Numerical Linear Algebra for Computational Sciences and Information Engineering
CITHN2006

Krylov Subspace 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
using MatrixMarket: mmread
```

1. *Implement and test the FOM method based on MGS and CGS2 orthogonalization.*

[2]:
```
function fom(A, b::Vector{Float64}, x::Vector{Float64}, m::Int;
             ortho=:MGS,
             tol::Float64=1e-6,
             verbose=true,
             debug::Bool=false)

  # In
  # A         : operator with mul! defined
  # b         : right-hand side
  # x         : initial iterate
  # 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 = length(x)
  r = Vector{Float64}(undef, n)                    # residual
  V = Array{Float64,2}(undef, n, m + 1)            # Arnoldi basis
  H = zeros(Float64, m+1, m)                       # Hessenberg matrix
  v = Vector{Float64}(undef, n)                    # temporary vector
```

```
 w = Vector{Float64}(undef, n)                          # temporary vector
 d = Vector{Float64}(undef, m)                          # solution of reduced␣
↪linear system
 vec = zeros(Float64, m + 1)                            # temporary vector
 vec0 = zeros(Float64, m + 1)                           # beta * e1
 backward_error = Vector{Float64}(undef, m + 1)         # backward_error[j] ␣
↪norm(r) / norm(b)

 copy!(r, b) # r = b
 mul!(r, A, x, -1, 1) # r = r - A * x = b - A * x
 beta = norm(r)
 vec0[1] = beta
 axpby!(1 / beta, r, 0, v) #  v = r / beta
 copyto!(view(V, :, 1), v)
 bnrm2 = norm(b)
 backward_error[1] = beta / bnrm2
 if verbose
   println("fom it = 0, backward_error_0 = $(backward_error[1])")
 end

 j = 0
 while j < m
   j += 1
   mul!(w, A, v) # w = A * v
   if ortho == :MGS
     for i in 1:j
       H[i, j] = dot(view(V, :, i), w)
       axpy!(-H[i, j], view(V, :, i), w)
     end
   elseif ortho == :CGS2
     mul!(view(vec, 1:j), view(V, :, 1:j)', w)
     mul!(w, view(V, :, 1:j), view(vec, 1:j), -1, 1)
     mul!(view(H, 1:j, j), view(V, :, 1:j)', w)
     mul!(w, view(V, :, 1:j), view(H, 1:j, j), -1, 1)
     H[1:j, j] += view(vec, 1:j)
   end
   H[j+1, j] = norm(w)
   axpby!(1 / H[j+1, j], w, 0, v) # v = w ./ H[j+1, j]
   copyto!(view(V, :, j+1), v)
   d[1:j] = view(H, 1:j, 1:j) \ view(vec0, 1:j) # solve reduced linear system

   backward_error[j+1] = H[j+1, j] * norm(d[j]) / bnrm2 # norm(r)
   if verbose
     println("fom it = $j, backward_error_$j = $(backward_error[j+1])")
   end
   if debug
     xj = x + view(V, :, 1:j) * view(d, 1:j)
```

```
      println("                                ", " "^floor(Int, log10(j) + 1),␣
↪norm(b - A * xj) / bnrm2)
    end
    if (backward_error[j+1] < tol)
      m = j
      break
    end
  end

  mul!(w, view(V, :, 1:m), view(d, 1:m)) # w = V[:, 1:m] * d[1:m]
  x .+= w

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

[3]:
```
A = mmread("matrices/cage3.mtx");
n, _ = size(A);
b = rand(n);
```

[4]:
```
x = zeros(n);
m = 9;
fom(A, b, x, m, ortho=:MGS, debug=true);
```

```
fom it = 0, backward_error_0 = 1.0
fom it = 1, backward_error_1 = 0.9306361105673551
                               0.9306361105673551
fom it = 2, backward_error_2 = 0.23082416190174393
                               0.23082416190174393
fom it = 3, backward_error_3 = 0.2378855264436731
                               0.2378855264436731
fom it = 4, backward_error_4 = 9.835756138174568e-16
                               1.0642400956964984e-15
```

[5]:
```
x = zeros(n);
m = 9;
fom(A, b, x, m, ortho=:CGS2, debug=true);
```

```
fom it = 0, backward_error_0 = 1.0
fom it = 1, backward_error_1 = 0.9306361105673547
                               0.9306361105673547
fom it = 2, backward_error_2 = 0.2308241619017438
                               0.23082416190174376
fom it = 3, backward_error_3 = 0.2378855264436729
                               0.23788552644367286
fom it = 4, backward_error_4 = 8.092357637000868e-16
                               1.2466512141893224e-15
```

2. *Implement, test and benchmark the GMRES method based on MGS and CGS2 orthogonalization.*

