Computerphysik Programmiertutorial 4b

Prof. Dr. Matteo Rizzi und Dr. Markus Schmitt - Institut für Theoretische Physik, Universität zu Köln

Github: https://github.com/markusschmitt/compphys2022

Inhalt dieses Notebooks: Array-Abstraktionen, LinearAlgebra: Matrix/Vektor-Produkte, spezielle Matrizen, Matrixinversion, besondere Funktionen von Matrizen und Vektoren, Eigenwertprobleme, Singulärwertzerlegung, QR-Zerlegung

Array-Abstraktionen (Array Comprehension)

Mit Array-Abstraktionen können Arrays oder andere iterierbare Datenstrukturen einfach "weiterverarbeitet" werden um daraus neue Arrays zu erzeugen.

Syntax:

`neues_array = [<Anweisung> for <Variable> in <iterierbare Datenstruktur> if <Bedingung>]`

Beispiel:

```
In [1]: arr = [2n for n in 1:10]

Out[1]: 10-element Vector{Int64}:
          2
          4
          6
          8
         10
         12
         14
         16
         18
         20
```

```
In [2]: arr = [2n for n in 1:10 if n%2==0]

Out[2]: 5-element Vector{Int64}:
          4
          8
         12
         16
         20
```

Lineare Algebra

Hier führen wir einen Teil der Funktionen des `LinearAlgebra` Pakets ein. Für einen vollständigen Überblick, siehe Dokumentation.

```
In [1]: using LinearAlgebra
```

Matrixprodukte und -transformationen

Wir beschaffen uns zunächst zwei 3×3 Matrizen

```
In [2]: A = reshape([1.0*n for n in 1:9], 3,3)

Out[2]: 3×3 Matrix{Float64}:
         1.0  4.0  7.0
         2.0  5.0  8.0
         3.0  6.0  9.0
```

```
In [3]: B=rand(3,3)

Out[3]: 3×3 Matrix{Float64}:
         0.866748  0.839678  0.655212
         0.862904  0.66954   0.418176
         0.492418  0.404241  0.975016
```

Das übliche Matrixprodukt wird durch den `*` Operator berechnet:

```
In [4]: A * B

Out[4]: 3×3 Matrix{Float64}:
          7.76529   6.34753   9.15302
          9.98736   8.26099  11.2014
         12.2094   10.1744   13.2498
```

Elementweise Multiplikation der Einträge erfolgt durch `.*`

```
In [5]: A .* B

Out[5]: 3×3 Matrix{Float64}:
         0.866748  3.35871  4.58648
         1.72581   3.3477   3.34541
         1.47726   2.42545  8.77514
```

Eine Matrix kann durch die Funktion `transpose` transponiert werden.

```
In [6]: transpose(A)

Out[6]: 3×3 transpose(::Matrix{Float64}) with eltype Float64:
         1.0  2.0  3.0
         4.0  5.0  6.0
         7.0  8.0  9.0
```

Durch Nachstellen von `'` wird die Matrix transponiert und komplex konjugiert:

```
In [8]: (1.0im*A)'

Out[8]: 3×3 adjoint(::Matrix{ComplexF64}) with eltype ComplexF64:
         0.0-1.0im  0.0-2.0im  0.0-3.0im
         0.0-4.0im  0.0-5.0im  0.0-6.0im
         0.0-7.0im  0.0-8.0im  0.0-9.0im
```

Vektorprodukte

Es gibt verschiedene Optionen das Skalarprodukt zweier Vektoren zu berechnen:

```
In [9]: v1=[1.0im,2.0,3.0]
        v2=[3.0,6.0,9.0]

Out[9]: 3-element Vector{Float64}:
         3.0
         6.0
         9.0
```

```
In [10]: dot(v1,v2)

Out[10]: 39.0 - 3.0im
```

```
In [11]: v1' * v2

Out[11]: 39.0 - 3.0im
```

Transponieren des ersten Vektors funktioniert ebenfalls bei reellen Zahlen, aber liefert bei komplexen Zahlen natürlich nicht das gewünschte Ergebnis:

```
In [12]: transpose(v1) * v2

Out[12]: 39.0 + 3.0im
```

Außerdem ist das Kreuzprodukt in der Funktion `cross` implementiert:

```
In [13]: cross(v1,v2)

Out[13]: 3-element Vector{ComplexF64}:
           0.0 + 0.0im
           9.0 - 9.0im
          -6.0 + 6.0im
```

Spezielle Matrizen

Spezielle Matrizen, die weitere Struktur haben, können besonders behandelt werden, z.B. aus Effizienzgründen. Die vollständige Liste besonderer Matrixformate ist in der Dokumentation zu finden.

