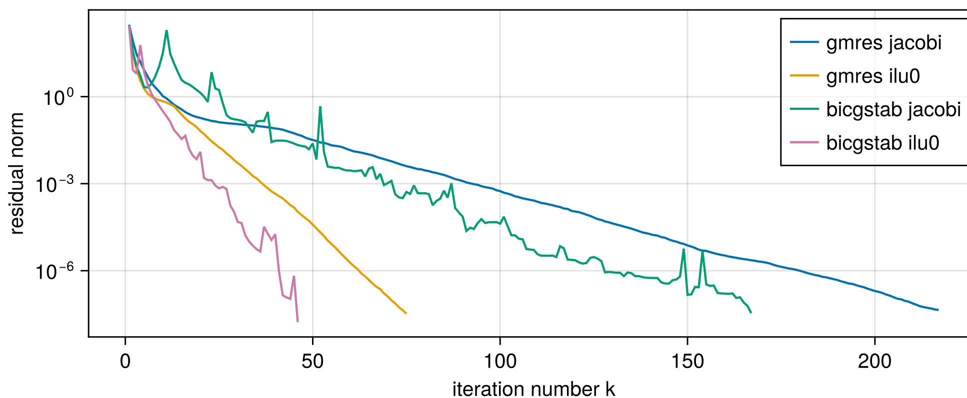
▸ ([0.477921, 0.388802, 0.290118, 0.312901, 0.281233, 0.711237, 0.689069, 0.384397, 0.8688

```julia
u1n_bicgstabjac,stats1n_bicgstabjac=bicgstab(A1N,b1n;M=jac1n,ldiv=true,history=true,rtol=1.0e-10)
```

▸ ([0.477921, 0.388802, 0.290118, 0.312901, 0.281233, 0.711237, 0.68907, 0.384397, 0.86885

```julia
u1n_bicgstabilu,stats1n_bicgstabilu=bicgstab(A1N,b1n;M=ilu1n,ldiv=true,history=true,rtol=1.0e-10)
```

Add plots of simple and cg results ? ☐



Complexity scaling

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:

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

$\Rightarrow$ 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 $\Rightarrow$ matrix-vector multiplication Au has complexity $O(N)$
- Solution of $Mv = r$ has complexity $O(N)$.

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

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

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

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

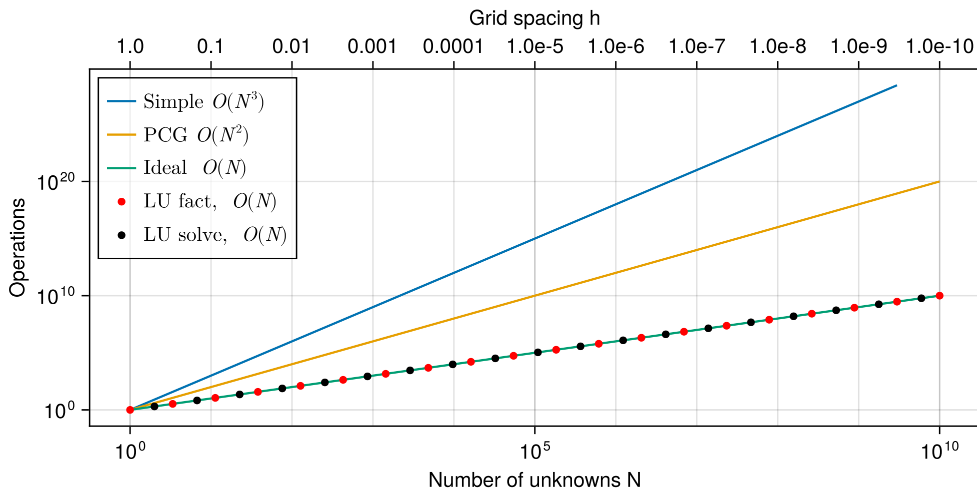
Back of the envelope complexity estimate

