Numerical Linear Algebra for Computational Sciences and Information Engineering
CITHN2006

Arnoldi and Lanczos Procedures

by Nicolas Venkovic

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

[1]:
```
using LinearAlgebra, Printf, ArnoldiMethod, SparseArrays
using MatrixMarket: mmread
```

1. *Implement MGS and CGS2-based Arnoldi procedures.*

[2]:
```
function MGS_arnoldi(A, q1::Vector{Float64}, k::Int)
  n, _ = size(A)
  Q = zeros(Float64, (n, k + 1))
  H = zeros(Float64, (k + 1, k))
  w = zeros(Float64,n)
  Q[:, 1] = q1 ./ norm(q1)
  for j in 1:k
    w .= A * view(Q, :, j)
    for i in 1:j
      H[i, j] = view(Q, :, i)'w
      w -= H[i, j] .* view(Q, :, i)
    end
    H[j + 1, j] = norm(w)
    Q[:, j + 1] = w ./ H[j + 1, j]
  end
  return Q, H
end;
```

[3]:
```
function CGS2_arnoldi(A, q1::Vector{Float64}, k::Int)
  n, _ = size(A)
  Q = zeros(Float64, (n, k + 1))
  H = zeros(Float64, (k + 1, k))
  w = zeros(Float64, n)
  Q[:, 1] = q1 ./ norm(q1)
```

```
  for j in 1:k
    w .= A * view(Q, :, j)
    H[1:j, j] = view(Q, :, 1:j)'w
    w -= view(Q, :, 1:j) * view(H, 1:j, j)
    h = view(Q, :, 1:j)'w
    w -= view(Q, :, 1:j) * h
    H[j + 1, j] = norm(w)
    Q[:, j + 1] = w ./ H[j + 1, j]
  end
  return Q, H
end;
```

2. *For both methods, check how close $Q_{k+1}^T A Q_k$ is to your Hessenberg matrix, and with what accuracy does your Arnoldi relation $AQ_k = Q_{k+1}\underline{H}_k$ hold. Quantify also the loss of orthgonality of the basis, and compare runtimes. For all this, use the matrix* ***G3 circuit*** *from the SuiteSparse collection.*

[4]:
```
A = mmread("matrices/G3_circuit.mtx");
n, _ = size(A);
```

[5]:
```
k = 100;
```

[6]:
```
Q, H = @time MGS_arnoldi(A, rand(n), k);
@printf "||Qjp1 * A * Qj - H|| = %.2E, ||A * Qj - Qjp1 * H|| = %.2E, ||I -␣
 ↪Q'Q|| = %.2E" norm(Q'A * Q[:, 1:k] - H) norm(A * Q[:, 1:k] - Q * H) norm(I -␣
 ↪Q'Q)
```

```
 46.902039 seconds (1.98 M allocations: 123.005 GiB, 11.88% gc time, 1.87%
compilation time)
||Qjp1 * A * Qj - H|| = 3.83E-07, ||A * Qj - Qjp1 * H|| = 5.49E-11, ||I - Q'Q||
= 2.52E-11
```

[7]:
```
Q, H = @time CGS2_arnoldi(A, rand(n), k);
@printf "||Qjp1 * A * Qj - H|| = %.2E, ||A * Qj - Qjp1 * H|| = %.2E, ||I -␣
 ↪Q'Q|| = %.2E" norm(Q'A * Q[:, 1:k] - H) norm(A * Q[:, 1:k] - Q * H) norm(I -␣
 ↪Q'Q)
```

```
 13.763498 seconds (536.18 k allocations: 8.344 GiB, 4.01% gc time, 2.32%
compilation time)
||Qjp1 * A * Qj - H|| = 3.17E-10, ||A * Qj - Qjp1 * H|| = 1.05E-10, ||I - Q'Q||
= 1.77E-14
```

3. *Implement a Rayleigh-Ritz procedure on top of your CGS2-based Arnoldi process. For the matrix* ***nxp1*** *from the SuiteSparse collection, solve for the 5 largest eigenpairs using the reduced expressions derived in class to monitor convergence of residuals. Verify your residual estimates and compare your converged eigenpairs with those obtained using* ***ArnoldiMethod.jl****.*