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

  # In
  # A         : operator with mul! defined
  # b         : right-hand side
  # x         : initial iterate
  # 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 = length(x)
  r = Vector{Float64}(undef, n)                    # residual
  V = Array{Float64,2}(undef, n, m + 1)            # Arnoldi basis
  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
  vec0 = zeros(Float64, m + 1)                     # beta * e1
  vec = Vector{Float64}(undef, m + 1)              # z = V * vec
  backward_error = Vector{Float64}(undef, m + 1)   # backward_error[j] ␣
↪norm(r) / norm(b)

  copy!(r, b) # r = b
  mul!(r, A, x, -1, 1) # r = r - A * x = b - A * x
  beta = norm(r)
  vec0[1] = beta
  axpby!(1 / beta, r, 0, v) #  v = r / beta
  copyto!(view(V, :, 1), v)
  bnrm2 = norm(b)
  backward_error[1] = beta / bnrm2
  if verbose
    println("gmres it = 0, backward_error_0 = $(backward_error[1])")
  end

  j = 0
  while j < m
    j += 1
```

```
    mul!(w, A, v) # w = A * v
    if ortho == :MGS
      for i in 1:j
        H[i, j] = dot(view(V, :, i), w)
        axpy!(-H[i, j], view(V, :, i), w)
      end
    elseif ortho == :CGS2
      mul!(view(vec, 1:j), view(V, :, 1:j)', w)
      mul!(w, view(V, :, 1:j), view(vec, 1:j), -1, 1)
      mul!(view(H, 1:j, j), view(V, :, 1:j)', w)
      mul!(w, view(V, :, 1:j), view(H, 1:j, j), -1, 1)
      H[1:j, j] += view(vec, 1:j)
    end
    H[j+1, j] = norm(w)
    axpby!(1 / H[j+1, j], w, 0, v) # v = w ./ H[j+1, j]
    copyto!(view(V, :, j+1), v)
    d[1:j] = view(H, 1:j+1, 1:j) \ view(vec0, 1:j+1) # solve LSQ problem
    vec[1:j+1] = view(vec0, 1:j+1) .- view(H, 1:j+1, 1:j) * view(d, 1:j)  # r =␣
↪V[: 1:j+1] * vec[1:j+1]
    backward_error[j+1] = norm(vec[1:j+1]) / bnrm2 # norm(r) = norm(vec[1:j+1])
    if verbose
      println("gmres it = $j, backward_error_$j = $(backward_error[j+1])")
    end
    if debug
      xj = x + view(V, :, 1:j) * view(d, 1:j)
      println("                                    ", " "^floor(Int, log10(j) + 1),␣
↪norm(b - A * xj) / bnrm2)
    end
    if (backward_error[j+1] < tol)
      m = j
      break
    end
  end

  mul!(w, view(V, :, 1:m), view(d, 1:m)) # w = V[:, 1:m] * d[1:m]
  x .+= w

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

[7]:
```
x = zeros(n);
m = 9;
gmres(A, b, x, m, ortho=:MGS, debug=true);
```

```
gmres it = 0, backward_error_0 = 1.0
gmres it = 1, backward_error_1 = 0.6812622984478455
                                 0.6812622984478455
gmres it = 2, backward_error_2 = 0.21861663364080008
```

```
                                    0.2186166336408
gmres it = 3, backward_error_3 = 0.16096699694207078
                                    0.1609669969420707
gmres it = 4, backward_error_4 = 1.1504954107678343e-15
                                    1.3127189961060117e-15
```

[8]:
```
x = zeros(n);
m = 9;
gmres(A, b, x, m, ortho=:CGS2, debug=true);
```

```
gmres it = 0, backward_error_0 = 1.0
gmres it = 1, backward_error_1 = 0.6812622984478454
                                    0.6812622984478455
gmres it = 2, backward_error_2 = 0.21861663364079995
                                    0.21861663364080003
gmres it = 3, backward_error_3 = 0.16096699694207073
                                    0.16096699694207073
gmres it = 4, backward_error_4 = 9.721701481930212e-16
                                    1.2777960174074395e-15
```

[9]:
```
A = mmread("matrices/G3_circuit.mtx");
n, _ = size(A);
b = rand(n);
```

[10]:
```
x = zeros(n);
m = 50;
@btime gmres($A, $b, $x, m, ortho=:MGS, verbose=false);
```

```
  2.629 s (1404 allocations: 655.63 MiB)
```

[11]:
```
x = zeros(n);
m = 50;
@btime gmres($A, $b, $x, m, ortho=:CGS2, verbose=false);
```

```
  2.156 s (1604 allocations: 655.66 MiB)
```

[12]:
```
x = zeros(n);
m = 100;
@btime gmres($A, $b, $x, m, ortho=:MGS, verbose=false);
```

```
  8.233 s (2804 allocations: 1.24 GiB)
```

[13]:
```
x = zeros(n);
m = 100;
@btime gmres($A, $b, $x, m, ortho=:CGS2, verbose=false);
```

```
  8.265 s (3204 allocations: 1.24 GiB)
```

Note that even though CGS2 requires twice the number of FLOPS as MGS does, the CGS2-based implementation remains competitive with the MGS-based implementation due to achieving a higher arithmetic intensity.

