Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann
Notebook 18

```
begin
    using Pkg
    using IterativeSolvers
    using IncompleteLU
    using PlutoUI
    using PyPlot
    using DataFrames
    using LinearAlgebra
    using HypertextLiteral
    using SparseArrays

    function pyplot(f;width=3,height=3)
      clf()
      f()
      fig=gcf()
      fig.set_size_inches(width,height)
      fig
    end
end;
```

Practical iterative methods

Incomplete LU (ILU) preconditioning

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.

Discussion

- Generally better convergence properties than Jacobi, Gauss-Seidel, though we mostly cannot apply the comparison theorem for regular splittings
- 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 approaches to preconditioning

These are based on ideas which are best explained and developed with multidimensional PDEs in mind.

- Multigrid: gives indeed $O(N)$ optimal solver complexity in some situations. This is the holy grail method... I will try to discuss this later in the course.
- Domain decomposition - based on the idea the subdivision of the computational domain into a number of subdomains and subsequent repeated solution of the smaller subdomain problems

Iterative methods in Julia

Julia has some well maintained packages for iterative methods and preconditioning.

- **IterativeSolvers.jl**: various Krylov subspace methods including conjugate gradients
- **IncompleteLU.jl**: Incomplete LU factorizations
- **AlgebraicMultigrid.jl**: Algebraic multigrid methods

Random sparse M-Matrices

We will test the methods with random sparse M matrices, so we define a function which gives us a random, strictly diagonally dominant M-Matrix which is not necessarily irreducible. For `skew=0` it is also symmetric:

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

```
function sprandm(n;p=0.5,skew=0)
    A=sprand(n,n,p) # random sparse matrix with positive entries
    for i=1:n       # set diagonal to zero
      A[i,i]=0
    end
    A=A+(1.0-skew)*transpose(A) # symmetrize if necessary
    d=0.001*rand(n) # define a positive random diagonal vector
    for i=1:n # update to dominance
       d[i]+=sum(A[:,i])
    end
    Diagonal(d)-A  # create final matrix
end
```

Test the method a bit...

```
N = 5
```

```
N=5
```

```
A=sprandm(N,p=0.6,skew=1);
```

	x1	x2	x3	x4	x5
1	0.868063	0.0	0.0	0.0	-0.955656
2	-0.317544	0.000260971	-0.169388	0.0	-0.592699
3	0.0	0.0	0.218173	0.0	-0.0168421
4	-0.16788	0.0	0.0	0.000395588	0.0
5	-0.381814	0.0	-0.0480429	0.0	1.56547

```
DataFrame(A,:auto)
```

Up to rounding errors, the inverse is nonnegative, as predicted by the theory. There are zero entries because it is not necessarily irreducible. Invertibility is guaranteed by strict diagonal dominance.

```
Ainv = 5×5 Matrix{Float64}:
          1.57622      -0.0      0.21239     -0.0      0.964508
       2812.41       3831.84   3681.27        -0.0   3207.24
          0.0297475     0.0      4.5984      -0.0      0.0676316
        668.92          0.0     90.1341    2527.88   409.319
          0.385351      0.0      0.192923     0.0      0.876104
```

```
Ainv=inv(Matrix(A))
```

```
-0.0
```

```
minimum(Ainv)
```

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

```
function  ρ_jacobi(A)
    B=I(size(A,1))-inv(Diagonal(A))*A;
    maximum(abs.(eigvals(Matrix(B))))
end
```

```
0.5204590778062587
```

```
ρ_jacobi(A)
```

Preconditioners for sparse matrices

Here, we define two preconditioners which are able to work together with **IterativeSolvers.jl**.

Jacobi

```
begin
    # Data struture: 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
    function  LinearAlgebra.ldiv!(precon::JacobiPreconditioner,v)
      ldiv!(v,precon,v)
    end

end
```

We can construct a the preconditioner then as follows:

```
preconJacobi =   JacobiPreconditioner([1.15199, 3831.84, 4.58351, 2527.88, 0.638787])
```

```
preconJacobi=JacobiPreconditioner(A)
```

And solve the preconditioning system:

```
[1.15199, 3831.84, 4.58351, 2527.88, 0.638787]
```

```
ldiv!(preconJacobi,ones(N))
```

ILU0

For this preconditioner, we need to store the matrix, the inverse of a modified diagonal and the indices of the main diagonal entries in the sparse matrix columns.

```
begin
```

```
preconILU0=ILU0Preconditioner(A);
```

```
[2544.56, 3831.84, 3306.42, 2527.88, 1484.17]
```

```
ldiv!(preconILU0,ones(N))
```

Simple iteration method with interface similar to IterativeSolvers.jl

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

```
begin
```

Iterative Method comparison: symmetric problems

```
N1 = 1000
```

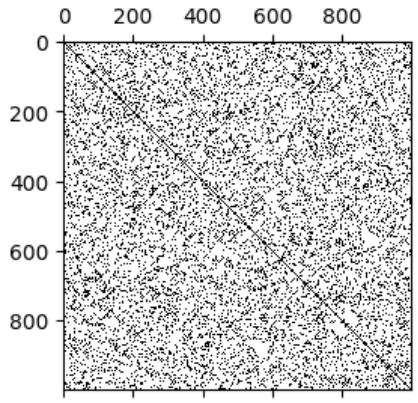
```
N1=1000
```

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

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

```
A1 = 1000×1000 SparseMatrixCSC{Float64, Int64} with 191032 stored entries:
```

```
A1=sprandm(N1,p=0.1,skew=0)
```




