Numerical Linear Algebra for Computational Sciences and Information Engineering
CITHN2006

Orthogonalization

by Nicolas Venkovic

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

[1]:
```
using LinearAlgebra, Printf, BenchmarkTools, Random, SparseArrays
```

1. *Write a function called* ***conditioned_matrix*** *that creates a random* $m \times n$ *matrix with condition number* κ *and maximum singular value* $\sigma_1 = 1$.

[2]:
```
function conditioned_matrix(m::Int, n::Int, cond::Float64)
  U, _ = qr(randn(m, m))
  V, _ = qr(randn(n, n))
  s = range(1.0, 1/cond; length=n)
  Σ = Diagonal(s)
  A = U[:, 1:n] * Σ * V'
  return A
end;
```

2. *Implement and test and time your implementation of Householder QR.*

[3]:
```
function HouseholderQR(A)
  m, n = size(A)
  Q = Matrix{Float64}(I, m, m)
  R = copy(A)
  for k = 1:n
    # Compute Householder vector
    v = R[k:m, k]
    v[1] -= sign(v[1]) * norm(v) # usage of sign(v[1]) allows to prevent␣
 ↪catastrophic cancellation
    beta = 2. / v'v
    # Apply Householder reflection
    R[k:m, k:n] -= beta .* v * (v'view(R, k:m, k:n))
    Q[1:m, k:m] -= view(Q, 1:m, k:m) * v * (beta .* v')
  end
```

```
  return Q, R
end;
```

[4]:
```
m = 500;
n = 200;
Random.seed!(3);
A1 = conditioned_matrix(m, n, 1e10);
A2 = conditioned_matrix(m, n, 1e12);
```

[5]:
```
Q, R = @btime HouseholderQR(A1);
@printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A1 - Q * R) norm(I - Q'Q)
```

```
  301.746 ms (6607 allocations: 1.10 GiB)
||A - QR|| = 9.09E-15, ||I - Q'Q|| = 3.10E-14
```

[6]:
```
Q, R = @btime HouseholderQR(A2);
@printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A2 - Q * R) norm(I - Q'Q)
```

```
  300.292 ms (6607 allocations: 1.10 GiB)
||A - QR|| = 9.21E-15, ||I - Q'Q|| = 3.07E-14
```

3. *Implement CholeskyQR, CholeskyQR2 and Shifted CholeskyQR3.*

[7]:
```
function CholeskyQR(A)
  X = A'A # ==> if A is 1e12 then A'A ==> Ae24
  chol = cholesky(X)
  Q = A / chol.U
  return Q, chol.U
end;
```

[8]:
```
function CholeskyQR2(A)
  Q1, R1 = CholeskyQR(A)
  Q, R2 = CholeskyQR(Q1)
  return Q, R2 * R1
end;
```

[9]:
```
function ShiftedCholeskyQR(A)
  X = A'A
  s = 11 * (m * n + n * (n + 1)) * eps() * norm(A)
  X += s * I
  chol = cholesky(X)
  Q = A / chol.U
  return Q, chol.U
end;
```

[10]:
```
function ShiftedCholeskyQR3(A)
  Q1, R1 = ShiftedCholeskyQR(A)
  Q, R2 = CholeskyQR2(Q1)
  return Q, R2 * R1
```

```
end;
```

4. *Compare the runtime, loss of orthogonality and residual obtained with each CholeskyQR implementation for multiple tall-and-skinny matrices with different conditioning numbers.*

[11]:
```
try
  Q, R = @btime CholeskyQR(A1);
  @printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A1 - Q * R) norm(I - Q'Q)
catch e
  println("Error", e)
end
```

```
ErrorPosDefException(200)
```

[12]:
```
try
  Q, R = @btime CholeskyQR(A2);
  @printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A1 - Q * R) norm(I - Q'Q)
catch e
  println("Error", e)
end
```

```
  1.573 ms (10 allocations: 1.37 MiB)
||A - QR|| = 1.16E+01, ||I - Q'Q|| = 1.00E+00
```

[13]:
```
try
  Q, R = @btime CholeskyQR2(A1);
  @printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A1 - Q * R) norm(I -␣
 ↪Q'Q)
catch e
  println("Error", e)
end
```

```
ErrorPosDefException(200)
```

[14]:
```
try
  Q, R = @btime CholeskyQR2(A2);
  @printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A2 - Q * R) norm(I - Q'Q)
catch e
  println("Error", e)
end
```

```
  3.182 ms (22 allocations: 3.05 MiB)
||A - QR|| = 3.47E-15, ||I - Q'Q|| = 7.00E-15
```

[15]:
```
Q, R = @btime ShiftedCholeskyQR3(A1);
@printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A1 - Q * R) norm(I - Q'Q)
```

```
  5.656 ms (51 allocations: 5.04 MiB)
||A - QR|| = 3.97E-15, ||I - Q'Q|| = 5.57E-15
```

