PRACTICAL QR ALGO

Hessenberg reduction

```julia
function gehrd!(A)
    """Reduced square matrix A to upper Hessenberg form"""
    n = size(A,1)
    for k=1:n-2
        # Compute Householder reflection for column k
        beta, v = house(A[k+1:n,k])
        # Apply the reflection on the left
        apply_left_householder!(A, k+1, k, beta, v)
        # Apply the reflection on the right
        _right_householder!(A, n, k+1, beta, v
```

```julia
"""
    apply_right_householder!(A, row, col, beta, v)

Apply a Householder reflection to the right of matrix `A`.

`row` is the ending row index to apply the transform.
`col` is the starting column index. `beta` and `v` are the
parameters for the Householder reflection.
"""
function apply_right_householder!(A, row, col, beta, v)
    n = size(A,1)
    Av = zeros(n)
    lv = length(v)
    # Apply transform to the right
    # Av = beta * A * v
    for j=col:col+lv-1
        for i=1:row
            Av[i] += v[j-col+1] * A[i,j]
        end
    end
    for i=1:row
        Av[i] *= beta
    end
    # A - beta (Av) v^T
    for j=col:col+lv-1, i=1:row
        A[i,j] -= Av[i] * v[j-col+1]
    end
end
```

Resulting A is upper Hessenberg

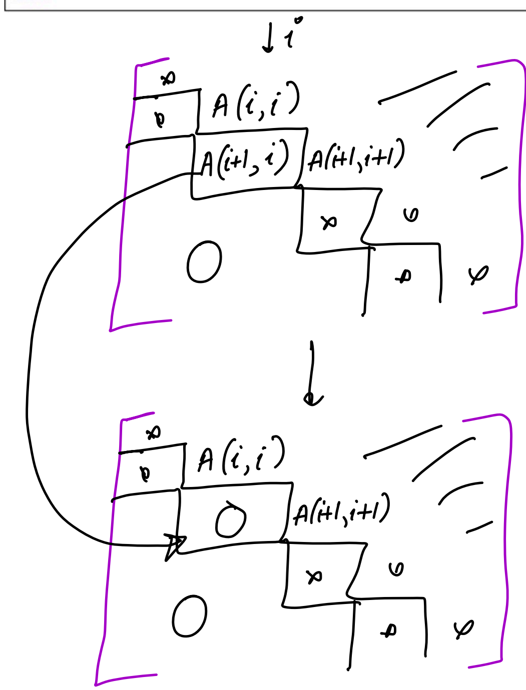
```julia
"""
    _householder!(A, row, col, beta, v)

Apply a Householder reflection to the left of matrix `A`.

`row` is the starting row index to apply the transform.
`col` is the starting column index. `beta` and `v` are the
parameters for the Householder reflection.

"""
function apply_left_householder!(A, row, col, beta, v)
    n = size(A,1)
    vA = zeros(n)
    lv = length(v)
    # Apply transform to the left
    # vA = beta * v^T * A
    for j=col:n
        for i=row:row+lv-1
            vA[j] += v[i-row+1] * A[i,j]
        end
        vA[j] *= beta
    end
    # A - beta v (v^T A)
    for j=col:n, i=row:row+lv-1
        A[i,j] -= v[i-row+1] * vA[j]
    end
end
```

PRACTICAL QR ALGO

```
function gees!(A)
    n = size(A,1)
    if n==1
        return A[1,1]
    end
    D = zeros(Complex{Float64},n)

    # Tolerance for deflation
    tol = eps(Float64)

    q = n # Size of the matrix we are currently working with
    iter = 1 # Counter to detect convergence failure
    iter_per_evalue = 0 # Used to trigger an exceptional shift

    reduce_eps!(A, tol) # Zero out small entries

    while q > 0

        if iter > 10*n
            @''Code failed to converge''
        end

        deflation = true # Were we able to deflate the matrix?

        while deflation
            @''Test for deflation and record eigenvalues if
              converged''

        end

        # If q <= 2 we compute the eigenvalues at the next
        # iteration.
        if q >= 3
            @''Perform Francis QR step''
        end
    end
end
```

