Computerphysik Programmiertutorial 5a

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

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

Inhalt dieses Notebooks: Jonglieren mit Arrays, dünne Matrizen, [E] Zusammengesetzte Datentypen: `struct` s

Jonglieren mit Arrays

```
In [1]:
```
```
A = collect(1:16)
```
```
Out[1]:
```
```
16-element Vector{Int64}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
 16
```

Dimensionen ändern: `reshape`

```
In [2]:
```
```
reshape(A,4,4)
```
```
Out[2]:
```
```
4×4 Matrix{Int64}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16
```
```
In [3]:
```
```
reshape(A,:,2)
```
```
Out[3]:
```
```
8×2 Matrix{Int64}:
  1  9
  2  10
  3  11
  4  12
  5  13
  6  14
  7  15
  8  16
```
```
In [4]:
```
```
A = reshape(A, 4, :)
```
```
Out[4]:
```
```
4×4 Matrix{Int64}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16
```

Eine einzelne Spalte extrahieren

```
In [5]:
```
```
v=A[:,3]
```
```
Out[5]:
```
```
4-element Vector{Int64}:
  9
 10
 11
 12
```
```
In [6]:
```
```
v
```
```
Out[6]:
```
```
4-element Vector{Int64}:
  9
 10
 11
 12
```

Eine einzelne Zeile extrahieren

```
In [7]:
```
```
A[2,:]
```
```
Out[7]:
```
```
4-element Vector{Int64}:
  2
  6
 10
 14
```

Eine Untermatrix extrahieren

```
In [8]:
```
```
A[2:4,2:3]
```
```
Out[8]:
```
```
3×2 Matrix{Int64}:
  6  10
  7  11
  8  12
```
```
In [9]:
```
```
A[[2,4],[1,3]]
```
```
Out[9]:
```
```
2×2 Matrix{Int64}:
  2  10
  4  12
```

Indizes vom Ende zählen:

```
In [10]:
```
```
A[:,1:end-1]
```
```
Out[10]:
```
```
4×3 Matrix{Int64}:
 1  5   9
 2  6  10
 3  7  11
 4  8  12
```

Dünne Matrizen

Matrizen, die in physikalischen Anwendungen interessant sind, haben oft viel Struktur. In verschiedenen Anwendungen tauchen *z.B.* Matrizen auf, die nur wenige von Null verschiedene Einträge haben — sogenannte **dünn besetzte Matrizen**. In solchen Fällen ist es günstig nicht die komplette Matrix im Speicher abzulegen, sondern nur die wenigen Einträge, die nicht Null sind.

Beispiel:

$$M = \begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 7 & 3 \\ 0 & 4 & 0 & 1 \end{pmatrix}$$

Um solche Matrizen effizient zu speichern gibt es verschiedene Datenstrukturen. Die Idee ist immer, dass es reicht die Werte der nicht-Null Einträge zu speichern, sowie ihre jeweilige Position in der Matrix.

Eine Möglichkeit ist das **Compressed Sparse Column (CSC)** Format. In diesem Format werden drei Arrays verwendet um die dünne Matrix zu Beschreiben:

1. Das erste Array enthält die **Werte** der nicht-Null Matrixeinträge entsprechend ihrer spaltenweisle Reihenfolge: `values` = [5, 8, 4, 7, 3, 1]
2. Das zweite Array enthält zugehörigen **Zeilenindizes** in der gleichen Reihenfolge: `row_idx` = [1, 2, 4, 3, 3, 4]
3. Das dritte Array enthält die sogenannten **Spaltenzeiger**. Die Spaltenzeiger sind die Indizes des ersten Eintrags jeder Spalte im `values` bzw. `row_idx` Array. Das Array enthält außerdem immer einen zusätzlichen letzten Eintrag, der der Zahl der Einträge in `values` plus 1 entspricht: `col_ptr` = [1, 2, 4, 5, 7]

Außerdem speichern wir die Dimension der Matrix, also m und n für eine $m \times n$-Matrix.

