1 QR razcep zgornje hessenbergove matrike

Martin Starič

QR razcep zgornje Hessenbergove matrike `H` je najbolj primeren s pomočjo Givensovih rotacij, saj zanj potrebujemo najmanjše število operacij. Givensove rotacije temeljijo na podlagi rotacije vektorja $v \in R^2$ s pomočjo rotacijske matrike $R = G(i, j, \theta)$. Ko rotacijsko matriko `R_i` pomnožimo z $H_i - 1$ in dobimo H_i (kjer i označuje v kateri iteraciji smo), bo drugi element, ki smo ga izbrali za v uničen ($= 0$). Ta postopek iteriramo dokler vseh poddiagonalnih elementov ne uničimo. QR razcep pa je nakoncu takšen, da je $H_n = R$ in zmnožek $R_1^T * R_2^T * ... * R_n^T = Q$

```
using Main.DN1
```

Sprva izberimo zgornje Hessenbergovo matriko

```
H = DN1.Hessenberg([2 7 2;
                    3 2 7;
                    0 1 4])
```

```
Main.DN1.ZgornjiHessenberg([2.0 7.0 2.0; 3.0 2.0 7.0; 0.0 1.0 4.0])
```

Nato izračunamo njen QR razcep s pomočjo Givensovih rotacij, saj je zaradi Hessebergove oblike uporaba Givensovih rotacij najbolj primerna in učinkovita metoda.

```
Q,R = DN1.qr(H)
```

```
(Main.DN1.Givens([0.5547001962252291 -0.9782400740023413; -0.83205029433784
37 -0.20747616156053694]), [3.6055512754639896 5.547001962252291 6.93375245
2815364; 0.0 4.8198308300986294 -1.3406151977757776; 0.0 0.0 -4.37330856612
8114])
```

1.1 QR iteracija

QR iteracija je postopek, kjer matriko A spreminjamo v zgornje trikotno tako, da izračunamo $A = QR$ in jo zamenjamo z $A = RQ$, to ponavljamo več iteracij in nazadnje preberemo diagonalne elemente matrike `A` - ki so lastne vrednosti matrike `A`. Sledi demonstracija te metode

```
lastnevrednosti,lastnivektorji = DN1.eigen(H,1000)
```

```
([7.770937597524546, 3.2439316280979136, -3.014869225622463], [0.7835968187
714317 -0.45907575948619295 0.41859941789671284; 0.600513267795882 0.386971
7342630519 -0.6997404462245693; 0.15924773408875592 0.7996888919658832 0.57
89109044179228])
```

Da rezultat primerjamo, si pomagamo s knjižnico LinearAlgebra, ki ima funkciji za izračun qr razcepa in lastnih vrednosti

```
using LinearAlgebra
mt =  [2 7 2;
       3 2 7;
       0 1 4]
QR = LinearAlgebra.qr(mt)
podatki = LinearAlgebra.eigen(mt)
podatki.values
```

```
3-element Vector{Float64}:
 -3.0148692256224603
  3.24393162809791
  7.770937597524555
```

Da preverimo čas potreben za izvedbo implementiranih funkcij si pomagamo z dekoratorjem @timed, oglejmo si časovno potratnost metode qr

```
dn_QR_Result = (@timed DN1.qr(H))
LA_QR_Result = (@timed LinearAlgebra.qr(mt))
dn_QR_Result.time - LA_QR_Result.time
```

```
-1.69e-5
```

Opazimo, da je naša funkcija malenkost hitrejša Sedaj preverimo še rezultate za računanje lastnih vrednosti

```
dn_eigen_Result = (@timed DN1.eigen(H,10))
LA_eigen_Result = (@timed LinearAlgebra.eigen(mt))
dn_eigen_Result.time - LA_eigen_Result.time

dn_eigen_Result2 = (@timed DN1.eigen(H,100))
dn_eigen_Result2.time - LA_eigen_Result.time
```

```
2.3299999999999997e-5
```

Izkaže se, da je vse odvisno od željene natančnosti, v kolikor želimo višjo natančnost metode, je naša implementirana funkcija z qr iteracijo počasnejša