Hier beschränken wir uns auf Diagonalmatrizen `Diagonal`:

```
In [14]: Diagonal([1,2,3])

Out[14]: 3×3 Diagonal{Int64, Vector{Int64}}:
          1  ⋅  ⋅
          ⋅  2  ⋅
          ⋅  ⋅  3
```

Matrixinversion

Das Inverse einer Matrix kann mit der Funktion `inv()` berechnet werden:

```
In [15]: M = rand(3,3)
         M_inv = inv(M)

Out[15]: 3×3 Matrix{Float64}:
           19.0886   -0.915527  -10.9617
            9.57293  -1.51403    -4.39773
          -19.3607    2.03031    11.1131
```

```
In [16]: M_inv * M

Out[16]: 3×3 Matrix{Float64}:
           1.0           1.77636e-15  0.0
           6.66134e-16   1.0          0.0
          -8.88178e-16  -1.77636e-15  1.0
```

Ein typisches Problem, das durch Matrixinversion gelöst werden kann, ist das Lösen eines linearen Gleichungssystems, z.B.

$$\begin{pmatrix} 1 & 2 & 3 \\ -3 & 2 & 5 \\ 0 & 6 & 23 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 42 \\ 13 \\ 1 \end{pmatrix}$$

Wir legen also entsprechend eine Matrix und einen Vektor an:

```
In [17]: A = [1.0 -3.0 0.0; 2.0 2.0 6.0; 3.0 5.0 23.0]
         b = [42.0, 13.0, 1.0]

Out[17]: 3-element Vector{Float64}:
          42.0
          13.0
           1.0
```

Und wir erhalten die Lösung als $\vec{x} = A^{-1}\vec{b}$

```
In [18]: x = inv(A) * b

Out[18]: 3-element Vector{Float64}:
          15.51
          -8.829999999999998
          -0.060000000000000456
```

"Probe Rechnen":

```
In [19]: A*x - b

Out[19]: 3-element Vector{Float64}:
          -7.105427357601002e-15
           0.0
          -7.771561172376096e-16
```

Übersichtlicheres "Probe Rechnen" mit `isapprox()`:

```
In [20]: isapprox(A*x,b)

Out[20]: true
```

Alternativ kannn das Gleichungssystem durch den Linksdivisionsoperator `\` gelöst werden. Da wir uns am Ende nur für das Produkt $A^{-1}\vec{b}$ interessieren, kann es Julia in diesem Fall vermeiden, die Matrix explizit zu invertieren, und stattdessen einen effizienteren Algorithmus verwenden:

```
In [21]: x1 = A\b

Out[21]: 3-element Vector{Float64}:
          15.509999999999998
          -8.829999999999997
          -0.0600000000000006
```

```
In [22]: isapprox(x,x1)

Out[22]: true
```

Besondere Funktionen von Matrizen und Vektoren

Die üblichen Funktionen, die wir auf Matrizen und Vektoren ausrechnen wollen, sind implementiert:

Norm `norm()` (standardmäßig L^2 Norm):

```
In [23]: norm(x)

Out[23]: 17.84748161506267
```

```
In [24]: norm(A)

Out[24]: 24.839484696748443
```

L^p Norm durch ein weiteres Argument:

```
In [25]: norm(x,7)

Out[25]: 15.552596726447272
```