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

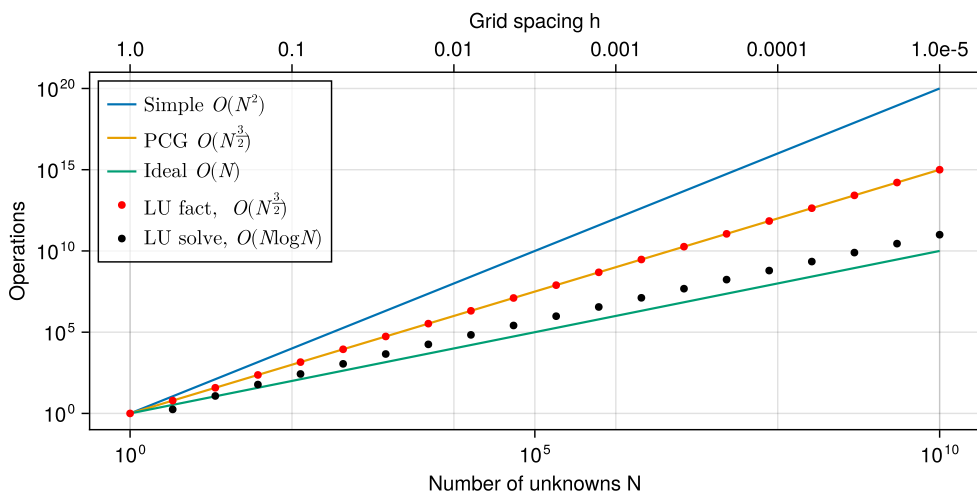
- $\rho = 1 - h^\delta$
  - $\Rightarrow \ln\rho \approx -h^\delta$
  - $\Rightarrow k_\rho = O(h^{-\delta})$
- d: space dimension:
  - $N \approx n^d$
  - $h \approx \frac{1}{n} \approx N^{-\frac{1}{d}}$
  - $\Rightarrow k_\rho = O(N^{\frac{\delta}{d}})$
- $O(N)$ complexity of one iteration step (e.g. Jacobi, ILU0)
- $\Rightarrow$ Overall complexity $O(N^{1+\frac{\delta}{d}})$=$O(N^{\frac{d+\delta}{d}})$
  - Typical scaling for simple iteration scheme: $\delta = 2$ (Jacobi, ILU0 $\dots$)
  - Estimate for preconditioned CG (PCG) gives $\delta = 1$

Overview on complexity estimates

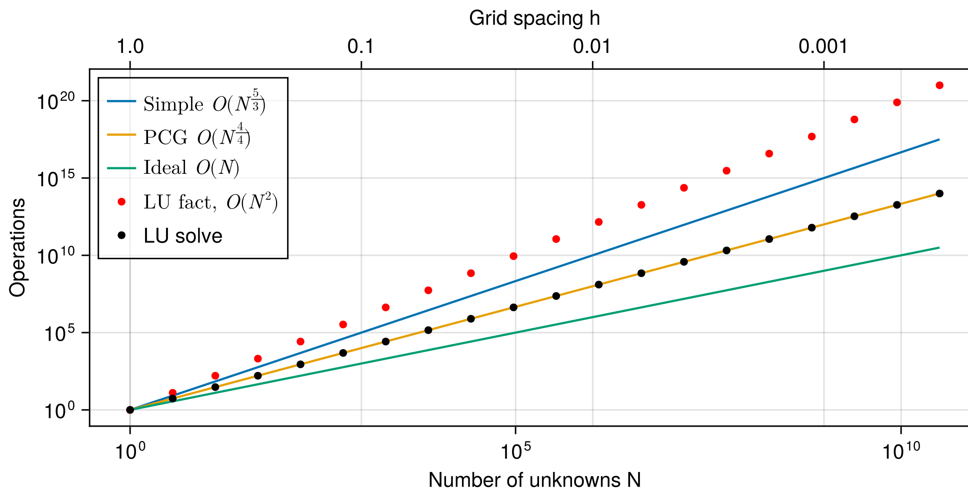
Space dim	*Simple*	*PCG*	LU fact	LU solve
1	$O(N^3)$	$O(N^2)$	$O(N)$	$O(N)$
2	$O(N^2)$	$O(N^{\frac{3}{2}})$	$O(N^{\frac{3}{2}})$	$O(N\log N)$
3	$O(N^{\frac{5}{3}})$	$O(N^{\frac{4}{3}})$	$O(N^2)$	$O(N^{\frac{4}{3}})$
Tendency with d $\uparrow$	$\downarrow$	$\downarrow$	$\uparrow\uparrow$	$\uparrow$



