pyplot (generic function with 1 method)

```
begin
    using Pkg
    Pkg.activate(mktempdir())
    Pkg.add("PyPlot")
    Pkg.add("PlutoUI")
    Pkg.add("IterativeSolvers")
    Pkg.add("IncompleteLU")
    Pkg.add("DataFrames")

    using IterativeSolvers
    using IncompleteLU
    using PlutoUI
    using PyPlot
    using DataFrames

    using LinearAlgebra
    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 those
- 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. 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
- 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	2.18746	-0.814626	-0.452134	-0.704067	0.0
2	-0.494782	2.59544	-0.183471	0.0	-0.288585
3	0.0	-0.795573	1.17374	-0.228419	-0.0102942
4	-0.993348	-0.985134	0.0	0.933306	0.0
5	-0.698695	0.0	-0.537345	0.0	0.299452

```
DataFrame(A)
```

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 Array{Float64,2}:
        199.183  198.919  198.709  198.892  198.531
        134.768  135.099  134.757  134.647  134.829
        166.991  167.226  167.737  167.027  166.924
```

```
354.249  354.317  353.733  354.883  353.62
764.396  764.203  764.629  763.781  766.096
```

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

```
134.64717151847654
```

```
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
```

```
0.9993509631121599
```

```
ρ_jacobi(A)
```

Preconditioners

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(Float64[0.457152,  0.385291,  0.851977,  1.07146,  3.33943])
```

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

```
Float64[0.457152,  0.385291,  0.851977,  1.07146,  3.33943]
```

```
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

    struct ILU0Preconditioner
    A::AbstractMatrix
    xdiag::Vector
    idiag::Vector
    end
```

```
    function ILU0Preconditioner(A::AbstractMatrix)
    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 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

    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)

    # forward substitution
        for j=1:n
            x=zero(T)
            for k=colptr[j]:idiag[j]-1
                x+=nzval[k]*u[rowval[k]]
            end
            u[j]=xdiag[j]*(v[j]-x)
        end

    # backward substitution
        for j=n:-1:1
        x=zero(T)
        for k=idiag[j]+1:colptr[j+1]-1
            x+=u[rowval[k]]*nzval[k]
        end
            u[j]-=x*xdiag[j]
        end
    u
    end

    function  LinearAlgebra.ldiv!(precon::ILU0Preconditioner,v)
    ldiv!(v,precon,v)
    end

    SparseArrays.nnz(precon::ILU0Preconditioner)=nnz(precon.A)

end
```

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

```
Float64[2.93032,  2.05551,  2.89032,  1.68817,  3.88721]
```

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

Simple iteration method with interface similar to IterativeSolvers.jl

```
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
```

Iterative Method comparison: symmetric problems

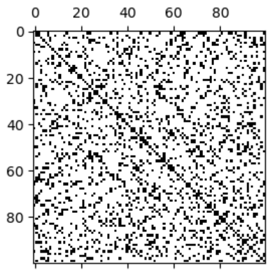
N1 = 100

```
N1=100
```

tol = 1.0e-10

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

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




```
pyplot() do
    spy(A1)
end
```

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

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

Create also ILU preconditioners from IncompleteLU.jl: These have drop tolerance τ as parameter. The larger τ, the more entries of the LU factors are ignored.

```
A1ILUT_1=IncompleteLU.ilu(A1,τ=0.15);
```

```
A1ILUT_2=IncompleteLU.ilu(A1,τ=0.05);
```

2032

```
nnz(A1ILU0)
```

2310

```
nnz(A1ILUT_1)
```

4860

```
nnz(A1ILUT_2)
```

6680

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

Create a right hand side for testing

```
b1 =
 Float64[0.000965422, 0.000284515, 9.91317e-5, 0.000509845, 0.000271775, 0.000599232,
```

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

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

```
(Float64[0.00508829, 0.0049741, 0.00498199, 0.00499919, 0.00498366, 0.00502427, 0.005
```

```
sol_simple_jacobi,hist_simple_jacobi=simple(A1,b1,tol=tol,maxiter=100,log=true,Pl=A1J
acobi)
```

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:

```
(0.99995, 0.99995)
```

```
ρ_jacobi(A1),(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:

```
(Float64[0.0146098, 0.0144986, 0.0144998, 0.0145256, 0.0145013, 0.0145409, 0.0145176,
```

```
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...

