1 Das *Linear Algebra*-Paket: eine kurze Auswahl

- zusätzliche Matrix-Typen
  - Subtypen von `AbstractMatrix`: genauso verwendbar, wie andere Matrizen
  - `Tridiagonal`
  - `SymTridiagonal`
  - `Symmetric`
  - `UpperTriangular`
  - …
- Arithmetik: Matrixmultiplikation, `inv`, `det`, `exp`
- Lineare Gleichungssysteme: `\`
- Matrixfaktorisierungen
  - `LU`
  - `QR`
  - `Cholesky`
  - `SVD`
  - …
- Eigenwerte/-vektoren
  - `eigen`, `eigvals`, `eigvecs`
- Zugriff auf BLAS/LAPACK-Funktionen

1.1 Matrixtypen

[1]:
```julia
using LinearAlgebra
```

[2]:
```julia
A = SymTridiagonal(fill(1.0, 4), fill(-0.3, 3))
```

[2]:
```
4×4 SymTridiagonal{Float64, Vector{Float64}}:
  1.0  -0.3    ⋅     ⋅
 -0.3   1.0  -0.3    ⋅
   ⋅   -0.3   1.0  -0.3
   ⋅     ⋅   -0.3   1.0
```

[3]:
```julia
B = UpperTriangular(A)
```

[3]:
```
4×4 UpperTriangular{Float64, SymTridiagonal{Float64, Vector{Float64}}}:
 1.0  -0.3   0.0   0.0
  ⋅    1.0  -0.3   0.0
  ⋅     ⋅    1.0  -0.3
  ⋅     ⋅     ⋅    1.0
```

[4]:
```julia
A + B
```

[4]:
```
4×4 Matrix{Float64}:
  2.0  -0.6   0.0   0.0
 -0.3   2.0  -0.6   0.0
  0.0  -0.3   2.0  -0.6
  0.0   0.0  -0.3   2.0
```

1.1.1 Einheitsmatrix `I`

`I` bezeichnet eine Einheitsmatrix (quadratisch, Diagonalelemente = 1, alle anderen = 0) in der jeweils erforderlichen Größe

[5]:
```julia
A + 4I
```

[5]:
```
4×4 SymTridiagonal{Float64, Vector{Float64}}:
  5.0  -0.3    ⋅     ⋅
```

```
 -0.3   5.0  -0.3    ⋅
   ⋅   -0.3   5.0  -0.3
   ⋅     ⋅   -0.3   5.0
```

1.2 Faktorisierungen

1.2.1 LU-Faktorisierung mit Zeilenpivoting

(‘Lower/Upper triangular matrix’, im Deutschen auch oft “LR-Zerlegung” für “Linke/Rechte Dreiecksmatrix”)

[6]:
```
A = [0 22 1.
    -1 2 3
    77 18 19]
```

[6]:
```
3×3 Matrix{Float64}:
  0.0  22.0   1.0
 -1.0   2.0   3.0
 77.0  18.0  19.0
```

[7]:
```
# Faktorisierungen geben eine spezielle Struktur zurück, die die Matrixfaktoren und weitere
# Informationen enthalten:

Af = lu(A);

@show Af.L  Af.U Af.p;
```

```
Af.L = [1.0 0.0 0.0; 0.0 1.0 0.0; -0.012987012987012988 0.10153482880755607 1.0]
Af.U = [77.0 18.0 19.0; 0.0 22.0 1.0; 0.0 0.0 3.145218417945691]
Af.p = [3, 1, 2]
```

[8]:
```
# man kann auch gleich auf der linken Seite ein entsprechendes Tupel verwenden

l,u,p = lu(A)

l
```

[8]:
```
3×3 Matrix{Float64}:
  1.0        0.0       0.0
  0.0        1.0       0.0
 -0.012987   0.101535  1.0
```

[9]:
```
u
```

[9]:
```
3×3 Matrix{Float64}:
 77.0  18.0  19.0
  0.0  22.0   1.0
  0.0   0.0   3.14522
```