[8]:
```
function RR_CGS2_arnoldi(A, q1::Vector{Float64}, k::Int, nev::Int, tol::Float64)
  n, _ = size(A)
```

```
  Q = zeros(Float64, (n, k + 1))
  H = zeros(Float64, (k + 1, k))
  h = zeros(Float64, k + 1)
  w = zeros(Float64, n)
  Q[:, 1] = q1 ./ norm(q1)
  j = 0
  res = 1.; vals = nothing; vecs = nothing; order = nothing
  while (maximum(res) > tol && j < k)
    j += 1
    w .= A * view(Q, :, j)
    H[1:j, j] = view(Q, :, 1:j)'w
    w -= view(Q, :, 1:j) * view(H, 1:j, j)
    h = view(Q, :, 1:j)'w
    w -= view(Q, :, 1:j) * h
    H[j + 1, j] = norm(w)
    Q[:, j + 1] = w ./ H[j + 1, j]
    vals, vecs = eigen(H[1:j, 1:j], sortby=norm)
    order = (j:-1:j-min(nev,j)+1)
    res = norm(H[j + 1, j]) .* norm.(vecs[j, order]) ./ norm.(vals[order])
    println(res)
  end
  return res, vals[order], Q[:, 1:j] * vecs[:, order]
end;
```

[9]:
```
A = mmread("matrices/nxp1.mtx");
n, _ = size(A);
```

[10]:
```
k = 100;
nev = 5;
tol = 1e-6;
res, vals, Y = RR_CGS2_arnoldi(A, rand(n), k, nev, tol);
```

```
[7.090374642347209]
[0.8284070330569548, 9.228841319541667]
[0.4434915240162376, 0.8927117264990008, 9.066083024182348]
[0.21457112249625093, 0.442920736933627, 0.8045647517454083, 7.21921493202951]
[0.11928602369578717, 0.3738453109408641, 0.7578931986655343,
1.2603882110411178, 9.131234735409956]
[0.16226225916015694, 0.2709985035242288, 0.2933756836768329,
0.6130263713084839, 1.401602927293391]
[0.07895957381988386, 0.08002950069169282, 0.2145538725760974,
0.38017175373775025, 0.5177242501755294]
[0.02489817728463199, 0.03349573999590089, 0.18570320706896684,
0.22293148095213805, 0.3991068230304654]
[0.005238621931462636, 0.010014222153382386, 0.08878030855684083,
0.12585345368914447, 0.25090740848559634]
[0.0014013584812090452, 0.0037623453910547327, 0.06157084374909609,
0.1631165668972671, 0.2795844926756254]
```

```
… (25 lines omitted) …
[4.256845501977682e-21, 6.707902344826623e-17, 0.005972799922898638,
0.0009896089956250628, 0.0007743314376812873]
[7.826221391186822e-22, 1.6653645916966283e-17, 0.0026005065311570253,
0.0003895826037765285 3, 0.0003366280324360674]
[1.308998596740863e-22, 3.7315048451089335e-18, 0.0009908157770838857,
0.0001461632726367831 7, 0.0001297260320526151 8]
[2.0169860070823306e-23, 7.519683340927816e-19, 0.0003279358119682916,
4.8347448888753564e-5, 4.3457983113715186e-5]
[3.322066732438434e-24, 1.6556601845260587e-19, 0.0001141117016226469 1,
1.6845815971244283e-5, 1.529855498695441e-5]
[6.549142178839764e-25, 3.972071403066795e-20, 4.516081780187923e-5,
6.678122956668222e-6, 6.1318961498526e-6]
[9.921257181390833e-26, 8.96905530797948e-21, 1.6662433365238596e-5,
2.468352951440891e-6, 2.2926651832135803e-6]
[5.397990835711857e-26, 1.3058046035058266e-20, 6.089837449860018e-6,
9.03639481679509e-7, 8.482992514879445e-7]
[7.48256224410475e-26, 5.945336102700589e-23, 2.482388315804868e-6,
3.6901088365144247e-7, 3.504236169691004e-7]
[4.5076162986800464e-27, 1.809486353391071e-21, 8.872940458754088e-7,
1.3213309916506576e-7, 1.2692111404267288e-7]
```

