Numerical Linear Algebra for Computational Sciences and Information Engineering
CITHN2006

Preconditioned Iterative Methods for Linear Systems

by Nicolas Venkovic

Chair of Computational Mathematics
School of Computation, Information and Technology (CIT)
Technical University of Munich, Germany

[1]:
```
using LinearAlgebra, Plots, Printf, Latexify, LaTeXStrings, BenchmarkTools,␣
↪SparseArrays, IncompleteLU
using MatrixMarket: mmread
```

1. *Implement the PCG method.*

[2]:
```
function pcg(A, b::Vector{Float64}, x::Array{Float64}, M, m::Int;
             tol::Float64=1e-6,
             verbose=true)

  # In
  # A          : operator with mul! defined
  # b          : right-hand side
  # x          : initial iterate
  # M          : preconditioner with z = M \ r defined
  # m          : maximum number of iterations
  # tol        : tolerance for convergence criterion
  # verbose    : if true, prints backward error
  # debug      : if true, computes and prints residuals

  # Out
  # x              : last iterate
  # backward_error : vector of iterated backward errors

  n, = size(x)
  r = Vector{Float64}(undef, n)
  z = Vector{Float64}(undef, n)
  p = Vector{Float64}(undef, n)
  Ap = Vector{Float64}(undef, n)
```

```
  backward_error = Vector{Float64}(undef, m+1)

  r .= b .- A * x
  rTr = dot(r, r)
  z .= M \ r
  rTz = dot(r, z)
  copyto!(p, z)
  bnorm = norm(b)
  backward_error[1] = sqrt(rTr) / bnorm
  if verbose
    println("pcg it = 0, backward_error_0 = $(backward_error[1])")
  end

  j = 0
  while (j < m) && (backward_error[j+1] > tol)
    j += 1
    mul!(Ap, A, p) # Ap = A * p
    d = dot(p, Ap)
    alpha = rTz / d
    beta = 1. / rTz
    axpy!(alpha, p, x) # x += alpha * p
    axpy!(-alpha, Ap, r) # r -= alpha * Ap
    rTr = dot(r, r)
    z .= M \ r
    rTz = dot(r, z)
    beta *= rTz
    axpby!(1, z, beta, p) # p = beta * p + z
    backward_error[j+1] = sqrt(rTr) / bnorm
    if verbose
      println("pcg it = $j, backward_error_$j = $(backward_error[j+1])")
    end
  end

  return x, backward_error[1:j+1]
end;
```

[3]:
```
A = mmread("matrices/bcsstm12.mtx");
n, _ = size(A);
b = rand(n);
```

[4]:
```
M = spdiagm(diag(A));
```

[5]:
```
x = zeros(n);
m = 5000;
pcg(A, b, x, M, m);
```

```
pcg it = 0, backward_error_0 = 1.0
pcg it = 1, backward_error_1 = 1.3857326633930065
```

```
pcg it = 2, backward_error_2 = 2.0266061153948054
pcg it = 3, backward_error_3 = 3.3235332851156008
pcg it = 4, backward_error_4 = 5.574905282917839
pcg it = 5, backward_error_5 = 5.657823663920716
pcg it = 6, backward_error_6 = 4.966225441008765
pcg it = 7, backward_error_7 = 4.678507394661282
pcg it = 8, backward_error_8 = 4.690058565530404
pcg it = 9, backward_error_9 = 5.227307992553279
… (354 lines omitted) …
pcg it = 364, backward_error_364 = 2.395980129541342e-6
pcg it = 365, backward_error_365 = 1.7356290806850178e-6
pcg it = 366, backward_error_366 = 1.358117771581629e-6
pcg it = 367, backward_error_367 = 1.2042272589257526e-6
pcg it = 368, backward_error_368 = 1.9640410390706286e-6
pcg it = 369, backward_error_369 = 2.312316027385653e-6
pcg it = 370, backward_error_370 = 2.692664341683532e-6
pcg it = 371, backward_error_371 = 2.0487269238608103e-6
pcg it = 372, backward_error_372 = 1.5542642175098471e-6
pcg it = 373, backward_error_373 = 9.933614988194455e-7
```

