QR Algorithm for Eigenvalue Problems with Hessenberg/Givens Tricks

- Mathe 3 (CES)
- WS20
- Lambert Theisen (`theisen@acom.rwth-aachen.de`)


```
Plots.PlotlyBackend()
```

```
begin
    using LinearAlgebra, PlutoUI, Random, SparseArrays, Plots
    plotly()
end
```

QR Algorithm Native

We use Julia's standard `qr()` function and implement:

1. Given $A \in \mathbb{R}^{n \times n}$
2. Initialize $Q^{(m+1)} = I$
3. For $k = 1, \dots, m$:
    1. Calculate QR-Decomposition: $A_k = Q_k R_k$
    2. Update: $A_{k+1} = R_k Q_k$
4. Return diagonal entries of A_m and $Q^{(m)} = Q_m \cdots Q_0 Q_1$

```
qra_general (generic function with 1 method)
```

```
function qra_general(A, m)
    @assert size(A)[1] == size(A)[2] && length(size(A))==2
    n = size(A)[1]
    Qm = I(n)
    for k=1:m
        Q, R = qr(A)
        A = R * Q
        Qm = Qm * Q
    end
    return diag(A), Qm
end
```

Check Validity of Implementation

```
A = 3×3 Array{Float64,2}:
     3.0  2.0   3.0
     4.0  7.0   6.0
     7.0  8.0  11.0
```

```
A = 1. * [
    1 2 3
    4 5 6
    7 8 9
] + 2I(3)
```

```
(Float64[18.1168,  2.0,  0.883156],  3×3 Array{Float64,2}:           )
                                      0.231971  -0.408248  -0.882906
                                      0.525322   0.816497  -0.23952
                                      0.818673  -0.408248   0.403865
```

```
qra_general(A, 100)
```

```
Float64[-1.77636e-15,  -2.22045e-16,  -7.10543e-15]
```

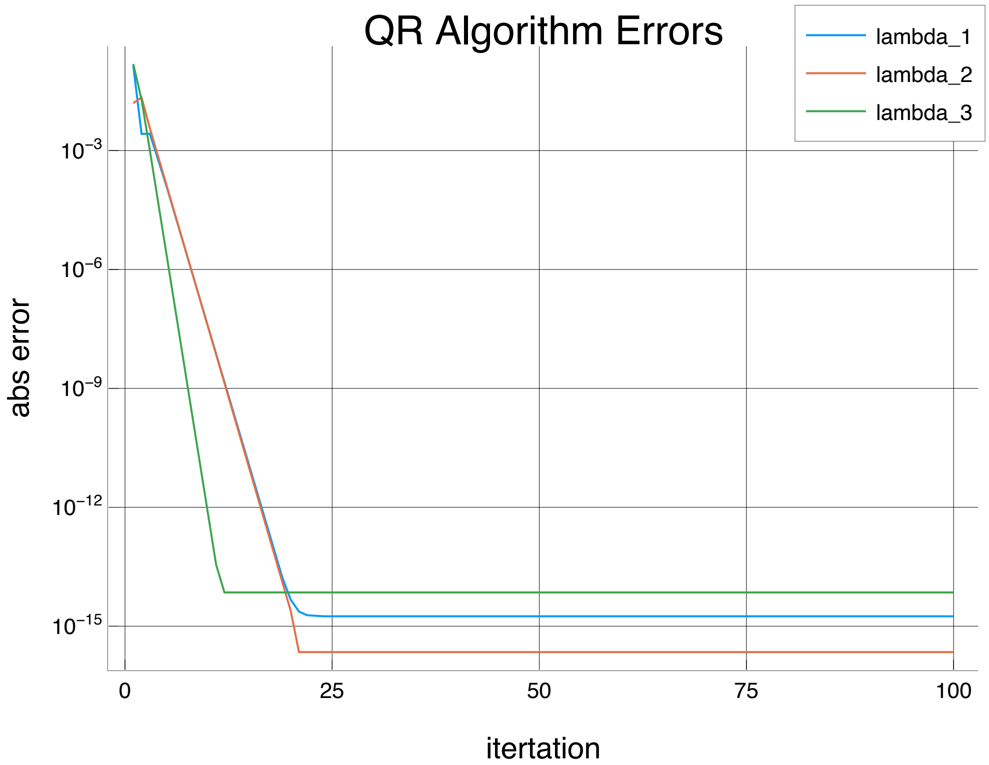
```
eigen(A).values - sort(qra_general(A, 100)[1], rev=false)
```

Check Convergence

```
begin
    N = 100
    tape = Array[]
    for k=1:N
        push!(tape, sort(qra_general(A, k)[1], rev=false)) # *dont do this, very inefficient*
    end
end
```

```
errors =
 Array{Float64,1}[Float64[0.137636,  0.00263023,  0.00266808,  0.000628078,  0.00012773,
```

```
errors = [
    abs.(map(x -> x[1], tape) .- eigen(A).values[1]),
    abs.(map(x -> x[2], tape) .- eigen(A).values[2]),
    abs.(map(x -> x[3], tape) .- eigen(A).values[3]),
]
```




```
plot([errors[1],errors[2],errors[3]], yaxis=:log, label=["lambda_1" "lambda_2" "lambda_3"], title="QR Algorithm Errors", xlabel="itertation", ylabel="abs error")
```