- 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 with preconditioned CG

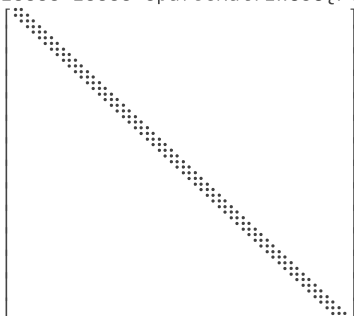
LinearSolve.jl

LinearSolve.jl (documentation) provides a common interface to many direct and iterative linear system solvers. The following video by its main author Chris Rackauckas reflects an early state of the package:

LinearSolve.jl: Because A\b is Not Good Enough | Chris Rackauckas | Julia...

```
1 using LinearSolve
```

A2 = 10000×10000 SparseMatrixCSC{Float64, Int64} with 49600 stored entries:



```
1 A2=mysparse(10_000,dim=2)
```

```
x2 =
[0.628121, 0.889718, 0.371049, 0.633373, 0.257532, 0.135517, 0.967475, 0.0166119, 0.6603
```

```
1 x2=rand(size(A2,1))
```

```
b2 =
[0.350472, 0.811409, -0.788911, 0.958453, -0.0716619, -1.34567, 2.1656, -1.62487, 0.8516
```

```
1 b2=A2*x2
```

```
p2 = LinearProblem. In-place: true
    b: 10000-element Vector{Float64}:
      0.35047238232310685
      0.8114088850153378
     -0.78891086931072
      0.9584531930284419
     -0.07166189170183623
     -1.3456662460976176
      2.1655962534568087
      ⋮
     -1.1162491730420085
      0.7717681232720359
      0.3985702563415793
      0.23261471327583283
     -2.257868814748549
      1.0936981269108266
```

```
1 p2=LinearProblem(A2,b2)
```

```
u2 =
SciMLBase.LinearSolution{Float64, 1, Vector{Float64}, Nothing, LinearSolve.DefaultLinearSolver, L
```

```
1 u2=solve(p2)
```

```
1.454392162258955e-14
```

```
1 norm(u2-x2,Inf)
```

LinearSolve.jl uses built-in heuristics to choose the solution method. One can investigate the choice made:

```
DefaultLinearSolver(DefaultAlgorithmChoice.UMFPACKFactorization = 6)
```

```
1 u2.alg
```

One can overwrite this. We can e.g. choose Sparspak.jl, a re-implementation of the Sparspak Fortran code in Julia as a solver:

```
u2s =
SciMLBase.LinearSolution{Float64, 1, Vector{Float64}, Nothing, LinearSolve.SparspakFactorization,
```

```
1 u2s=solve(p2,SparspakFactorization())
```

```
7.28583859910259e-15
```

```
1 norm(u2s-x2,Inf)
```

One can also access iterative solvers and preconditioners:

```
u2i =
SciMLBase.LinearSolution{Float64, 1, Vector{Float64}, Float64, LinearSolve.KrylovJL{typeof(Krylov
```

```
1 u2i=solve(p2,KrylovJL_CG(rtol=1.0e-12),Pl=ilu0(A2))
```

```
KrylovJL(cg! (generic function with 2 methods), 0, 0, nothing, (), Pairs(:rtol ⇒ 1.0e-12
```

```
1 u2i.alg
```

```
7.070195429115245e-9
```

```
1 norm(u2i-x2,Inf)
```

Scaling test

Maximum problem size is defined as $10 \cdot 2^{k_{max}}$. On a good notebook `kmax=13` is bearable and results in 81920 unknowns.

```
kmax = 13
```

```
kmax=13
```

```
Ns = ▸[10, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240, 20480, 40960, 81920]
```

```
Ns = [10 * 2^k for k in 0:kmax]
```

```
dims = ▸(1, 2, 3)
```

```
dims = (1, 2, 3)
```