[11]:
```
res2 = [norm(A * Y[:, i] - vals[i] .* Y[:, i]) / norm(vals[i]) for i in 1:nev];
maximum(res - res2)
```

[11]: 2.1498225596906177e-18

[12]:
```
decomp, hist = partialschur(A, nev=nev);
vals2, Y2 = partialeigen(decomp);
```

[13]:
```
maximum(abs.(vals2[end:-1:1] .- vals))
```

[13]: 3.9466385715059005e-10

[14]:
```
norm(abs.(Y'Y2[:, end:-1:1]) - I)
```

[14]: 6.466847979616974e-10

[15]:
```
vals
```

[15]:
```
5-element Vector{Float64}:
 414.73857473466364
 341.5391676335877
 257.38523662112124
 257.1813229756431
 255.88531378929133
```

4. *Implement a haronic Ritz procedure on top of your CGS2-based Arnoldi process. Still using the* ***nxp1*** *matrix, solve for the 2 eigenpairs closest to* $\sigma = 150$ *using the reduced expressions derived in*

class to monitor convergence of residuals. Verify your residual estimates.

[16]:
```
function HR_CGS2_arnoldi(A, q1::Vector{Float64}, k::Int, nev::Int, sigma::
 ↪ComplexF64, tol::Float64)
  n, _ = size(A)
  Q = zeros(Float64, (n, k + 1))
  H = zeros(Float64, (k + 1, k))
  h = zeros(Float64, k + 1)
  w = zeros(Float64, n)
  Q[:, 1] = q1 ./ norm(q1)
  j = 0
  res = 1.; rho = nothing; vecs = nothing
  while (maximum(res) > tol && j < k)
    j += 1
    w .= A * view(Q, :, j)
    H[1:j, j] = view(Q, :, 1:j)'w
    w -= view(Q, :, 1:j) * view(H, 1:j, j)
    h = view(Q, :, 1:j)'w
    w -= view(Q, :, 1:j) * h
    H[j + 1, j] = norm(w)
    Q[:, j + 1] = w ./ H[j + 1, j]
    e_j = zeros(Float64, j); e_j[j] = 1.
    f = (H[1:j, 1:j] - sigma * I)' \ e_j
    G = H[1:j, 1:j] + norm(H[j + 1, j])^2 .* f * e_j'
    vals, vecs = eigen(G)
    order = sortperm(norm.(vals .- sigma))[1:minimum([j, nev])]
    vals = vals[order]
    vecs = vecs[:, order]
    rho = [vals[i] - norm(H[j + 1, j])^2 * vecs[:, i]'f * vecs[j, i] for i in 1:
 ↪minimum([j, nev])]
    res = sqrt.(real.((vals .- rho) .* conj.(rho .- sigma)))
    res ./= norm.(rho)
    #hres = (vals .- rho) .* conj.(vals .- sigma)
    println(res)
  end
  return res, rho, Q[:, 1:j] * vecs
end;
```

[17]:
```
k = 500
nev = 2;
sigma = 150. + 0im;
res, vals, Y = HR_CGS2_arnoldi(A, rand(n), k, nev, sigma, tol);
```

```
[7.1155478135111645]
[0.7917887588059354, 10.565381044252234]
[0.7337796177874971, 11.007911720270512]
[0.6707995043891417, 11.632680331865295]
[0.18760095207065153, 0.47420054402674255]
```

```
[0.15591064746004635, 0.35979801053080485]
[0.1427914084986224, 0.29264493139156017]
[0.13914491322019323, 0.2882027051377429]
[0.10321000818763305, 0.19835120301048825]
[0.10486359698587726, 0.19843318999089624]
… (187 lines omitted) …
[5.883159959566654e-6, 1.777925763899913e-5]
[5.681505502962584e-6, 1.7121143239333742e-5]
[5.543118851923545e-6, 1.6759962663749463e-5]
[4.875406111042551e-6, 1.4730944193744079e-5]
[3.7858382647010084e-6, 1.1330468758042961e-5]
[2.3377612159301556e-6, 6.885497896899051e-6]
[1.2501745385465917e-6, 3.636393636389095e-6]
[6.699209137066696e-7, 1.9327337711045314e-6]
[4.1778625626173557e-7, 1.1975393560219007e-6]
[3.025649410892489e-7, 8.613799410473239e-7]
```