2. *Implemement the split-preconditioned GMRES method.*

[6]:
```
function gmres(A, b::Vector{Float64}, x::Vector{Float64}, ML, MR, m::Int;
               tol::Float64=1e-6,
               verbose=true,
               debug::Bool=false)

  # In
  # A         : operator with mul! defined
  # b         : right-hand side
  # x         : initial iterate
  # ML        : left-preconditioner  with z = M \ r defined
  # MR        : right-preconditioner with z = M \ r defined
  # m         : maximum number of iterations
  # tol       : tolerance for convergence criterion
  # verbose   : if true, prints backward error
  # debug     : if true, computes and prints left-preconditioned residuals

  # Out
  # x              : last iterate
  # backward_error : vector of iterated backward errors

  n = length(x)
  z = Vector{Float64}(undef, n)                    # left-preconditioned␣
↪residual
  V = Array{Float64,2}(undef, n, m + 1)            # Arnoldi basis of ML^-1␣
↪* A * MR^-1
  H = zeros(Float64, m+1, m)                       # Hessenberg matrix
```

```
 v = Vector{Float64}(undef, n)                          # temporary vector #1
 w = Vector{Float64}(undef, n)                          # temporary vector #2
 d = Vector{Float64}(undef, m)                          # solution of reduced LSQ␣
↪problem
 vec = Vector{Float64}(undef, m+1)                      # z = V * vec
 vec0 = zeros(Float64, m+1)                             #
 backward_error = Vector{Float64}(undef, m+1)           # backward_error[j]  ␣
↪norm(z) / norm(ML * b)

 copy!(z, b) # z = b
 mul!(z, A, x, -1, 1) # z = z - A * x
 #ldiv!(ML, z) # z = ML \ z, i.e., z = ML \ (b - A * x)
 z .= ML \ z
 beta = norm(z)
 vec0[1] = beta
 axpby!(1 / beta, z, 0, v) #  v = z / beta
 V[:, 1] = v
 bnrm2 = norm(ML \ b)
 backward_error[1] = beta / bnrm2

 j = 0
 while j < m
   j += 1
   #ldiv!(MR, v) # v = MR \ v#
   v .= MR \ v
   mul!(w, A, v) # w = A * v
   #ldiv!(ML, w) # w = ML \ w
   w .= ML \ w
   for i in 1:j
     H[i, j] = dot(V[:, i], w)
     axpy!(-H[i, j], V[:, i], w)
   end
   H[j+1, j] = norm(w)
   axpby!(1 / H[j+1, j], w, 0, v) # v = w ./ H[j+1, j]
   V[:, j+1] = v
   d[1:j] = H[1:j+1, 1:j] \ vec0[1:j+1] # solve LSQ problem
   vec[1:j+1] = vec0[1:j+1] .- H[1:j+1, 1:j] * d[1:j]  # z = V[: 1:j+1] *␣
↪vec[1:j+1]
   backward_error[j+1] = norm(vec[1:j+1]) / bnrm2 # norm(z) = norm(vec[1:j+1])
   println("backward_error_$j = $(backward_error[j+1])")
   if debug
     xj = x + MR \ (V[:, 1:j] * d[1:j])
     println("                  ", " "^floor(Int, log10(j) + 1), norm(ML \ (b␣
↪- A * xj)) / bnrm2)
   end
   if (backward_error[j+1] < tol)
     m = j
```

```
      break
    end
  end

  mul!(w, V[:, 1:m], d[1:m]) # w = V[:, 1:m] * d[1:m]
  #ldiv!(MR, w) # w = MR \ w
  w .= MR \ w
  #axpby!(1, w, 1, x) # x += w, i.e., x = x + MR \ (V[:, 1:m] * d[1:m])
  x += w

  return x, backward_error[1:j+1]
end;
```