Eine solche Datenstruktur können wir zum Beispiel als Dictionary implementieren:

```
In [11]:
```
```
M = Dict(
            "values" => float([5, 8 , 4, 7 , 3, 1]),
            "row_idx" => [1, 2, 4, 3, 3, 4],
            "col_ptr" => [1, 2, 4, 5, 7],
            "n_rows" => 4,
            "n_cols" => 4
        )
```
```
Out[11]:
```
```
Dict{String, Any} with 5 entries:
  "values"  => [5.0, 8.0, 4.0, 7.0, 3.0, 1.0]
  "col_ptr" => [1, 2, 4, 5, 7]
  "n_cols"  => 4
  "row_idx" => [1, 2, 4, 3, 3, 4]
  "n_rows"  => 4
```

Um die Funktionsweise zu illustrieren, implementieren wir ein Matrix-Vektor-Produkt für unsere dünne Matrix:

```
In [13]:
```
```
function sparse_mvp(A, x)
    # Array für Ergebnis anlegen
    res = zeros(A["n_rows"])

    # nur Übersichtlichkeit
    col_ptr = A["col_ptr"]
    row_idx = A["row_idx"]
    values = A["values"]

    # Schleife für das Matrix-Vektor-Produkt
    for j in 1:(size(col_ptr)[1]-1)
        col_range = col_ptr[j]:col_ptr[j+1]-1 # Bereich der aktuellen Spalte in values Array
        row_indices = row_idx[col_range]      # Zeilen-Indizes, die zur aktuellen Spalte gehören

        res[row_indices] += values[col_range] * x[j]
    end

    return res
end
```
```
Out[13]:
```
```
sparse_mvp (generic function with 1 method)
```
```
In [14]:
```
```
mein_vektor = [1, 0, 3, 4]

sparse_mvp(M, mein_vektor)
```
```
Out[14]:
```
```
4-element Vector{Float64}:
  5.0
  8.0
 33.0
  4.0
```
```
In [15]:
```
```
dense_M = float([5 0 0 0; 0 8 0 0; 0 0 7 3; 0 4 0 1])

dense_M * mein_vektor
```
```
Out[15]:
```
```
4-element Vector{Float64}:
  5.0
  8.0
 33.0
  4.0
```

Julia stellt standardmäßig eine Datenstruktur für dünne Matrizen im CSC-Format bereit (im Standardpaket `SparseArrays`, Dokumentation):

```
In [17]:
```
```
using SparseArrays
```

Dünne Matrizen können damit auf unterschiedliche Weise erzeugt werden.

Entweder aus einer dichten Matrix:

```
In [18]:
```
```
sparse_M1 = sparse(dense_M)
```
```
Out[18]:
```
```
4×4 SparseMatrixCSC{Float64, Int64} with 6 stored entries:
 5.0   ⋅    ⋅    ⋅
  ⋅   8.0   ⋅    ⋅
  ⋅    ⋅   7.0  3.0
  ⋅   4.0   ⋅   1.0
```

Oder durch Übergeben der Zeileninidizes, Spalteninidizes, und Werte aller Einträge

```
In [19]:
```
```
sparse_M2 = sparse([1,2,3,3,4,4], [1,2,3,4,2,4], float([5,8,7,3,4,1]))
```
```
Out[19]:
```
```
4×4 SparseMatrixCSC{Float64, Int64} with 6 stored entries:
 5.0   ⋅    ⋅    ⋅
  ⋅   8.0   ⋅    ⋅
  ⋅    ⋅   7.0  3.0
  ⋅   4.0   ⋅   1.0
```

Genau wie bei dichten Matrizen können wir einzelne Einträge auslesen ...

```
In [20]:
```
```
sparse_M2[3,4]
```
```
Out[20]:
```
```
3.0
```

... und ändern

```
In [21]:
```
```
sparse_M2[1,3] = pi
sparse_M2
```
```
Out[21]:
```
```
4×4 SparseMatrixCSC{Float64, Int64} with 7 stored entries:
 5.0   ⋅    3.14159   ⋅
  ⋅   8.0    ⋅        ⋅
  ⋅    ⋅     7.0     3.0
  ⋅   4.0    ⋅       1.0
```