[18]:
```
res2 = [norm(A * Y[:, i] - vals[i] .* Y[:, i]) / norm(vals[i]) for i in 1:nev];
maximum(res - res2)
```

[18]:
```
4.902515130704452e-13
```

[19]:
```
vals
```

[19]:
```
2-element Vector{ComplexF64}:
 151.07427423341338 - 3.4414301000434583e-15im
 151.37728529983102 + 1.2401664587727536e-15im
```

[20]:
```
res
```

[20]:
```
2-element Vector{Float64}:
 3.025649410892489e-7
 8.613799410473239e-7
```

5. *Implement the Lanczos algorithm. Check how close $Q_{k+1}AQ_k$ is to your tridiagonal matrix, and with what accuracy does your Lanczos relation $AQ_k = Q_{k+1}\underline{T}_k$ hold. Quantify also the loss of orthogonality of the basis. For all this, use the matrix* **Kuu** *from the SuiteSparse collection.*

[21]:
```
function lanczos(A, q1::Vector{Float64}, k::Int)
  n, _ = size(A)
  Q = zeros(Float64, (n, k + 1))
  v = zeros(Float64, n)
  T = zeros(Float64, k + 1, k + 1)
  Q[:, 1] = q1 ./ norm(q1)
  for j in 1:k
    v = A * view(Q, :, j)
    T[j, j] = view(Q, :, j)'v
    v = v - T[j, j] * view(Q, :, j)
```

```
    if (j > 1)
      v -= T[j, j - 1] * view(Q, :, j - 1)
    end
    T[j + 1, j] = norm(v); T[j, j + 1] = T[j + 1, j]
    Q[:, j + 1] = v ./ T[j + 1, j]
  end
  return Q, T[1:k+1, 1:k]
end;
```

[22]:
```
A = mmread("matrices/Kuu.mtx");
n, _ = size(A);
```

[23]:
```
k = 100;
```

[24]:
```
Q, T = @time lanczos(A, rand(n), k);
@printf "||Qjp1 * A * Qj - T|| = %.2E, ||A * Qj - Qjp1 * T|| = %.2E, ||I -␣
 ↪Q'Q|| = %.2E" norm(Q'A * Q[:, 1:k] - T) norm(A * Q[:, 1:k] - Q * T) norm(I -␣
 ↪Q'Q)
```

```
  0.124377 seconds (73.43 k allocations: 42.022 MiB, 80.62% compilation time)
||Qjp1 * A * Qj - T|| = 5.36E-09, ||A * Qj - Qjp1 * T|| = 2.42E-14, ||I - Q'Q||
= 1.09E-10
```

6. *Implement a Rayleigh-Ritz procedure on top of your Lanczos process. For the matrix* `Kuu` *from the SuiteSparse collection, solve for the 5 largest eigenpairs using the reduced expressions derived in class to monitor convergence of residuals. Verify your residual estimates.*

[25]:
```
function RR_lanczos(A, q1::Vector{Float64}, k::Int, nev::Int, tol::Float64)
  n, _ = size(A)
  Q = zeros(Float64, (n, k + 1))
  v = zeros(Float64, n)
  T = zeros(Float64, k + 1, k + 1)
  vals = zeros(Float64, nev, k)
  Q[:, 1] = q1 ./ norm(q1)
  j = 0
  res = 1.; vecs = nothing; order = nothing
  while (maximum(res) > tol && j < k)
    j += 1
    v = A * view(Q, :, j)
    T[j, j] = view(Q, :, j)'v
    v = v - T[j, j] * view(Q, :, j)
    if (j > 1)
      v -= T[j, j - 1] * view(Q, :, j - 1)
    end
    T[j + 1, j] = norm(v); T[j, j + 1] = T[j + 1, j]
    Q[:, j + 1] = v ./ T[j + 1, j]
    _vals, vecs = eigen(T[1:j, 1:j], sortby=norm)
    order = (j:-1:j-min(nev,j)+1)
```

```
    vals[1:minimum([j, nev]), j] = _vals[order]
    res = norm(T[j + 1, j]) .* norm.(vecs[j, order]) ./ norm.(_vals[order])
    println(res)
  end
  return res, vals[:, 1:j], Q[:, 1:j] * vecs[:, order]
end;
```