[10]:
```
p
```

[10]:
```
3-element Vector{Int64}:
 3
 1
 2
```

Der Permutationsvektor `p` zeigt an, wie die Zeilen der Matrix permutiert wurden:

[11]:
```
A[p, :]   # 3. Zeile, 1. Zeile, 2. Zeile von A
```

[11]:
```
3×3 Matrix{Float64}:
 77.0  18.0  19.0
  0.0  22.0   1.0
 -1.0   2.0   3.0
```

Es gilt:

$$L \cdot U = PA$$

[12]:
```julia
l * u - A[p,:]
```

[12]:
```
3×3 Matrix{Float64}:
 0.0   0.0          0.0
 0.0   0.0          0.0
 0.0  -2.22045e-16  0.0
```

1.2.2 QR-Zerlegung

[13]:
```julia
q, r = qr(A);

q
```

[13]:
```
3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}}:
  0.0        -0.994886    -0.101007
  0.0129859  -0.100999     0.994802
 -0.999916   -0.00131167   0.0129195
```

[14]:
```julia
r
```

[14]:
```
3×3 Matrix{Float64}:
 -77.0065  -17.9725  -18.9594
   0.0     -22.1131   -1.3228
   0.0       0.0       3.12887
```

1.2.3 Singular value decomposition

[15]:
```julia
u, s, vt = svd(A);

s
```

[15]:
```
3-element Vector{Float64}:
 81.49863518679489
 21.487573868706654
  3.0424713518113586
```

[16]:
```julia
u
```

[16]:
```
3×3 Matrix{Float64}:
 -0.0672717   -0.992532   -0.101754
 -0.00288554  -0.101791    0.994802
 -0.997731     0.0672156   0.00398368
```

[17]:
```julia
vt
```

[17]:
```
3×3 adjoint(::Matrix{Float64}) with eltype Float64:
 -0.942621   0.245602     -0.226151
 -0.238592  -0.96937      -0.0582696
 -0.233535  -0.000968442   0.972348
```

[18]:
```julia
evalues, evectors = eigen(A)
evalues
```

[18]:
```
3-element Vector{ComplexF64}:
 -4.259744130930172 - 12.741487891217805im
 -4.259744130930172 + 12.741487891217805im
 29.519488261860335 + 0.0im
```

[19]:
```
evectors
```

[19]:
```
3×3 Matrix{ComplexF64}:
 -0.253921-0.195951im  -0.253921+0.195951im  -0.110787+0.0im
 -0.106268+0.185001im  -0.106268-0.185001im  -0.103725+0.0im
  0.922827-0.0im        0.922827+0.0im       -0.988417+0.0im
```

1.2.4 Lineare Gleichungssysteme

Das lineare Gleichungssystem

$$Ax = b$$

kann man in Julia einfach lösen mit

`x = A\b`

[20]:
```
b = [2,3,4]

A\b
```

[20]:
```
3-element Vector{Float64}:
 -0.18318318318318316
  0.04973723723723724
  0.9057807807807807
```

Dabei wird eine geeignete Matrixfaktorisierung vorgenommen (meist LU). Wenn man Lösungen zu mehreren rechten Seiten $b_1, b_2, \ldots$ benötigt, sollte man die Faktorisierung nur einmal durchführen:

[21]:
```
Af = factorize(A)
```

[21]:
```
LU{Float64, Matrix{Float64}}
L factor:
3×3 Matrix{Float64}:
  1.0       0.0       0.0
  0.0       1.0       0.0
 -0.012987  0.101535  1.0
U factor:
3×3 Matrix{Float64}:
 77.0  18.0  19.0
  0.0  22.0   1.0
  0.0   0.0   3.14522
```

[22]:
```
Af\b
```

[22]:
```
3-element Vector{Float64}:
 -0.18318318318318316
  0.04973723723723724
  0.9057807807807807
```

[23]:
```
Af\[5,7,9]
```

[23]:
```
3-element Vector{Float64}:
 -0.43243243243243246
  0.13175675675675677
  2.1013513513513513
```

[ ]:
```

```