A is assumed to be upper Hessenberg

```
function reduce_eps!(A, tol)
    # Zero out all small entries on the sub-diagonal
    n = size(A,1)
    for i=1:n-1
        if abs(A[i+1,i]) < tol * (abs(A[i,i])+abs(A[i+1,i
            +1]))
            A[i+1,i] = 0
        end
    end
end
```



PRACTICAL QR ALGO

```
function gees!(A)
    n = size(A,1)
    if n==1
        return A[1,1]
    end
    D = zeros(Complex{Float64},n)

    # Tolerance for deflation
    tol = eps(Float64)

    q = n # Size of the matrix we are currently working with
    iter = 1 # Counter to detect convergence failure
    iter_per_evalue = 0 # Used to trigger an exceptional shift

    reduce_eps!(A, tol) # Zero out small entries

    while q > 0

        if iter > 10*n
            @''Code failed to converge''
        end

        deflation = true # Were we able to deflate the matrix?

        while deflation
            @''Test for deflation and record eigenvalues if
              converged''

        end

        # If q <= 2 we compute the eigenvalues at the next
        # iteration.
        if q >= 3
            @''Perform Francis QR step''
        end
    end
end
```

```
# Definition of @"Test for deflation and record eigenvalues if converged"
deflation = false
@"Test deflation for the last 2x2 block"
@"Test deflation for the last 1x1 block"
```

The 2 × 2 block case is as follows:

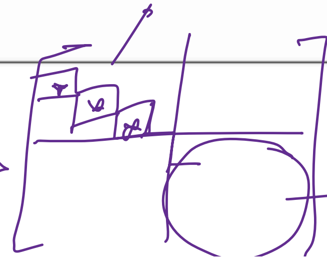
```
# Definition of @"Test deflation for the last 2x2 block"
if q <= 2 || A[q-1,q-2] == 0
    if q >= 2
        # The last 2x2 block has converged
        deflation = true # Deflating now
        # Compute the eigenvalues
        a = A[q-1,q-1]; b = A[q-1,q]; c = A[q,q-1];
            d = A[q,q]
        htr = (a+d)/2; dis = (a-d)^2/4 + b*c
        if dis > 0 # Pair of real eigenvalues
            D[q-1] = htr - sqrt(dis)
            D[q]   = htr + sqrt(dis)
        else # Complex conjugate eigenvalues
            D[q-1] = htr - sqrt(-dis)*im
            D[q]   = htr + sqrt(-dis)*im
        end
        # Reduce the size of the matrix
        q -= 2
        if q>=1
            A = A[1:q,1:q]
        end
    end
end

if q==0
    return D
end
```

In the 1 × 1 case, we perform similar operations:

```
# Definition of @"Test deflation for the last 1x1 block"
if q <= 1 || A[q,q-1] == 0
    deflation = true
    D[q] = A[q,q]
    q -= 1
    if q>=1
        A = A[1:q,1:q]
    end

if q==0
    return D
end

if deflation
    iter_per_evalue = 0 # Reset the counter
end
```

upper Hessenberg

This loop finds

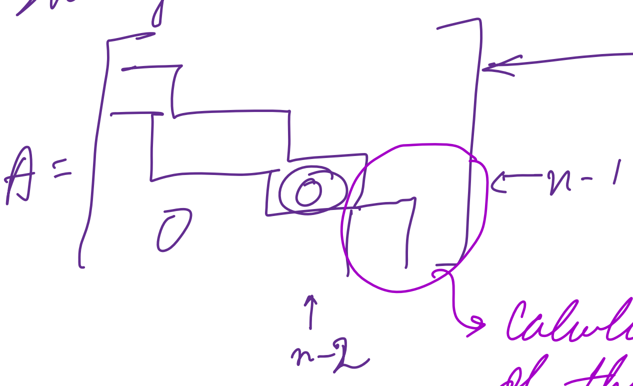


largest quasi-triangular block.

PRACTICAL QR ALGO

see below

Start from $q = n$.