Improve QR Algorithm with Upper Hessenberg Matrix Preconditioning

- Idea: QR decomposition needs $\mathcal{O}(n^3)$, QR for Hessenberg matrices is easier done in $\mathcal{O}(n^2)$. Linear complexity is even possible if A is symmetric. In this case, the QR decomposition only needs $\mathcal{O}(n)$ Givens rotations with constant effort.
- Therefore transform the matix A to upper Hessenberg form with similarity transforms in $\mathcal{O}(n^3)$ (also cubic, but only needs to be done once) and use this matrix for the QR algorithm.

Upper Hessenberg Shape

$$H = \begin{pmatrix} h_{11} & h_{12} & h_{13} & \cdots & h_{1n} \\ h_{21} & h_{22} & h_{23} & \cdots & h_{2n} \\ 0 & h_{32} & h_{33} & \cdots & h_{3n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & h_{nn-1} & h_{nn} \end{pmatrix}$$

Algorithm [1]:

1. Given: $A \in \mathbb{R}^{n \times n}$
2. For $k = 1 \ldots n-2$ do:

1. $$[v,\beta] \leftarrow \text{house}(A(k+1:n,k))$$

2. $$A(k+1:n,k:n) \leftarrow (I-\beta vv^T)A(k+1:n,k:n)$$

3. $$A(1:n,k+1:n) \leftarrow A(1:n,k+1:n)(I-\beta vv^T)$$

with Householder reflection vector v and weight $\beta = 2/(v^T v)$.

[1]: https://www.tu-chemnitz.de/mathematik/numa/lehre/nla-2015/Folien/nla-kapitel6.pdf

```
upperhessenberg (generic function with 1 method)
```

```
function upperhessenberg(A)
    @assert size(A)[1] == size(A)[2] && length(size(A))==2
    n = size(A)[1]
    for k = 1:n-2
        v, β = householdervec(A[k+1:n,k])
        A[k+1:n, k:n] = (I(n-k) - β * v * v') * A[k+1:n, k:n]
        A[1:n, k+1:n] = A[1:n, k+1:n] * (I(n-k) - β * v * v')
    end
    return A
end
```

```
householdervec (generic function with 1 method)
```

```
function householdervec(x)
    @assert size(x)[1]>0 && length(size(x))==1
    n = size(x)[1]
    e1 = I(n)[:,1]
    v = x + norm(x, 2) * e1
    β = 2 / (v' * v)
    return v, β
end
```

Check if Householder ad Upperhessenberg Transformations work

```
md"""
### Check if Householder ad Upperhessenberg Transformations work
"""
```

```
BB = 3×3 Array{Float64,2}:
      1.0  2.0  3.0
      4.0  5.0  6.0
      7.0  8.0  9.0
```

```
BB = 1. * [
    1 2 3
    4 5 6
    7 8 9
]
```

```
4×4 Array{Float64,2}:
 -1.22404      -0.153046   -0.578656  -1.10883
  4.16334e-17   0.18988     0.119543   0.268484
  1.21431e-17   0.473465    0.34666    0.659726
 -1.11022e-16  -0.0362976   0.590894  -0.554913
```

```
begin
    CC = rand(4,4)
    v, β = householdervec(CC[:,1])
    CC = (I(4) - 2/(v' * v) * v * v') * CC
    # CC should now have zeros in first column except diagonal
```

```
    # To get all other columns to zero, repeat with sub blockmatrices
end
```

```
3×3 Array{Float64,2}:
  1.0          -3.59701   -0.248069
 -8.06226      14.0462     2.83077
 -4.44089e-16   0.830769  -0.0461538
```

```
upperhessenberg(BB) # Should have only one lower sub diagonal
```

QR Algorithm for symmetric Hessenberg matrices

Symmertric hessenberg matrices are tridiagonal (only diag plus uppe and lower sub-diagonal). For the QR decomposition, we only have to make the lower sub diagonal entries to zero to obtain the upper right triangular matrix. This can be done by using Givens roations:

Givens Rotations [1]

Given a matrix A, we can make to entry A_{ij} to zero with $A_{\text{new}} = G(A, i, j)$ where

$$G(A,i,j) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix},$$

with

- $$r = \sqrt{A_{jj}^2 + A_{ij}^2}$$
- $$s = -A_{ij}/r$$
- $$c = A_{jj}/r$$

[1]: https://en.wikipedia.org/wiki/Givens_rotation

```
givens_rotation_matrix (generic function with 1 method)
```

```
function givens_rotation_matrix(A, i, j)
    n = size(A)[1]
    r = sqrt(A[j,j]^2 + A[i,j]^2)
    s = -A[i,j]/r
    c = A[j,j]/r
    G = sparse(1.0I, n, n)
    G[i,i] = c
    G[j,j] = c
    G[i,j] = s
```

```
    G[j,i] = -s
    return G
end
```