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

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

```
191032
```

```
nnz(A1ILU0)
```

```
935676
```

```
nnz(lu(A1))
```

Create a right hand side for testing

```
b1 =
```

```
[0.000919511, 0.000364151, 0.000792563, 0.0001801, 0.000833396, 0.000314367, 0.00097769, 0
```

```
b1=A1*ones(N1)
```

So let us run this with Jacobi preconditioner. Theory tells it should converge...

```
([0.000510771, 0.000504588, 0.000509992, 0.000503071, 0.000508997, 0.000504462, 0.00051168
```

```
sol_simple_jacobi,hist_simple_jacobi=simple(A1,b1,tol=tol,maxiter=100,log=true,Pl=A1Jacobi)
```

After 100 steps we are far from the solution, and we need lots of steps to converge, so let us have a look at the spectral radius of the iteration matrix and compare it with the residual reduction in the last iteration step:

```
0.9999949343874537
```

```
ρ_jacobi(A1)
```

```
0.9999949343874565
```

```
hist_simple_jacobi[:resnorm][end]/hist_simple_jacobi[:resnorm][end-1]
```

It seams we have found a simple spetral radius estimator here ...

Now for the ILU0 preconditioner:

```
([0.00152324, 0.00151677, 0.00152284, 0.00151561, 0.00152153, 0.00151691, 0.00152398, 0.00
```

```
sol_simple_ilu0,hist_simple_ilu0=simple(A1,b1,tol=tol,maxiter=100,log=true,Pl=A1ILU0)
```

... the spectral radius estimate is a little bit better...

```
0.9999848597498143
```

```
hist_simple_ilu0[:resnorm][end]/hist_simple_ilu0[:resnorm][end-1]
```

We have symmetric matrices, so let us try CG:

Without preconditioning:

```
([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,    more ,1.0], Converged after 18 iterations
```

```
sol_cg,hist_cg=cg(A1,b1, reltol=tol,log=true,maxiter=100)
```

With Jacobi preconditioning:

```
([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,    more ,1.0], Converged after 15 iterations
```

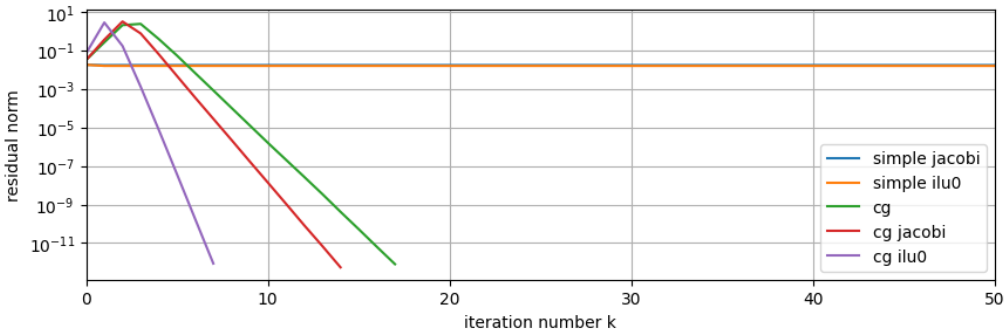
```
sol_cg_jacobi,hist_cg_jacobi=cg(A1,b1, reltol=tol,log=true,maxiter=100,Pl=A1Jacobi)
```

With ILU0 preconditioning

```
([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,    more ,1.0], Converged after 8 iterations.
```

```
sol_cg_ilu0,hist_cg_ilu0=cg(A1,b1, reltol=tol,log=true,maxiter=100,Pl=A1ILU0)
```

- As we see, all CG variants converge within the given number of iterations steps.
- Preconditioning 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



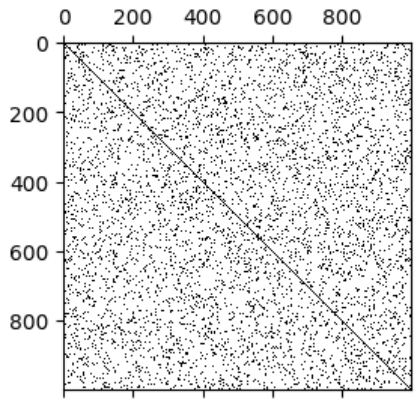
Nonsymmetric problems

Here, we skip the simple iteration and look at the performance of some Krylov subspace methods.

```
N2 = 1000
```

```
N2=1000
```

```
A2=sprandm(N2,p=0.1,skew=1);
```



```
pyplot() do
```

```
b2=A2*ones(N2);
```

```
A2Jacobi=JacobiPreconditioner(A2);
```

```
A2ILU0=ILU0Preconditioner(A2);
```