Deflate

Calculate the eigenvalues of this block & store them in $D[q-1]$ & $D[q]$

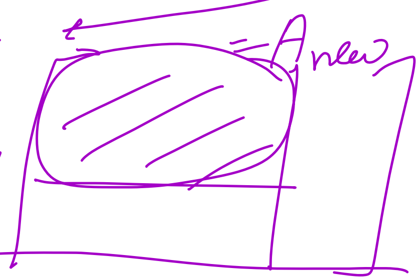
```
# Definition of @"Test for deflation and record eigenvalues if converged"
deflation = false
@"Test deflation for the last 2x2 block"
@"Test deflation for the last 1x1 block"
```

The 2×2 block case is as follows:

```
# Definition of @"Test deflation for the last 2x2 block"
if q <= 2 || A[q-1,q-2] == 0
    if q >= 2
        # The last 2x2 block has converged
        deflation = true # Deflating now
        # Compute the eigenvalues
        a = A[q-1,q-1]; b = A[q-1,q]; c = A[q,q-1];
            d = A[q,q]
        htr = (a+d)/2; dis = (a-d)^2/4 + b*c
        if dis > 0 # Pair of real eigenvalues
            D[q-1] = htr - sqrt(dis)
            D[q]   = htr + sqrt(dis)
        else # Complex conjugate eigenvalues
            D[q-1] = htr - sqrt(-dis)*im
            D[q]   = htr + sqrt(-dis)*im
        end
        # Reduce the size of the matrix
        q -= 2
        if q>=1
            A = A[1:q,1:q]
        end
    end
end

if q==0
    return D
end
```

Reduce $q = q-2$

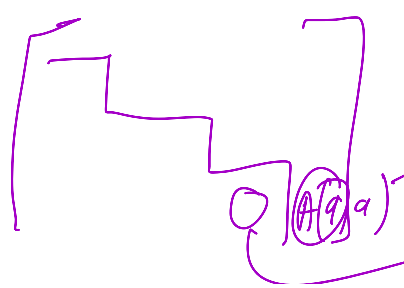
& change $A_{old} =$



In the 1×1 case, we perform similar operations:

```
# Definition of @"Test deflation for the last 1x1 block"
if q <= 1 || A[q,q-1] == 0
    deflation = true
    D[q] = A[q,q]
    q -= 1
    if q>=1
        A = A[1:q,1:q]
    end
```

```
if q==0
    return D
end

if deflation
    iter_per_evalue = 0 # Reset the counter
end
```