```
0.9998541700844135
```

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

We have symmetric matrices, so let us try CG:

Without preconditioning:

```
(Float64[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, :
```

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

With Jacobi preconditioning:

```
(Float64[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, :
```

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

With various variants of ILU preconditioners:

```
(Float64[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, :
```

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

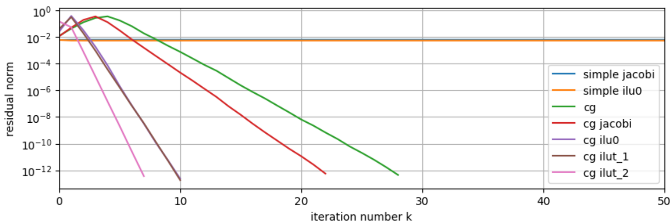
```
(Float64[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, :
```

```
sol_cg_ilut_1,hist_cg_ilut_1=cg(A1,b1, reltol=tol,log=true,maxiter=100,Pl=A1ILUT_1)
```

(Float64[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,

```
sol_cg_ilut_2,hist_cg_ilut_2=cg(A1,b1, reltol=tol,log=true,maxiter=100,Pl=A1ILUT_2)
```

- 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




```
pyplot(width=10) do
    semilogy(hist_simple_ilu0[:resnorm],label="simple jacobi")
    semilogy(hist_simple_jacobi[:resnorm],label="simple ilu0")
    semilogy(hist_cg[:resnorm],label="cg")
    semilogy(hist_cg_jacobi[:resnorm],label="cg jacobi")
    semilogy(hist_cg_ilu0[:resnorm],label="cg ilu0")
    semilogy(hist_cg_ilut_1[:resnorm],label="cg ilut_1")
    semilogy(hist_cg_ilut_2[:resnorm],label="cg ilut_2")
    xlim(0,50)
    xlabel("iteration number k")
    ylabel("residual norm")
    legend(loc="lower right")
    grid()
end
```

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);
```

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

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

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

```
A2ILUT=IncompleteLU.ilu(A2,τ=0.1);
```

Try CG:

(Float64[1.68069, 1.483, 1.50429, 1.65334, 1.70164, 1.62845, 1.50726, 1.67786, 1.65

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

(Float64[632.834, 446.586, 462.752, 595.862, 642.002, 574.132, 460.705, 619.735, 59

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

```
(Float64[-0.206584, 0.140421, 0.110268, -0.156276, -0.247687, -0.114002, 0.103738, -
```

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

Use the `bicgstabl` method from IterativeSolvers.jl:

```
md"""
Use the `bicgstabl` method from IterativeSolvers.jl:
"""
```

```
(Float64[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
```

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

```
(Float64[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
```

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

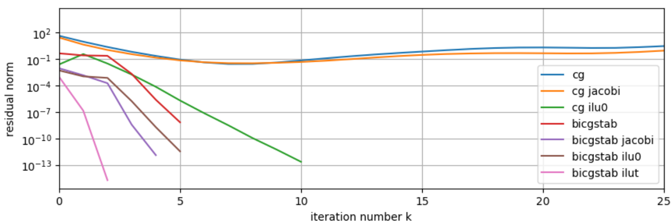
```
(Float64[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
```

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

```
(Float64[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
```

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

- CG does not converge - the case is also not covered by the theory
- Various preconditioners improve the convergence
- Is there a bug in the implementation of my `ILU0` ?




```
pyplot(width=10) do
    semilogy(hist2_cg[:resnorm],label="cg")
    semilogy(hist2_cg_jacobi[:resnorm],label="cg jacobi")
    semilogy(hist_cg_ilu0[:resnorm],label="cg ilu0")
    semilogy(hist2_bicgstab[:resnorm],label="bicgstab")
    semilogy(hist2_bicgstab_jacobi[:resnorm],label="bicgstab jacobi")
    semilogy(hist2_bicgstab_ilu0[:resnorm],label="bicgstab ilu0")
    semilogy(hist2_bicgstab_ilut[:resnorm],label="bicgstab ilut")
    xlim(0,25)
    xlabel("iteration number k")
    ylabel("residual norm")
    legend(loc="lower right")
    grid()
end
```