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

```
function benchmark(solve, dim, N;
                   tol = 100 * sqrt(eps(Float64)),
                   trun = 0.2)
    A = mysparse(N;dim, dd=1.0e-3)
    n = size(A, 1)
    x = ones(n)
    b = A * x
    tall = 0.0
    tmin = 1.0e10
    while tall < trun
        t = @elapsed y = solve(A, b)
        nm = norm(x - y, Inf)
        if nm > tol
            @warn "insufficient accuracy: error=$(nm), eps: $(tol), ($nx,$ny,$nz)"
        end
        tmin = min(t, tmin)
        tall += t
    end
    tmin
end

```

```
function benchmarks(solve, Ns; dims = dims)
    Ts = zeros(length(dims), length(Ns))
    for i in 1:length(Ns)
        for dim in dims
            Ts[dim, i] = benchmark(solve, dim, Ns[i])
        end
    end
    Ts
end;
```

```
Tumfpack =
3×14 Matrix{Float64}:
 7.332e-6    1.1437e-5  1.9173e-5  3.45e-5      …  0.00838318  0.0178858  0.0391585
 1.1757e-5  1.6246e-5  3.4527e-5  5.105e-5        0.0300066   0.069343   0.151633
 1.9393e-5  1.93e-5    5.083e-5   0.000202372     0.285623    0.997055   4.33564
```

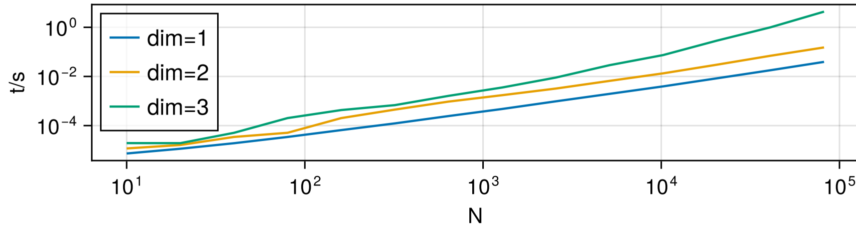
```
Tumfpack = benchmarks(Ns) do A, b
    solve(LinearProblem(A, b), UMFPACKFactorization())
end
```

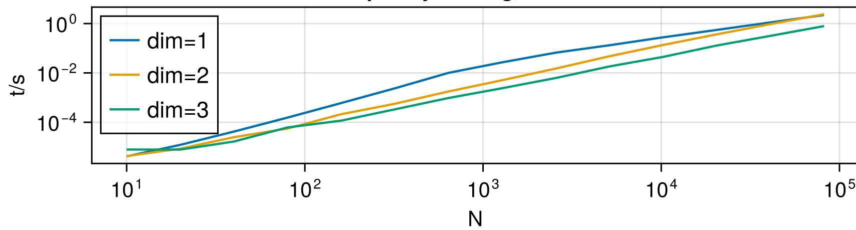
```
Tcgjacobi =
3×14 Matrix{Float64}:
 4.144e-6  1.2141e-5  4.1931e-5  …  0.276967   0.553069  1.11139   2.21964
 4.244e-6  8.527e-6   2.4762e-5     0.135308   0.365828  0.949117  2.39049
 7.848e-6  7.805e-6   1.6569e-5     0.0445749  0.128854  0.317362  0.789272
```

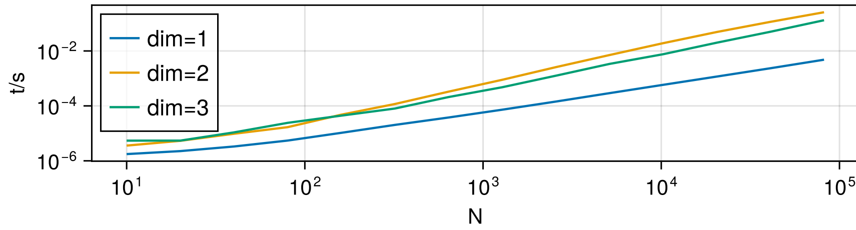
```
Tcgjacobi = benchmarks(Ns) do A, b
    solve(LinearProblem(A, b), KrylovJL_CG(), Pl=JacobiPreconditioner(A))
end
```

