

Computerphysik Programmiertutorial 6

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

ILIAS: https://www.ilias.uni-koeln.de/iliass/goto_uk_crs_3862489.html

GitHub: <https://github.com/markusschmitt/compphys2021>

Inhalt dieses Notebooks: Jonglieren mit Arrays, Matrix-Diagonalisierung (Anwendungsbeispiel), 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]: 4x4 Matrix{Int64}:
 1  5  9 13
 2  6 10 14
 3  7 11 15
 4  8 12 16

In [3]: reshape(A, 4, 2)

Out[3]: 8x2 Matrix{Int64}:
 1  2
 2 10
 3 11
 4 12
 5 13
 6 14
 7 15
 8 16

In [6]: A = reshape(A, 4, 4)

Out[6]: 4x4 Matrix{Int64}:
 1  5  9 13
 2  6 10 14
 3  7 11 15
 4  8 12 16

Eine einzelne Spalte extrahieren

In [8]: v=A[1,3]

Out[8]: 4-element Vector{Int64}:
 9
10
11
12

In [9]: v

Out[9]: 4-element Vector{Int64}:
 9
10
11
12

Eine einzelne Zeile extrahieren

In [10]: A[2,:]

Out[10]: 4-element Vector{Int64}:
 2
 6
10
14

Eine Untermatrix extrahieren

In [11]: A[2:4,2:3]

Out[11]: 3x2 Matrix{Int64}:
 6 10
 7 11
 8 12

In [12]: A[[2,4],[1,3]]

Out[12]: 2x2 Matrix{Int64}:
 2 10
 4 12

Indizes vom Ende zählen:

In [14]: A[1,1:end-1]

Out[14]: 4x3 Matrix{Int64}:
 1  5  9
 2  6 10
 3  7 11
 4  8 12
```

Anwendungsbeispiel Matrix-Diagonalisierung

Nicht-lineare Funktionen von Matrizen

Nicht-lineare Funktionen von Matrizen wie z.B. $\exp(\cdot)$ oder $\log(\cdot)$ sind über die entsprechende Taylorreihe definiert. So ist z.B.

$$\exp(A) = \sum_n \frac{1}{n!} A^n$$

Wenn A diagonalisierbar ist, können wir das Berechnen hoher Potenzen der Matrix vermeiden. Mit der Zerlegung $A = VDV^{-1}$ bekommen wir

$$A^n = (VDV^{-1})^n = VD^nV^{-1}$$

wobei D die Diagonalmatrix mit den Eigenwerten d_i auf der Diagonalen ist. Dadurch vereinfacht sich das Problem, denn das Potenzieren einer Diagonalmatrix entspricht dem Potenzieren der Diagonaleinträge. Durch Einsetzen in die Taylorreihe bekommen wir

$$\exp(A) = \sum_n \frac{1}{n!} V D^n V^{-1} = V \left(\sum_n \frac{1}{n!} D^n \right) V^{-1} = V \begin{pmatrix} \exp(d_1) & & 0 \\ & \ddots & \\ 0 & & \exp(d_N) \end{pmatrix} V^{-1}$$

Diese Konstruktion ist immer Möglich, wenn eine Funktion als Taylorreihe geschrieben werden kann.

Beispiel aus der Quantenmechanik

Die zeitabhängige Schrödingergleichung

$$-i \frac{d}{dt} \psi = H \psi$$

hat die formale Lösung

$$\psi(t) = \underbrace{\exp(-iHt)}_{=U_H(t)} \psi(t=0)$$

Dabei ist der Hamilton-Operator H ein linearer Operator auf dem Hilbertraum \mathcal{H} , der die Wellenfunktion ψ beherbergt. Falls \mathcal{H} endlich-dimensional ist, ist H also eine Matrix. Ein sehr grundlegendes Beispiel ist ein magnetisches Teilchen, das sich in einem externen Magnetfeld befindet. Im einfachsten Fall ist der Hilbertraum zweidimensional, also $\psi \in \mathbb{C}^2$. Wenn ein Magnetfeld in Richtung

$$\vec{B} = B \begin{pmatrix} \cos \phi \sin \theta \\ \sin \phi \sin \theta \\ \cos \theta \end{pmatrix}$$

angelegt wird, ist die zugehörige Schrödingergleichung

$$-i \frac{d}{dt} \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix} = \begin{pmatrix} B \cos \theta & B(\cos \phi - i \sin \phi) \sin \theta \\ B(\cos \phi + i \sin \phi) \sin \theta & -B \cos \theta \end{pmatrix} \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix}$$