This is a ILU preconditioner with drop tolerance:

```
A2ILUT=IncompleteLU.ilu(A2,τ=1);
```

Try CG:

```
([3.22551, 2.72997, 3.82519, 3.49427, 3.09176, 2.7142, 3.05958, 3.17942, 4.47714,    more
```

```
sol2_cg,hist2_cg=cg(A2,b2, reltol=tol,log=true,maxiter=100)
```

([4.41629e5, 3.51379e5, 547723.0, 4.97462e5, 4.21232e5, 3.50886e5, 4.23256e5, 4.31918e5, 6

```
sol2_cg_jacobi,hist2_cg_jacobi=cg(A2,b2, reltol=tol,log=true,maxiter=100,Pl=A2Jacobi)
```

([0.0220723, 0.233973, -0.251647, -0.104811, 0.0714345, 0.243108, 0.092034, 0.0349546, -0.

```
sol2_cg_ILU0,hist2_cg_ILU0=cg(A2,b2, reltol=tol,log=true,maxiter=100,Pl=A2ILU0)
```

Use the `bicgstabl` method 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 7 iterations.

```
sol2_bicgstab,hist2_bicgstab=bicgstabl(A2,b2,reltol=tol,log=true,max_mv_products=100)
```

([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, more ,1.0], Converged after 5 iterations.

```
sol2_bicgstab_jacobi,hist2_bicgstab_jacobi=bicgstabl(A2,b2,reltol=tol,log=true,max_mv_
products=100,Pl=A2Jacobi)
```

([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, more ,1.0], Converged after 6 iterations.

```
sol2_bicgstab_ilu0,hist2_bicgstab_ilu0=bicgstabl(A2,b2,reltol=tol,log=true,max_mv_prod
ucts=100,Pl=A2ILU0)
```

([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, more ,1.0], Converged after 5 iterations.

```
sol2_bicgstab_ilut,hist2_bicgstab_ilut=bicgstabl(A2,b2,reltol=tol,log=true,max_mv_prod
ucts=100,Pl=A2ILUT)
```

([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, more ,1.0], Converged after 24 iterations

```
sol2_gmres,hist2_gmres=gmres(A2,b2,reltol=tol,log=true,maxiter=100)
```

([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, more ,1.0], Converged after 17 iterations

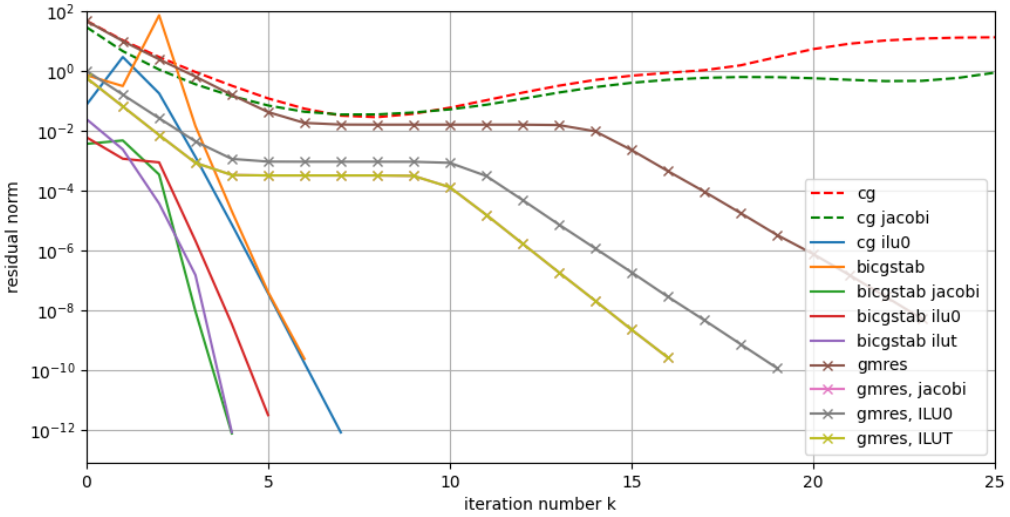
```
sol2_gmres_jacobi,hist2_gmres_jacobi=gmres(A2,b2,Pl=A2Jacobi,reltol=tol,log=true,maxit
er=100)
```

([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, more ,1.0], Converged after 20 iterations

```
sol2_gmres_ilu0,hist2_gmres_ilu0=gmres(A2,b2,Pl=A2ILU0,reltol=tol,log=true,maxiter=100
)
```

([1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, more ,1.0], Converged after 17 iterations

```
sol2_gmres_ilut,hist2_gmres_ilut=gmres(A2,b2,Pl=A2ILUT,reltol=tol,log=true,maxiter=100
)
```



- CG may not converge - the case is also not covered by the theory
- Various preconditioners improve the convergence
- In this case, GMRES is generally slower than BiCGstab (and each iteration is more expensive)
- Generally, finding the optimal approach for nosymmetric systems is an trial-and-error porcess: try Jacobi, ILU, . . ., try GMRES, BiCGstab, CGS . . ., vary parameters
- We well make another comprison with systems from PDEs later.

```
begin
```