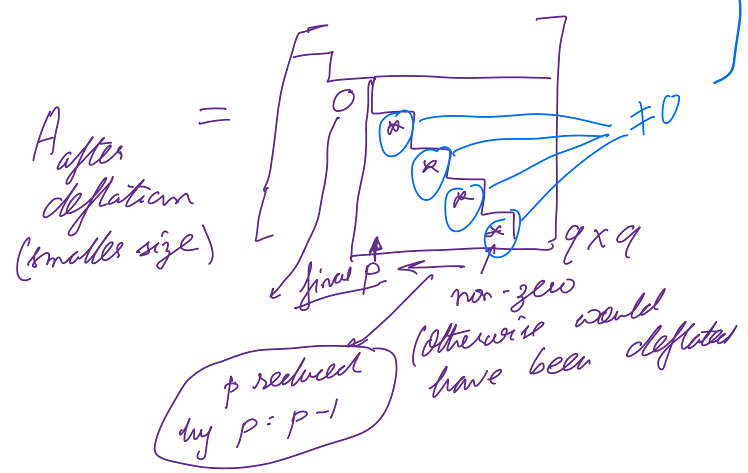
eigenvalue stored in $D[q]$

$A(q, q-1) = 0$

```
function gees!(A)
    n = size(A,1)
    if n==1
        return A[1,1]
    end
    D = zeros(Complex{Float64},n)

    # Tolerance for deflation
    tol = eps(Float64)

    q = n # Size of the matrix we are currently working with
    iter = 1 # Counter to detect convergence failure
    iter_per_evalue = 0 # Used to trigger an exceptional shift

    reduce_eps!(A, tol) # Zero out small entries

    while q > 0

        if iter > 10*n
            @''Code failed to converge''
        end

        deflation = true # Were we able to deflate the matrix?

        while deflation
            @''Test for deflation and record eigenvalues if
               converged''
        end

        # If q <= 2 we compute the eigenvalues at the next
        # iteration.
        if q >= 3
            @''Perform Francis QR step''
        end
    end
end
```

```
# Definition of @"Perform Francis QR step"
# Searching for the smallest unreduced sub-block
p = q
while p > 1 && A[p,p-1] != 0
    p -= 1
end

# If the unreduced sub-block has size 2 or less, we move on
# to the next iteration.
if q-p+1 >= 3
    B = A[p:q,p:q] # Extract sub-block
    exceptional_shift = ((iter_per_evalue%5) == 0 &&
                         iter_per_evalue>0)
    # Francis QR step
    gees_single_step!(B, exceptional_shift)
    # Reduce matrix
    reduce_eps!(B, tol)
    A[p:q,p:q] = B # Copy the resulting matrix back
    iter += 1 # Increment iteration counter
    iter_per_evalue += 1 # Counter for exceptional_shift
end
```

Say after possible deflation

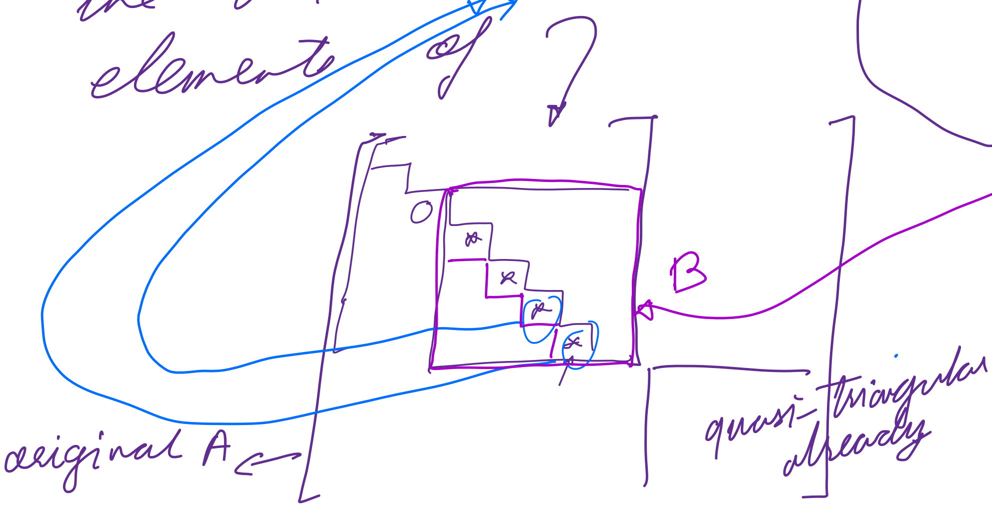


If $\underline{q-p+1}$ > 3 then ◁
do Francis step

Otherwise do not do Francis step. Deflate in the next step.

So after Francis step we do not reduce q.

The program returns to ◁ and does deflation if the Francis step was successful in reducing the bottom corner sub-diagonal elements of ?



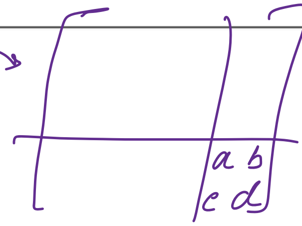

```
function gees!(A)
    n = size(A,1)
    if n==1
        return A[1,1]
    end
    D = zeros(Complex{Float64},n)

    # Tolerance for deflation
    tol = eps(Float64)

    q = n # Size of the matrix we are currently working with
    iter = 1 # Counter to detect convergence failure
    iter_per_evalue = 0 # Used to trigger an exceptional shift

    reduce_eps!(A, tol) # Zero out small entries

    while q > 0

        if iter > 10*n
            @''Code failed to converge''
        end

        deflation = true # Were we able to deflate the matrix?

        while deflation
            @''Test for deflation and record eigenvalues if
                converged''
```

```
        end

        # If q <= 2 we compute the eigenvalues at the next
        # iteration.
        if q >= 3
            @''Perform Francis QR step''
        end
    end
end
```

```
# Definition of @"Perform Francis QR step"
# Searching for the smallest unreduced sub-block
p = q
while p > 1 && A[p,p-1] != 0
    p -= 1
end

# If the unreduced sub-block has size 2 or less, we move on
# to the next iteration.
if q-p+1 >= 3
    B = A[p:q,p:q] # Extract sub-block
    exceptional_shift = ((iter_per_evalue%5) == 0 &&
                        iter_per_evalue>0)
    # Francis QR step
    gees_single_step!(B, exceptional_shift)
    # Reduce matrix
    reduce_eps!(B, tol)
    A[p:q,p:q] = B # Copy the resulting matrix back
    iter += 1 # Increment iteration counter
    iter_per_evalue += 1 # Counter for exceptional_shift
end
```