[7]:
```
A = mmread("matrices/G3_circuit.mtx");
n, _ = size(A);
b = rand(n);
```

[8]:
```
ML = spdiagm(diag(A));
```

[9]:
```
MR = ilu(A);
```

[10]:
```
x = zeros(n);
m = 5000;
gmres(A, b, x, ML, MR, m, tol=1e-2, debug=true);
```

```
backward_error_1 = 0.4408339460525594
                   0.44083394605255927
backward_error_2 = 0.30103254767530974
                   0.30103254767530935
backward_error_3 = 0.187908166306327
                   0.18790816630632612
backward_error_4 = 0.1330396395236413
                   0.13303963952364045
backward_error_5 = 0.07280073446719472
                   0.07280073446719414
… (40 lines omitted) …
backward_error_26 = 0.014307288588569894
                    0.014307288588569054
backward_error_27 = 0.013619409672447125
                    0.01361940967244601
backward_error_28 = 0.011635550967458365
                    0.011635550967457255
backward_error_29 = 0.01024595205173779
                    0.010245952051737487
backward_error_30 = 0.00993740284715508
                    0.009937402847154367
```

3. *Implement the right-preconditioned BiCGSTAB method.*

[11]:
```
function bicgstab(A, b::Vector{Float64}, x::Vector{Float64}, MR, m::Int;
                  tol::Float64=1e-6,
                  verbose=true)

  # In
  # A          : operator with mul! defined
  # b          : right-hand side
  # x          : initial iterate
  # MR         : right-preconditioner with z = M \ r defined
  # m          : maximum number of iterations
  # tol        : tolerance for convergence criterion
  # verbose    : if true, prints backward error

  # Out
  # x              : last iterate
  # backward_error : vector of iterated backward errors

  n = A.n
  p  = Vector{Float64}(undef, n)
  q  = Vector{Float64}(undef, n)
  r  = Vector{Float64}(undef, n)
  r0 = Vector{Float64}(undef, n)
  Ap = Vector{Float64}(undef, n)
  Aq = Vector{Float64}(undef, n)
  w  = Vector{Float64}(undef, n)
  backward_error = zeros(m+1)

  copy!(r0, b)
  ldiv!(w, MR, x)
  mul!(r0, A, w, -1, 1) # r0 .= b - A * x (MR \ x)
  copy!(r, r0) # r .= r0
  copy!(p, r) # p .= r
  bnrm2 = norm(b)
  backward_error[1] = norm(r) / bnrm2
  rtr0 = dot(r, r0)

  j = 0
  while (j < m) && (backward_error[j + 1] > tol)
    j += 1
    w .= MR \ p
    mul!(Ap, A, w) # Ap .= A * (MR \ p)
    ptAtr0 = dot(Ap, r0)
    alpha = rtr0 / ptAtr0
    copy!(q, r) # q .= r
    axpy!(-alpha, Ap, q) # q .-= alpha * Ap
    w .= MR \ q
    mul!(Aq, A, w) # Aq .= A * (MR \ q)
```

```
    qtAq = dot(q, Aq)
    qtAtAq = dot(Aq, Aq)
    omega = qtAq / qtAtAq
    axpy!(alpha, p, x) # x .+= alpha * p
    axpy!(omega, q, x) # x .+= omega * q
    copy!(r, q) # r .= q
    axpy!(-omega, Aq, r) # r .-= omega * Aq
    new_rtr0 = dot(r, r0)
    backward_error[j + 1] = norm(r) / bnrm2
    if verbose
      println("bicgstab it = $j, backward error = $(backward_error[j + 1])")
    end
    beta = (alpha / omega) * (new_rtr0 / rtr0)
    rtr0 = new_rtr0
    axpby!(1, r, beta, p) # p .= r + beta * p
    axpy!(-beta * omega, Ap, p) # p .-= beta * omega * Ap
  end

  x .= MR \ x
  return x, backward_error[1:j+1]
end;
```

[12]:
```
x = zeros(n);
m = 5000;
bicgstab(A, b, x, MR, m, tol=1e-2);
```

```
bicgstab it = 1, backward error = 0.6569850656016885
bicgstab it = 2, backward error = 0.10948728187687652
bicgstab it = 3, backward error = 0.028363224107138665
bicgstab it = 4, backward error = 0.01388512348307391
bicgstab it = 5, backward error = 0.007707896554901543
```