```
In [15]: using LinearAlgebra
function U(H, t)
    E, V = eigen(H)
    return V * diagm(exp.(-1.0im * t * E)) * inv(V)
end

Out[15]: U (generic function with 1 method)

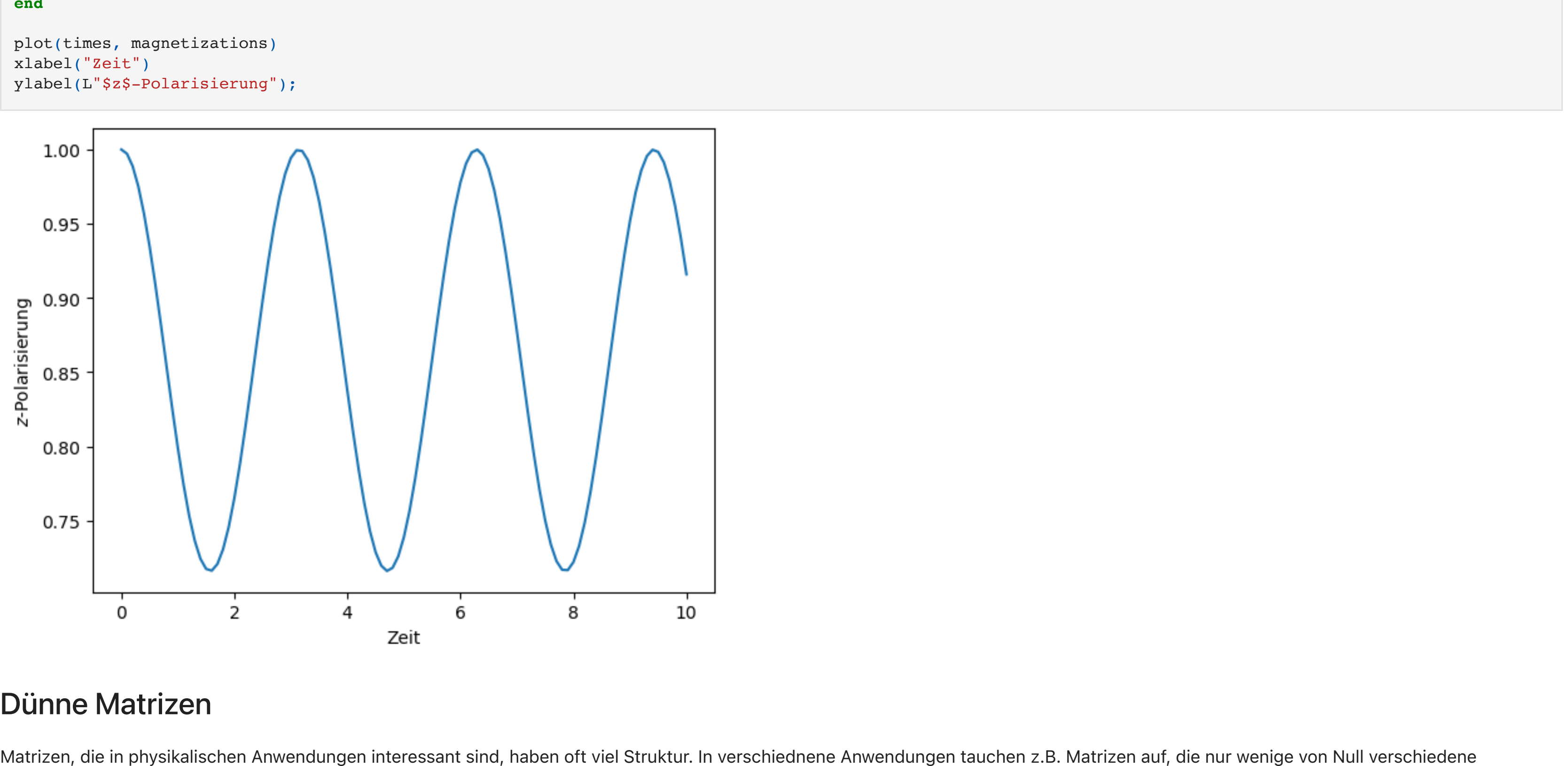
In [16]: phi=0.2*pi
theta=0.123*pi
B = 1.0