FRANCIS step

```
function double_shift_st(A)
    a = A[1,1]
    b = A[1,2]
    c = A[2,1]
    d = A[2,2]
    s = a+d         # sum
    t = a*d - b*c   # product
    return s,t
end
```



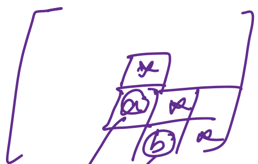

```
function gees_single_step!(A, exceptional_shift)
    n = size(A,1); tol = 0.01
    # This tolerance is used to test for early convergence of the last
    # 2x2 or 1x1 block.

    # Which shift should we apply?
    if    abs(A[n-1,n-2]) < tol * (abs(A[n-2,n-2])
                                 + abs(A[n-1,n-1])) ||
      ! ( abs(A[n,n-1])   < tol * (abs(A[n-1,n-1])
                                 + abs(A[n,n])) )
        # If either (double-shift test) == true
        # or (single-shift test) == false, do a double shift:
        s, t = double_shift_st(A[n-1:n,n-1:n])

    else # Single shift should be used
        s = 2*A[n,n]
        t = A[n,n]^2
    end

    if exceptional_shift
        @''Exceptional shift''
    end

    @''Apply the Francis QR step''
end
```

```
# Definition of @"Apply the Francis QR step"
# Assembling the first column
v = [ A[1,1]*A[1,1] + A[1,2]*A[2,1] - s*A[1,1] + t;
      A[2,1]*(A[1,1]+A[2,2]-s);
      A[2,1]*A[3,2] ]

beta, v = house(v)

apply_left_householder!( A, 1, 1, beta, v)
apply_right_householder!(A, min(4,n), 1, beta, v)

for k=2:n-1
    beta, v = house(A[k:min(k+2,n),k-1])
    apply_left_householder!( A, k, k-1, beta, v)
    apply_right_householder!(A, min(k+3,n), k, beta, v)
end
```

Whether to apply a double or a single-shift

Heuristic

1) Is the last 2x2 block converging



$A(n-1, n-2)$

$$abs(A(n-1, n-2)) < tol * [abs(A(n-2, n-2)) + abs(A(n-1, n-1))]$$

OR

$$abs(A(n, n-1)) > tol * (abs(A(n-1, n-1)) + abs(A(n, n)))$$

Idea: If a is big we should do single-shift
If b is small we should do single-shift
If a is small or b is large we do double-shift.

s, t computation for double shift

Using the old notation (from the notes)

$$Me_1 = \begin{bmatrix} x \\ y \\ z \\ 0 \\ \vdots \\ 0 \end{bmatrix} \left\{ \begin{bmatrix} h_{11} & h_{12} & \cdots \\ h_{21} & h_{22} & \cdots \\ 0 & h_{32} & \ddots \end{bmatrix} - a_1 I \right\} \left\{ \qquad \right\}$$

$$x = h_{11}^2 + h_{12}h_{21} - \overset{s}{(a_1 + a_2)}\, h_{11} + \overset{t}{a_1 a_2}$$

$$y = h_{21}(h_{11} + h_{22} - (a_1 + a_2))$$

$$z = h_{21} h_{32}$$

$$\left| \lambda I - \begin{bmatrix} a & b \\ c & d \end{bmatrix} \right| = (\lambda - a)(\lambda - d) - cb = \lambda^2 - \underbrace{(a+d)}_{s}\lambda + \underbrace{ad - cb}_{t}$$