[26]:
```
k = 500;
nev = 5;
res, vals, Y = RR_lanczos(A, rand(n), k, nev, tol);
```

```
[2.4625425441344784]
[0.41327830292244055, 3.7722324074932017]
[0.2510568529477696, 0.581963288490275, 5.1191287759785205]
[0.13214937773564095, 0.2509002371189868, 0.5672855738650147, 5.84520612441056]
[0.06318540113497328, 0.18088196525273564, 0.42974424602198563,
0.7518629701134443, 7.7093291185020325]
[0.06381923955794075, 0.25153104510319113, 0.2411710024313941,
0.3925753201730301, 0.7187164302863596]
[0.04643432281942248, 0.10510976211523534, 0.10645083390489103,
0.28560123841325585, 0.4688847000049568]
[0.03684137221228557, 0.07815329068451546, 0.20820240729312756,
0.28979415309419065, 0.30226489036027243]
[0.03280835974630158, 0.07350667658763825, 0.18466270766009438,
0.10713158490498861, 0.283511425685189]
[0.03416109882841125, 0.07483695661098766, 0.11579948964955973,
0.1391628277897495, 0.23139822897987483]
… (139 lines omitted) …
[3.0224887029860663e-12, 6.857924837742378e-10, 4.0025431201706356e-8,
3.2100919300610337e-7, 5.07582479408715e-6]
[2.8825560986521067e-12, 6.671881651280143e-10, 4.0343414138142013e-8,
3.307418102883707e-7, 5.35779992241e-6]
[1.9548058220944314e-12, 4.6033415265749854e-10, 2.8724069106120366e-8,
2.402340138046094e-7, 3.980081456177748e-6]
[1.5908482900677428e-12, 3.8027269281376653e-10, 2.4383266238205483e-8,
2.0749579051295376e-7, 3.505674185521048e-6]
[1.4726754126286423e-12, 3.586544310043204e-10, 2.3796305263548744e-8,
2.0698976810291565e-7, 3.5855376900584096e-6]
[1.0031562154755567e-12, 2.4937061908407693e-10, 1.7191355116064675e-8,
1.533456746499118e-7, 2.735178337977455e-6]
[5.881249687196117e-13, 1.4784965383306883e-10, 1.040629774161694e-8,
9.408747530563529e-8, 1.7047532936364098e-6]
[5.324852236264892e-13, 1.355626740151096e-10, 9.759802613388351e-9,
8.950837728718988e-8, 1.6480204776292318e-6]
[3.264143349138407e-13, 8.429624947080699e-11, 6.226491411467588e-9,
5.80325730478253e-8, 1.0880384900102223e-6]
[1.9448829826343476e-13, 5.067548335766482e-11, 3.802324630060077e-9,
3.578331928739272e-8, 6.780925668354701e-7]
```

[27]:
```
res2 = [norm(A * Y[:, i] - vals[i, end] .* Y[:, i]) / norm(vals[i, end]) for i␣
↪in 1:nev];
maximum(res - res2)
```

[27]:
```
3.473576237571479e-18
```

[28]:
```
vals[:, end]
```

[28]:
```
5-element Vector{Float64}:
 54.08205599641254
 53.98165804506267
 53.81549645283014
 53.72040205150596
 53.62254145479188
```

7. *Implement a harmonic Ritz procedure on top of your Lanczos process. For the matrix* **Kuu** *from the SuiteSparse collection, solve for the 2 eigenpairs closest to* $\sigma = 25$ *using the reduced expressions derived in class to monitor convergence of residuals. Verify your residual estimates.*