Das Matrix-Vektor-Produkt ist mit dieser Datenstruktur auch bereits implementiert:

```
In [22]:
```
```
sparse_M1 * mein_vektor
```
```
Out[22]:
```
```
4-element Vector{Float64}:
  5.0
  8.0
 33.0
  4.0
```

Genauso gibt es dünne Vektoren:

```
In [23]:
```
```
sparsevec(mein_vektor)
```
```
Out[23]:
```
```
4-element SparseVector{Int64, Int64} with 3 stored entries:
  [1]  =  1
  [3]  =  3
  [4]  =  4
```

[E] Zusammengesetzte Datentypen: `struct` s

Julia bietet die Möglichkeit eigene Zusammengesetzte Datentypen als `struct` s zu definieren (Dokumentation).

```
In [24]:
```
```
struct MySparse
    n_rows::Int
    n_cols::Int
    values::Vector{Float64}
    row_idx::Vector{Int}
    col_ptr::Vector{Int}
end
```

```
In [25]:
```
```
mat = MySparse(4, 4, float([5, 8 ,4 , 7, 3, 1]), [1, 2, 4, 3, 3, 4], [1, 2, 4, 5, 7])
```
```
Out[25]:
```
```
MySparse(4, 4, [5.0, 8.0, 4.0, 7.0, 3.0, 1.0], [1, 2, 4, 3, 3, 4], [1, 2, 4, 5, 7])
```

Unser neuer zusammengesetzter Datentyp ist Julia jetzt bekannt:

```
In [26]:
```
```
typeof(mat)
```
```
Out[26]:
```
```
MySparse
```

Auf die einzelnen Felder können wir mit `.<Feldname>` zugreifen:

```
In [27]:
```
```
mat.n_rows
```
```
Out[27]:
```
```
4
```
```
In [28]:
```
```
mat.values
```
```
Out[28]:
```
```
6-element Vector{Float64}:
 5.0
 8.0
 4.0
 7.0
 3.0
 1.0
```

Damit können wir unser dünnes Matrix-Vektor-Produkt umschreiben (*multiple dispatch*):

```
In [29]:
```
```
function sparse_mvp(A::MySparse, x)
    # Array für Ergebnis anlegen
    res = zeros(A.n_rows)

    # Schleife für das Matrix-Vektor-Produkt
    for j in 1:(size(A.col_ptr)[1]-1)
        col_range = A.col_ptr[j]:A.col_ptr[j+1]-1 # Bereich der aktuellen Spalte in values Array
        row_indices = A.row_idx[col_range]      # Zeilen-Indizes, die zur aktuellen Spalte gehören

        res[row_indices] += A.values[col_range] * x[j]
    end

    return res
end

sparse_mvp(mat, mein_vektor)
```
```
Out[29]:
```
```
4-element Vector{Float64}:
  5.0
  8.0
 33.0
  4.0
```

Hinweis: `struct` s sind immutable:

```
In [30]:
```
```
mat.n_rows = 5
```
```
ERROR: setfield!: immutable struct of type MySparse cannot be changed

Stacktrace:
 [1] setproperty!(x::MySparse, f::Symbol, v::Int64)
   @ Base ./Base.jl:43
 [2] top-level scope
   @ In[30]:1
 [3] eval
   @ ./boot.jl:373 [inlined]
 [4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
   @ Base ./loading.jl:1196
```

Das kann mit `mutable struct` s umgangen werden:

```
In [31]:
```
```
mutable struct MyMutableSparse
    n_rows::Int
    n_cols::Int
    values::Vector{Float64}
    row_idx::Vector{Int}
    col_ptr::Vector{Int}
end
```

```
In [32]:
```
```
mat_mutable = MyMutableSparse(4, 4, float([5, 8, 4, 7, 3, 1]), [1, 2, 4, 3, 3, 4], [1, 2, 4, 5, 7])

mat_mutable.n_rows=5
```
```
Out[32]:
```
```
5
```