Here $V = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$

s, t for single shift

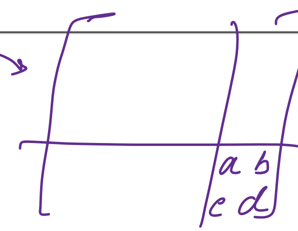
$s = 2d$, $t = d^2$

If $a_1 = a_2 \boxed{= d} \to$ current guess.

$x = h_{11}^2 + h_{12}h_{21} - (2d)\,h_{11} + d^2$

y $z = \ldots$

```
function double_shift_st(A)
    a = A[1,1]
    b = A[1,2]
    c = A[2,1]
    d = A[2,2]
    s = a+d          # sum
    t = a*d - b*c    # product
    return s,t
end
```

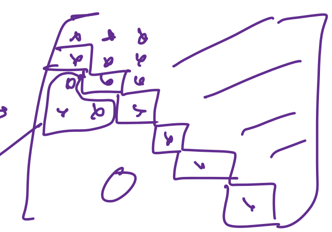


```julia
# Definition of @"Apply the Francis QR step"
# Assembling the first column
v = [ A[1,1]*A[1,1] + A[1,2]*A[2,1] - s*A[1,1] + t;
      A[2,1]*(A[1,1]+A[2,2]-s);
      A[2,1]*A[3,2] ]

beta, v = house(v)

apply_left_householder!( A, 1, 1, beta, v)
apply_right_householder!(A, min(4,n), 1, beta, v)

for k=2:n-1
    beta, v = house(A[k:min(k+2,n),k-1])
    apply_left_householder!( A, k, k-1, beta, v)
    apply_right_householder!(A, min(k+3,n), k, beta, v)
end
```

```julia
"""
    apply_right_householder!(A, row, col, beta, v)

Apply a Householder reflection to the right of matrix `A`.

`row` is the ending row index to apply the transform.
`col` is the starting column index. `beta` and `v` are the
parameters for the Householder reflection.
"""
function apply_right_householder!(A, row, col, beta, v)
    n = size(A,1)
    Av = zeros(n)
    lv = length(v)
    # Apply transform to the right
    # Av = beta * A * v
    for j=col:col+lv-1
        for i=1:row
            Av[i] += v[j-col+1] * A[i,j]
        end
    end
    for i=1:row
        Av[i] *= beta
    end
    # A - beta (Av) v^T
    for j=col:col+lv-1, i=1:row
        A[i,j] -= Av[i] * v[j-col+1]
    end
end
```

```julia
"""
Click to collapse the range._householder!(A, row, col, beta, v)

Apply a Householder reflection to the left of matrix `A`.

`row` is the starting row index to apply the transform.
`col` is the starting column index. `beta` and `v` are the
parameters for the Householder reflection.

"""
function apply_left_householder!(A, row, col, beta, v)
    n = size(A,1)
    vA = zeros(n)
    lv = length(v)
    # Apply transform to the left
    # vA = beta * v^T * A
    for j=col:n
        for i=row:row+lv-1
            vA[j] += v[i-row+1] * A[i,j]
        end
        vA[j] *= beta
    end
    # A - beta v (v^T A)
    for j=col:n, i=row:row+lv-1
        A[i,j] -= v[i-row+1] * vA[j]
    end
end
```

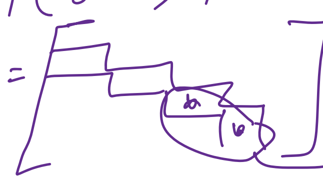
$P_0 H P_0 \rightarrow$



bulge

$\rightarrow P_{n-2} \cdots P_1 (P_0 H P_0) P_1 \cdots P_{n-2}$

$=$



hopefully low values

Q. Where did the $QR = H - \mu I$, $\bar{H} = RQ + \mu I$ step go?