Determinante `det()`:

```
In [26]: det(A)

Out[26]: 100.0
```

Spur `tr()`:

```
In [27]: tr(A)

Out[27]: 26.0
```

Rang `rank()`:

```
In [28]: rank(A)

Out[28]: 3
```

Eigenwertprobleme

Durch Eigenzerlegung einer Matrix M finden wir Eigenwerte $\vec{\lambda}$ und eine Matrix V, deren Spalten den Eigenvektoren entsprechen, so dass

$$M = V \cdot \mathrm{diag}\big(\vec{\lambda}\big) \cdot V^{-1}$$

Auch das ist in Julia ein Einzeiler:

```
In [29]: lambda, V = eigen(A)

Out[29]: Eigen{ComplexF64, ComplexF64, Matrix{ComplexF64}, Vector{ComplexF64}}
         values:
         3-element Vector{ComplexF64}:
          0.8846859357719979 - 1.8287533881621232im
          0.8846859357719979 + 1.8287533881621232im
           24.23062812845602 + 0.0im
         vectors:
         3×3 Matrix{ComplexF64}:
           -0.841368-0.0im        -0.841368+0.0im       0.0332714+0.0im
          -0.0323405-0.512885im  -0.0323405+0.512885im  -0.257639+0.0im
            0.130144+0.105195im    0.130144-0.105195im  -0.965668+0.0im
```

```
In [30]: isapprox(A, V * Diagonal(lambda) * inv(V))

Out[30]: true
```

Es können auch jeweils nur Eigenwerte oder nur Eigenvektoren berechnet werden:

```
In [31]: lambda = eigvals(A)

Out[31]: 3-element Vector{ComplexF64}:
          0.8846859357719979 - 1.8287533881621232im
          0.8846859357719979 + 1.8287533881621232im
           24.23062812845602 + 0.0im
```

```
In [32]: V = eigvecs(A)

Out[32]: 3×3 Matrix{ComplexF64}:
           -0.841368-0.0im        -0.841368+0.0im       0.0332714+0.0im
          -0.0323405-0.512885im  -0.0323405+0.512885im  -0.257639+0.0im
            0.130144+0.105195im    0.130144-0.105195im  -0.965668+0.0im
```

Singulärwertzerlegung

Durch Eigenzerlegung einer Matrix M finden wir Singulärwerte $\vec{\sigma}$ und zwei unitäre Matrizen U und V, so dass

$$M = U \cdot \mathrm{diag}\big(\vec{\sigma}\big) \cdot V^{\dagger}$$

Auch das ist in Julia ein Einzeiler:

```
In [33]: U, sigma, V = svd(A)

Out[33]: SVD{Float64, Float64, Matrix{Float64}}
         U factor:
         3×3 Matrix{Float64}:
           0.0212047   0.996202   -0.0844512
          -0.264569   -0.0758676  -0.961378
          -0.964134    0.0427289   0.261955
         singular values:
         3-element Vector{Float64}:
          24.60738043552128
           3.128885785804746
           1.2988078537310093
         Vt factor:
         3×3 Matrix{Float64}:
          -0.138183  -0.219992  -0.965665
           0.310863  -0.93538    0.168609
          -0.940356  -0.27689    0.197641
```

```
In [34]: U' * U

Out[34]: 3×3 Matrix{Float64}:
           1.0           2.42861e-16  -1.11022e-16
           2.42861e-16   1.0          -9.54098e-17
          -1.11022e-16  -9.54098e-17   1.0
```

```
In [35]: isapprox(A, U * Diagonal(sigma) * V')

Out[35]: true
```

QR-Zerlegung

Durch Eigenzerlegung einer Matrix M finden wir eine unitäre Matrix Q und eine obere Dreiecksmatrix R, so dass

$$M = Q \cdot R$$

Auch das ist in Julia ein Einzeiler:

```
In [36]: Q,R = qr(A)

Out[36]: LinearAlgebra.QRCompactWY{Float64, Matrix{Float64}}
         Q factor:
         3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}}:
          -0.267261   0.93306     0.240772
          -0.534522   0.0643489  -0.842701
          -0.801784  -0.353919    0.481543
         R factor:
         3×3 Matrix{Float64}:
          -3.74166  -4.27618  -21.6482
           0.0      -4.44008   -7.75405
           0.0       0.0        6.01929
```

```
In [37]: isapprox(A, Q*R)

Out[37]: true
```

```
In [38]: Q' * Q

Out[38]: 3×3 Matrix{Float64}:
          1.0           2.77556e-16   1.11022e-16
          2.77556e-16   1.0          -1.11022e-16
          1.11022e-16  -1.11022e-16   1.0
```

```
In [39]: R

Out[39]: 3×3 Matrix{Float64}:
          -3.74166  -4.27618  -21.6482
           0.0      -4.44008   -7.75405
           0.0       0.0        6.01929
```