H = B * [cos(theta) (cos(phi)-1.0im*sin(phi))*sin(theta); (cos(phi)+1.0im*sin(phi))*sin(theta) -cos(theta)]

psi_0 = [1.0, 0.0]
U(H,4.3)*psi_0

Out[16]: 2-element Vector{ComplexF64}:
 -0.4007991720789755 + 0.84861312946132061im
 -0.2029483654737684 + 0.2793344461027271051im
```

Nach den Regeln der Quantenmechanik ist die Polarisierung unseres magnetischen Teilchens in z-Richtung gegeben durch $M_z = |\psi_1|^2 - |\psi_2|^2$. Die Zeitentwicklung dieser Polarisierung können wir uns nun anschauen:



Dünne Matrizen

Matrizen, die in physikalischen Anwendungen interessant sind, haben oft viel Struktur. In verschiedene 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:

- Das erste Array enthält die **Werte** der nicht-Null Matrixeinträge entsprechend ihrer *spaltenweise* Reihenfolge:
values = [5, 8, 4, 7, 3, 1]
- Das zweite Array enthält zugehörigen **Zeilenindizes** in der gleichen Reihenfolge:
row_idx = [1, 2, 4, 3, 3, 4]
- 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 [18]: 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[18]: 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 [19]: function sparse_mv(A, x)
    # Array für Ergebnis anlegen
    res = zeros(A["n_rows"])

    # zur 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 im 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[19]: sparse_mv (generic function with 1 method)

In [20]: mein_vektor = [1, 0, 3, 4]

sparse_mv(M, mein_vektor)

Out[20]: 4-element Vector{Float64}:
 5.0
 0.0
33.0
 4.0

In [21]: dense_M = float{[5 0 0 0; 0 8 0 0; 0 0 7 3; 0 4 0 1]}

dense_M * mein_vektor

Out[21]: 4-element Vector{Float64}:
 5.0
 0.0
33.0
 4.0
```

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

```
In [22]: using SparseArrays

Dünne Matrizen können damit auf unterschiedliche Weise erzeugt werden.

Entweder aus einer dichten Matrix:

In [23]: sparse_M1 = sparse(dense_M)

Out[23]: 4x4 SparseMatrixCSC{Float64, Int64} with 6 stored entries:
 5.0  .  .
 .  8.0  .
 .  .  7.0  3.0
 .  4.0  .  1.0

Oder durch Übergeben der Zeilenindizes, Spaltenindizes, und Werte aller Einträge

In [25]: sparse_M2 = sparse{[1,2,3,4,4], [1,2,3,4,2,4], float{[5,8,7,3,4,1]}}

Out[25]: 4x4 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 [26]: sparse_M2[3,4]

Out[26]: 3.0

... und ändern

In [27]: sparse_M2[1,3] = pi
sparse_M2

Out[27]: 4x4 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 [28]: sparse_M1 * mein_vektor

Out[28]: 4-element Vector{Float64}:
 5.0
 0.0
33.0
 4.0

Genauso gibt es dünne Vektoren:

In [29]: sparsevec(mein_vektor)

Out[29]: 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 [30]: struct MySparse
    n_rows::Int
    n_cols::Int
    values::Vector{Float64}
    row_idx::Vector{Int}
    col_ptr::Vector{Int}
end

In [31]: mat = MySparse(4, 4, float{[5, 8, 4, 7, 3, 1]}, [1, 2, 4, 3, 3, 4], [1, 2, 4, 5, 7])

Out[31]: 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 [32]: typeof(mat)

Out[32]: MySparse

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

In [33]: mat.n_rows

Out[33]: 4

In [34]: mat.values

Out[34]: 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 [39]: function sparse_mv(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 im 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_mv(mat, mein_vektor)

Out[39]: sparse_mv (generic function with 3 methods)
```

Hinweis: `struct s` sind *immutable*:

```
In [36]: mat.n_rows = 5

setfield!(immutable struct of type MySparse cannot be changed

Stacktrace:
 [1] setproperty!(x::MySparse, f::Symbol, v::Int64)
 @ Base ./base.jl:134
 [2] top-level scope
 @ In[36]:1
 [3] eval
 @ ./boot.jl:360 [inlined]
 [4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
 @ Base ./loading.jl:1094
```

Das kann mit `mutable struct s` umgangen werden:

```
In [37]: mutable struct MyMutableSparse
    n_rows::Int
    n_cols::Int
    values::Vector{Float64}
    row_idx::Vector{Int}
    col_ptr::Vector{Int}
end

In [38]: 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[38]: 5
```