[29]:
```
function HR_lanczos(A, q1::Vector{Float64}, k::Int, nev::Int, sigma::Float64,␣
↪tol::Float64)
 n, _ = size(A)
 Q = zeros(Float64, (n, k + 1))
 v = zeros(Float64, n)
 T = zeros(Float64, k + 1, k + 1)
 rhos = zeros(ComplexF64, nev, k)
 Q[:, 1] = q1 ./ norm(q1)
 j = 0
 res = 1.; vecs = nothing; order = nothing
 while (maximum(res) > tol && j < k)
   j += 1
   v = A * view(Q, :, j)
   T[j, j] = view(Q, :, j)'v
   v = v - T[j, j] * view(Q, :, j)
   if (j > 1)
     v -= T[j, j - 1] * view(Q, :, j - 1)
   end
   T[j + 1, j] = norm(v); T[j, j + 1] = T[j + 1, j]
   Q[:, j + 1] = v ./ T[j + 1, j]
   e_j = zeros(Float64, j); e_j[j] = 1.
   f = (T[1:j, 1:j] - sigma * I) \ e_j
   G = T[1:j, 1:j] + T[j + 1, j]^2 .* f * e_j'
   vals, vecs = eigen(G)
   order = sortperm(norm.(vals .- sigma))[1:minimum([j, nev])]
   vals = vals[order]
   vecs = vecs[:, order]
   rho = [vals[i] - T[j + 1, j]^2 * vecs[:, i]'f * vecs[j, i] for i in 1:
↪minimum([j, nev])]
```

```
    res = sqrt.(real.((vals .- rho) .* conj.(rho .- sigma)))
    res ./= norm.(rho)
    rhos[1:minimum([j, nev]), j] = rho
    println(res)
  end
  return res, rhos[:, 1:j], Q[:, 1:j] * vecs
end;
```

[30]:
```
k = 1_000;
nev = 2;
sigma = 25.;
res, vals, Y = HR_lanczos(A, rand(n), k, nev, sigma, tol);
```

```
[2.456305001476329]
[2.9114411692788744, 0.4116839286405057]
[0.5131279159794808, 0.24836741591103992]
[0.2292669847676493, 0.5097142134673092]
[0.16978014514406614, 0.37881723553376795]
[0.1545356899507271, 0.2973795108291189]
[0.10337829443590445, 0.23062969436460493]
[0.1997294117419717, 0.11529547192304967]
[0.1792739467704381, 0.08995516105308346]
[0.14569275147853783, 0.10663381622571275]
… (953 lines omitted) …
[1.5584255913804753e-6, 2.1701912259439833e-6]
[1.2975234409166461e-6, 1.8050122902834276e-6]
[1.2729837539553341e-6, 1.7682917455116285e-6]
[1.0687607996721643e-6, 1.4823072691583365e-6]
[1.035712610424044e-6, 1.4332000971818719e-6]
[9.143012778807161e-7, 1.2653361786146835e-6]
[9.007723300036238e-7, 1.2449562292339883e-6]
[7.712904440787114e-7, 1.0651286026687723e-6]
[7.613637923314611e-7, 1.0500496828269974e-6]
[6.766786674673904e-7, 9.321333525268427e-7]
```

[31]:
```
res2 = [norm(A * Y[:, i] - vals[i, end] .* Y[:, i]) / norm(vals[i, end]) for i␣
 ↪in 1:nev];
maximum(res - res2)
```

[31]:
```
3.653102329643186e-15
```

[32]:
```
vals[:, end]
```

[32]:
```
2-element Vector{ComplexF64}:
 25.005559914325264 + 0.0im
   25.0615529753166 + 0.0im
```

[33]:
```
Y
```

```
[33]: 7102×2 Matrix{Float64}:
        0.00166798    0.0017046
        0.00334999    0.00421569
        0.00610706    0.0100049
       -0.00360304   -0.00390964
       -0.0101474    -0.0159741
       -0.00257309   -0.00371062
        0.00166762    0.00151604
        0.00857563    0.010363
        0.012858      0.0192622
       -0.00519503   -0.00561935
       -0.0140196    -0.0192928
       -0.00487764   -0.00556107
       -0.000283369  -0.00149261

       -4.78245e-6   -6.13793e-6
        7.13208e-7    1.10287e-6
        4.82961e-6    6.40523e-6
        1.10087e-6    1.10232e-6
       -3.46603e-7   -3.18188e-7
       -9.21224e-7   -1.03736e-6
       -2.61658e-6   -3.9056e-6
       -1.87492e-8    3.46207e-8
        2.66212e-6    3.80099e-6
        1.06069e-6    1.39057e-6
       -2.01083e-7   -2.64097e-7
       -3.83476e-7   -5.12058e-7
```