3. *Implement and test the CG algorithm.*

[14]:
```
function cg(A, b::Vector{Float64}, x::Array{Float64}, m::Int;
            tol::Float64=1e-6,
            verbose=true)

  # In
  # A          : operator with mul! defined
  # b          : right-hand side
  # x          : initial iterate
  # 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, = size(x)
  r = Vector{Float64}(undef, n)
  p = Vector{Float64}(undef, n)
  Ap = Vector{Float64}(undef, n)
  backward_error = Vector{Float64}(undef, m+1)

  copy!(r, b) # r = b
  mul!(r, A, x, -1, 1) # r = r - A * x = b - A * x
  rTr = dot(r, r)
  copyto!(p, r)
  bnorm = norm(b)
  backward_error[1] = sqrt(rTr) / bnorm
  if verbose
    println("cg 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 = rTr / d
    beta = 1. / rTr
    axpy!(alpha, p, x) # x += alpha * p
    axpy!(-alpha, Ap, r) # r -= alpha * Ap
    rTr = dot(r, r)
    beta *= rTr
    axpby!(1, r, beta, p) # p = beta * p + r
    backward_error[j+1] = sqrt(rTr) / bnorm
```

```
    if verbose
      println("cg it = $j, backward_error_$j = $(backward_error[j+1])")
    end
  end

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

[15]:
```
A = mmread("matrices/bcsstm12.mtx");
n, _ = size(A);
b = rand(n);
```

[16]:
```
x = zeros(n);
m = 5000;
cg(A, b, x, m);
```

```
cg it = 0, backward_error_0 = 0.9999999999999999
cg it = 1, backward_error_1 = 2.809777258666116
cg it = 2, backward_error_2 = 1.7668169553602087
cg it = 3, backward_error_3 = 1.9318006697217833
cg it = 4, backward_error_4 = 1.4048762714334473
cg it = 5, backward_error_5 = 1.5135091586263296
cg it = 6, backward_error_6 = 2.494526113159993
cg it = 7, backward_error_7 = 3.5198344488170767
cg it = 8, backward_error_8 = 2.751257251096199
cg it = 9, backward_error_9 = 2.3656241710902894
… (2958 lines omitted) …
cg it = 2968, backward_error_2968 = 1.7654065715306709e-6
cg it = 2969, backward_error_2969 = 2.607488975113951e-6
cg it = 2970, backward_error_2970 = 3.1728999428301996e-6
cg it = 2971, backward_error_2971 = 1.9042502016699563e-6
cg it = 2972, backward_error_2972 = 1.0365019545008502e-6
cg it = 2973, backward_error_2973 = 1.1204494351768702e-6
cg it = 2974, backward_error_2974 = 1.8345004669762876e-6
cg it = 2975, backward_error_2975 = 1.702577908838141e-6
cg it = 2976, backward_error_2976 = 1.044749666149192e-6
cg it = 2977, backward_error_2977 = 8.751728662108027e-7
```

4. *Implement and test the BiCGSTAB method.*

[17]:
```
function bicgstab(A, b::Vector{Float64}, x::Vector{Float64}, m::Int;
                  tol::Float64=1e-6,
                  verbose=true)

  # In
  # A         : operator with mul! defined
  # b         : right-hand side
  # x         : initial iterate
```

```
# 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)
copy!(r0, b)
mul!(r0, A, x, -1, 1) # r0 .= b - A * x
copy!(r, r0) # r .= r0
copy!(p, r) # p .= r
bnrm2 = norm(b)
backward_error = zeros(m + 1)
backward_error[1] = norm(r) / bnrm2
if verbose
  println("bicgstab it = 0, backward_error_0 = $(backward_error[1])")
end

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

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

[18]:
```
A = mmread("matrices/cage3.mtx");
n, _ = size(A);
b = rand(n);
```

[19]:
```
x = zeros(n);
m = 5000;
bicgstab(A, b, x, m);
```

```
bicgstab it = 0, backward_error_0 = 1.0
bicgstab it = 1, backward_error_1 = 0.18298807306466225
bicgstab it = 2, backward_error_2 = 0.028930678058760786
bicgstab it = 3, backward_error_3 = 0.012889342550749638
bicgstab it = 4, backward_error_4 = 6.661525774511565e-17
```