[16]:
```
Q, R = @btime ShiftedCholeskyQR3(A2);
@printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A2 - Q * R) norm(I - Q'Q)
```

```
  4.986 ms (51 allocations: 5.04 MiB)
||A - QR|| = 3.90E-15, ||I - Q'Q|| = 5.57E-15
```

5. *Implement CGS, CGS2 and MGS procedures.*

[17]:
```
function CGS(X::Array{Float64,2})
  n, m = size(X)
  Q = Array{Float64,2}(undef, n, m)
  R = zeros(Float64, m, m)
  R[1, 1] = norm(view(X, :, 1))
  Q[:, 1] = view(X,:, 1) ./ R[1, 1]
  for i in 2:m
    Q[:, i] = view(X, :, i)
    R[1:i-1, i] .= view(Q, :, 1:i-1)'view(X, :, i)
    Q[:, i] -= view(Q, :, 1:i-1) * view(R, 1:i-1, i)
    R[i, i] = norm(view(Q, :, i))
    Q[:, i] ./= R[i, i]
  end
  return Q, R
end;
```

[18]:
```
function MGS(X::Array{Float64,2})
  n, m = size(X)
  Q = Array{Float64,2}(undef, n, m)
  R = zeros(Float64, m, m)
  R[1, 1] = norm(view(X, :, 1))
  Q[:, 1] = view(X, :, 1) ./ R[1, 1]
  for i in 2:m
    Q[:, i] = view(X, :, i)
    for j in 1:i-1
      R[j, i] = view(Q, :, j)'view(Q, :, i)
      Q[:, i] -= R[j, i] .* view(Q, :, j)
    end
    R[i, i] = norm(view(Q, :, i))
    Q[:, i] ./= R[i, i]
  end
  return Q, R
end;
```

[19]:
```
function CGS2(X::Array{Float64,2})
  n, m = size(X)
  Q = Array{Float64,2}(undef, n, m)
  R = zeros(Float64, m, m)
  R[1, 1] = norm(view(X, :, 1))
  Q[:, 1] = view(X, :, 1) ./ R[1, 1]
```

```
  for i in 2:m
    Q[:, i] = view(X, :, i)
    R[1:i-1, i] .= view(Q, :, 1:i-1)'view(X, :, i)
    Q[:, i] -= view(Q, :, 1:i-1) * view(R, 1:i-1, i)
    r = view(Q, :, 1:i-1)'view(Q, :, i)
    Q[:, i] -= view(Q,:, 1:i-1) * r
    R[i, i] = norm(view(Q, :, i))
    Q[:, i] ./= R[i, i]
  end
  return Q, R
end;
```

6. *Compare the runtime, loss of orthogonality and residual obtained with each Gram-Schmidt implementation for multiple tall-and-skinny matrices with different conditioning numbers.*

[20]:
```
Q, R = @btime CGS(A1);
@printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A1 - Q * R) norm(I - Q'Q)
```

```
  2.610 ms (2596 allocations: 4.35 MiB)
||A - QR|| = 2.45E-15, ||I - Q'Q|| = 3.08E-05
```

[21]:
```
Q, R = @btime CGS(A2);
@printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A2 - Q * R) norm(I - Q'Q)
```

```
  2.620 ms (2596 allocations: 4.35 MiB)
||A - QR|| = 2.44E-15, ||I - Q'Q|| = 8.85E-04
```

[22]:
```
Q, R = @btime MGS(A1);
@printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A1 - Q * R) norm(I - Q'Q)
```

```
  32.690 ms (179507 allocations: 235.50 MiB)
||A - QR|| = 3.81E-15, ||I - Q'Q|| = 1.42E-06
```

[23]:
```
Q, R = @btime MGS(A2);
@printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A2 - Q * R) norm(I - Q'Q)
```

```
  32.371 ms (179507 allocations: 235.50 MiB)
||A - QR|| = 3.81E-15, ||I - Q'Q|| = 1.59E-04
```

[24]:
```
Q, R = @btime CGS2(A1);
@printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A1 - Q * R) norm(I - Q'Q)
```

```
  4.964 ms (4785 allocations: 6.86 MiB)
||A - QR|| = 3.18E-15, ||I - Q'Q|| = 6.61E-15
```

[25]:
```
Q, R = @btime CGS2(A2);
@printf "||A - QR|| = %.2E, ||I - Q'Q|| = %.2E" norm(A2 - Q * R) norm(I - Q'Q)
```

```
  4.975 ms (4785 allocations: 6.86 MiB)
||A - QR|| = 3.17E-15, ||I - Q'Q|| = 6.48E-15
```

Note that despite requiring twice the number of FLOPs compared to MGS, CGS2 is still significantly faster due to a higher arithmetic intensity.