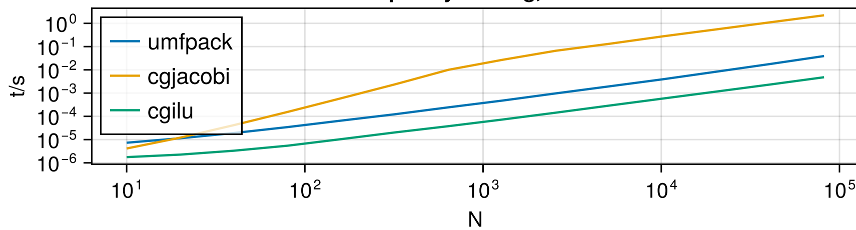
```
Tcgilu =
3×14 Matrix{Float64}:
 1.761e-6  2.261e-6  3.327e-6   5.465e-6   …  0.00116913  0.00234748  0.00480707
 3.588e-6  5.357e-6  9.669e-6   1.6809e-5     0.0489637   0.114756    0.256702
 5.434e-6  5.449e-6  1.0849e-5  2.4116e-5     0.0200063   0.0499793   0.13144
```

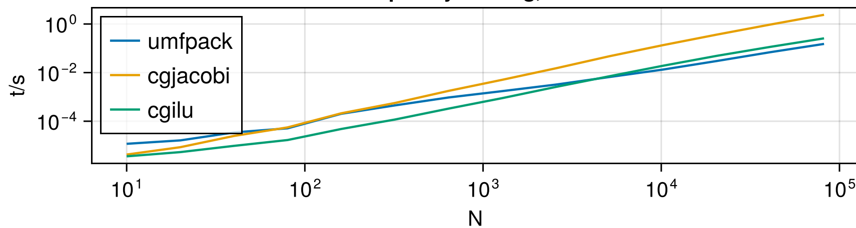
```
1 Tcgilu = benchmarks(Ns) do A, b
2     solve(LinearProblem(A, b), KrylovJL_CG(), Pl=ilu0(A))
3 end
```

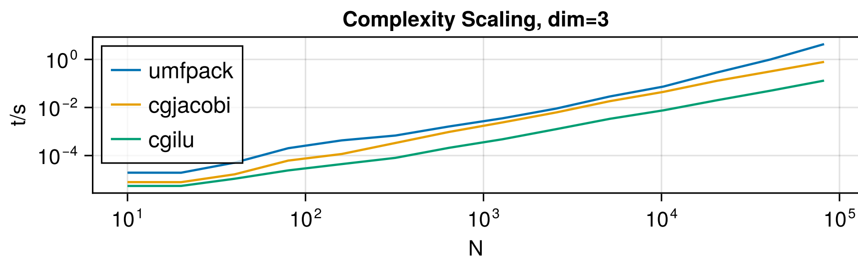












Compare Sparse direct solver, PCG and bicgstab:

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 coming from the discretization of PDEs in 3D
- Convergence theory for iterative solvers needs more matrix properties than just nonsingularity

Julia packages

- Iteration schemes
  - Krylov.jl
  - IterativeSolvers.jl
- Preconditioners
  - ILUZero.jl for zero fill-in ILU decomposition
  - IncompleteLU.jl - ILU with drop tolerance
  - AlgebraicMultigrid.jl - Multigrid methods with automatic coarsening
- LinearSolve.jl - "One-stop shop" for linear system solution

Table of Contents

Iterative methods for linear systems
- Simple iteration scheme
  - General convergence theorem
  - Convergence estimate for symmetric positive definite A,M
  - Convergence estimate for regular splittings
  - Sparse matrix generation for solver benchmarking
- Preconditioners
  - Jacobi preconditioner
  - Incomplete LU factorization
  - Further preconditioners
  - Implementation of the simple iteration scheme
- Krylov subspace methods
  - The idea of the GMRES method
  - Conjugated Gradients
  - Krylov methods in Julia
- Complexity scaling
  - LinearSolve.jl
  - Scaling test
- Final remarks