Check Givens Rotation

```
3×3 Array{Float64,2}:
 4.12311   5.33578    6.54846
 0.0      -0.727607  -1.45521
 7.0       8.0        9.0
```

```
begin
    AA =    1. * [
        1 2 3
        4 5 6
        7 8 9
    ]
    AA = givens_rotation_matrix(AA, 2, 1) * AA
end
```

Implement QR Decomposition for Symmetric Hessenberg Matrices

- Just iterate over sub-diagonal entries and make them zero to get R. Store all Givens rotations to get Q.

```
qr_symm_hess (generic function with 1 method)
```

```
function qr_symm_hess(A)
    # QR decomposition for symmetric Hessenberg matrix <=> tridiagonal matrix
    n = size(A)[1]
    abstol = 1E-14
    isapproxsymmetric = any(isapprox.(-0.5 * (A - A'), zeros(n, n), atol=abstol))
    isapproxtridiagonal = any(isapprox.(A, Tridiagonal(A), atol=abstol))
    @assert isapproxsymmetric && isapproxtridiagonal

    Am = A
    Qm = sparse(1.0I, n, n)
    for m=1:n-1
        G = givens_rotation_matrix(Am, m+1,m)
        Am = G * Am
        Qm = Qm * G'
    end
    Q = Qm
    R = Qm' * A
    return Array(Q), R
end
```

Check QR Decomposition for Tridiagonal Matrices

```
md"""
## Check QR Decomposition for Tridiagonal Matrices
```

```
"""
```

```
EE = 3×3 Array{Int64,2}:
      6  5  0
      5  1  4
      0  4  3
```

```
EE = [6 5 0; 5 1 4; 0 4 3]
```

```
3×3 Array{Float64,2}:
 6.0  5.0  4.44089e-16
 5.0  1.0  4.0
 0.0  4.0  3.0
```

```
qr(EE).Q * qr(EE).R
```

```
3×3 Array{Float64,2}:
 6.0          5.0  0.0
 5.0          1.0  4.0
 4.20473e-16  4.0  3.0
```

```
qr_symm_hess(EE)[1] * qr_symm_hess(EE)[2]
```

QR Algorithm for Tridiagonal Matrices (Symmetric Upper Hessenberg)

- Same as native implementation, but first transform to Hessenberg and then use the cheap Givens rotation style to get all the QR decompositions.

```
qra_symm (generic function with 1 method)
```

```
function qra_symm(A, m)
    n = size(A)[1]
    @assert size(A)[1] == size(A)[2] && length(size(A))==2
    isapproxsymmetric = any(isapprox.(-0.5 * (A - A'), zeros(n, n), atol=1E-8))
    @assert isapproxsymmetric
    Qm = I(n)
    # A = upperhessenberg(A)
    for k=1:m
        Q, R = qr_symm_hess(A)
        A = R * Q
        Qm = Qm * Q
    end
    return diag(A), Array(Qm)
end
```

Check Validity of our QR Algorithm for Tridiagonal Matrice

```
DD = 3×3 Array{Int64,2}:
      6  5  0
      5  1  4
      0  4  3
```

```
DD = [6 5 0; 5 1 4; 0 4 3]
```

```
(Float64[9.84316, 4.02176, -3.86492], 3×3 Array{Float64,2}:                    )
                                      0.746882  -0.530327   0.401149
                                      0.574078   0.209823  -0.79146
                                      0.335563   0.821418   0.461162
```

```
qra_symm(DD, 1000)
```

```
(Float64[9.84316, 4.02176, -3.86492], 3×3 Array{Float64,2}:                    )
                                      0.746882   0.530327  -0.401149
                                      0.574078  -0.209823   0.79146
                                      0.335563  -0.821418  -0.461162
```

```
qra_general(DD, 1000)
```

```
3×3 Array{Float64,2}:
 6.0          5.0  4.88498e-15
 5.0          1.0  4.0
 5.32907e-15  4.0  3.0
```

```
qra_symm(DD, 1000)[2] * diagm(qra_symm(DD, 1000)[1]) * qra_symm(DD, 1000)[2]'
```

Speed Check

We still loose against Julia's native `qr` -method. 😐

- Homework: Improve this.

```
QRA General:
  0.166218 seconds (30.90 k allocations: 51.228 MiB, 4.84% gc time)
QRA Own:
  0.417899 seconds (99.22 k allocations: 752.204 MiB, 13.56% gc time)
```

```
with_terminal() do
    N = 100
    M = 50
    C = Array(Symmetric(rand(N,N)))

    # @time qr(C)
    # @time qr_symm_hess(C)
    # @time upperhessenberg(C)

    println("QRA General:")
    @time Am1, Qm1 = qra_general(C, M)
    println("QRA Own:")
    @time Am2, Qm2 = qra_symm(C, M)
end
```