Computerphysik Programmiertutorial 4a

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

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

Github: https://github.com/markusschmitt/compphys2021

Inhalt dieses Notebooks: Zeilennumerierung anzeigen, Array-Abstraktionen, Anonyme Funktionen, Rechnen auf dem Rechner: Maschinengenauigkeit, [E] Multiple Dispatch

Zeilennumerierung anzeigen

Insbesondere beim Lesen von Fehlermeldungen ist es hilfreich sich die Zeilennumerierung innerhalb der Jupyter Notebook Zellen anzeigen zu lassen. Das geht über das Menü `View -> Toggle Line Numbers` oder den Shortcut `L`.

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 1:11]
```

Out[1]:

```
11-element Vector{Int64}:
  2
  4
  6
  8
 10
 12
 14
 16
 18
 20
 22
```

In [2]:

```
arr = [2n for n in 1:11 if n%2==0]
```

Out[2]:

```
5-element Vector{Int64}:
  4
  8
 12
 16
 20
```

Anonyme Funktionen

Bisher haben wir die Funktionendefinition der Form

```
function <Funktionenname>(<Argumente>)
    <Anweisungsblock>
    return <Rückgabewert>
end
```

kennengelernt. So wird jeder Funktion insbesondere ein *Name* zugewiesen. Alternativ kann man aber auch **anonyme Funktionen** definieren.

In [3]:

```
x->x^2
```

Out[3]:

```
#7 (generic function with 1 method)
```

Anonyme Funktionen können einer Variablen zugewiesen werden:

In [4]:

```
myfunction = x->x^2
```

Out[4]:

```
#9 (generic function with 1 method)
```

In [5]:

```
myfunction(3)
```

Out[5]:

```
9
```

Anonyme Funktionen können als Argument an andere Funktionen übergeben werden:

In [6]:

```
map(x->x^2,[1,2,3])
```

Out[6]:

```
3-element Vector{Int64}:
 1
 4
 9
```

Genauso können anonyme Funktionen mit mehreren Argumenten definiert werden

In [8]:

```
neue_funktion = (x,y)->x+y

neue_funktion(3,4)
```

Out[8]:

```
7
```

Rechnen auf dem Rechner: Maschinengenauigkeit

Zahlen werden im Computer in einem Binärcode dargestellt und für jede Zahl steht nur eine begrenzte Anzahl von Bits zur Verfügung. Es können daher weder alle ganzen Zahlen $\mathbb{Z}$ noch alle reellen Zahlen $\mathbb{R}$ dargestellt werden.

Ganze Zahlen - `Int`

Wir haben schon in einem früheren Tutorial gesehen, dass Ganzzahlen in 64 bits als Binärzahlen dargestellt werden. Das ergibt automatisch eine Grenze für die größte darstellbare Zahl. Schauen wir uns diese Grenze an:

In [9]:

```
zahl1=2^63
bitstring(zahl1)
```

Out[9]:

```
"1000000000000000000000000000000000000000000000000000000000000000"
```

In [10]:

```
zahl2=2^63-1
bitstring(zahl2)
```

Out[10]:

```
"0111111111111111111111111111111111111111111111111111111111111111"
```

In [11]:

```
zahl1
```

Out[11]:

```
-9223372036854775808
```

In [12]:

```
zahl2
```

Out[12]:

```
9223372036854775807
```

Die ganzen Zahlen auf dem Computer sind ein "Kreis":

In [13]:

```
start=2^63-4
for i in 1:6
    println("$start + $i = $(start+i)")
end
```

```
9223372036854775804 + 1 = 9223372036854775805
9223372036854775804 + 2 = 9223372036854775806
9223372036854775804 + 3 = 9223372036854775807
9223372036854775804 + 4 = -9223372036854775808
9223372036854775804 + 5 = -9223372036854775807
9223372036854775804 + 6 = -9223372036854775806
```

Fließkommazahlen - `Float`

Auf dem Computer können wir sehr leicht große Summen ausrechnen. Ein Beispiel ist die **Harmonische Reihe**

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

Schreiben wir eine Funktion, die die n-te Harmonische Zahl H_n berechnet:

In [14]:

```
function H_forward(n, mytype=Float32)
    S = mytype(0.0)
    for k in 1:n
        S += mytype(1.0)/k
    end

    return S
end

H_forward(1000)
```

Out[14]:

```
7.4854784f0
```

Da Addition kommutativ ist, können wir die Summe in beliebiger Reihenfolge ausrechnen, z.B. auch in umgekehrter Reihenfolge:

In [15]:

```
function H_backward(n, mytype=Float32)
    S = mytype(0.0)
    for k in n:-1:1
        S += mytype(1.0)/k
    end

    return S
end

H_backward(1000)
```

Out[15]:

```
7.4854717f0
```

Mit unterschiedlicher Reihenfolge der Summation erhalten wir unterschiedliche Ergebnisse!

In [16]:

```
H_forward(100000)-H_backward(100000)
```

Out[16]:

```
0.0006980896f0
```

Reelle Zahlen werden im Computer als **Fließkommazahlen** behandelt. Das bedeutet, dass sie bezüglich einer festen **Basis** b in **Vorzeichen** $\pm$, **Mantisse** m und **Exponent** e zerlegt werden. Eine reelle Zahl $r \in \mathbb{R}$ wird also geschrieben als

$$r = \pm m \times b^e$$

Das Vorzeichen wird in einem Bit kodiert, für Mantisse und Exponent steht jeweils eine feste Zahl weiterer Bits zur Verfügung. Das Kodieren der Mantisse in einer begrenzten Anzahl von Bits bedeutet, dass wir bei jeder Zahl nur eine feste Anzahl von **signifikanten Ziffern** kennen. Die begrenzte Anzahl von Bits für den Exponenten bedeutet, dass es wie bei Ganzzahlen auch eine größte und kleinste darstellbare Fließkommazahl gibt.

Da der Computer nur mit einer bestimmten Zahl von signifikanten Ziffern rechnet, ist der Unterschied zwischen Zahlen nur begrenzt auflösbar. Diese "Auflösung" können wir experimentell bestimmen, indem wir fragen was die kleinste Zahl ϵ ist, so dass auf dem Computer $1.0 + \epsilon > 1.0$:

In [17]:

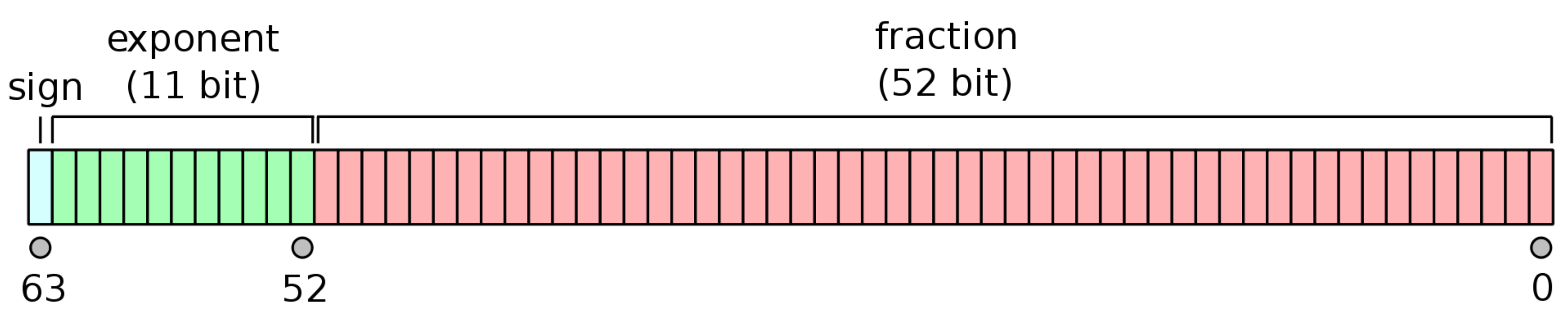
```
eps=1.0
n=0

while 1.0 + eps > 1.0
    eps /= 2.0
    n += 1
end

println("n = $n")
println("eps = 1/2^$n = $eps")
println("1 + eps = $(1+eps)")
println("1 + 2eps = $(1+2eps)")
```

```
n = 53
eps = 1/2^53 = 1.1102230246251565e-16
1 + eps = 1.0
1 + 2eps = 1.0000000000000002
```

Die 64 bits des Datentyps `Float64` sind wie folgt aufgeteilt (Bild gestohlen von benjaminjurke.com):



Wir haben also 1 Bit für das Vorzeichen, 11 Bits kodieren den Exponenten als ganze Zahl zwischen -1022 und 1023. Als Basis wird $b = 2$ verwendet. Die darstellbaren Zahlen bewegen sich also (in etwa) zwischen $2^{-1022} \approx 10^{-308}$ und $2^{1023} \approx 10^{308}$. Die übrigen 52 Bits werden verwendet um die Mantisse als

$$m = 1 + \sum_{n=1}^{52} \mathrm{bit}_n \frac{1}{2^n}$$

zu kodieren.

Die folgende Funktion stellt eine gegebnene Fließkommazahl entsprechend dar.

In [19]:

```
using Printf

function maschinendarstellung(x::Float64)

    bits = bitstring(x)
    sgn = bits[1]
    exponent = bits[2:12]
    mantissa = bits[13:64]

    println("Dezimal               | Vorz.  Exponent     Mantisse")
    println(@sprintf("%.15e |   %s    %s  %s", x, sgn, exponent, mantissa))
    return nothing
end
```

Out[19]:

```
maschinendarstellung (generic function with 1 method)
```

Schauen wir uns also an wie $1 + \epsilon = 1$ zustande kommt:

In [20]:

```
maschinendarstellung(eps)
```

```
Dezimal               | Vorz.  Exponent     Mantisse
1.110223024625157e-16 |   0    01111001010  0000000000000000000000000000000000000000000000000000
```

In [21]:

```
maschinendarstellung(1.0)
```

```
Dezimal               | Vorz.  Exponent     Mantisse
1.000000000000000e+00 |   0    01111111111  0000000000000000000000000000000000000000000000000000
```

In [23]:

```
maschinendarstellung(1.0+2eps)
```

```
Dezimal               | Vorz.  Exponent     Mantisse
1.000000000000000e+00 |   0    01111111111  0000000000000000000000000000000000000000000000000001
```

Beim Addieren zweier Zahlen unterschiedlicher Größenordnung geht Information über die kleinere Zahl verloren. Summationen sollten also immer so durchgeführt werden, dass nur ähnlich große Zahlen miteinander addiert werden.

Zurück zur Harmonischen Reihe. Welcher Summationsreihenfolge können wir also trauen? Für großes n gilt

$$H_n \approx \log(n) + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4}$$

mit der Euler-Gamma Konstante γ. Wir können also unsere beiden Ergebnisse damit vergleichen:

In [24]:

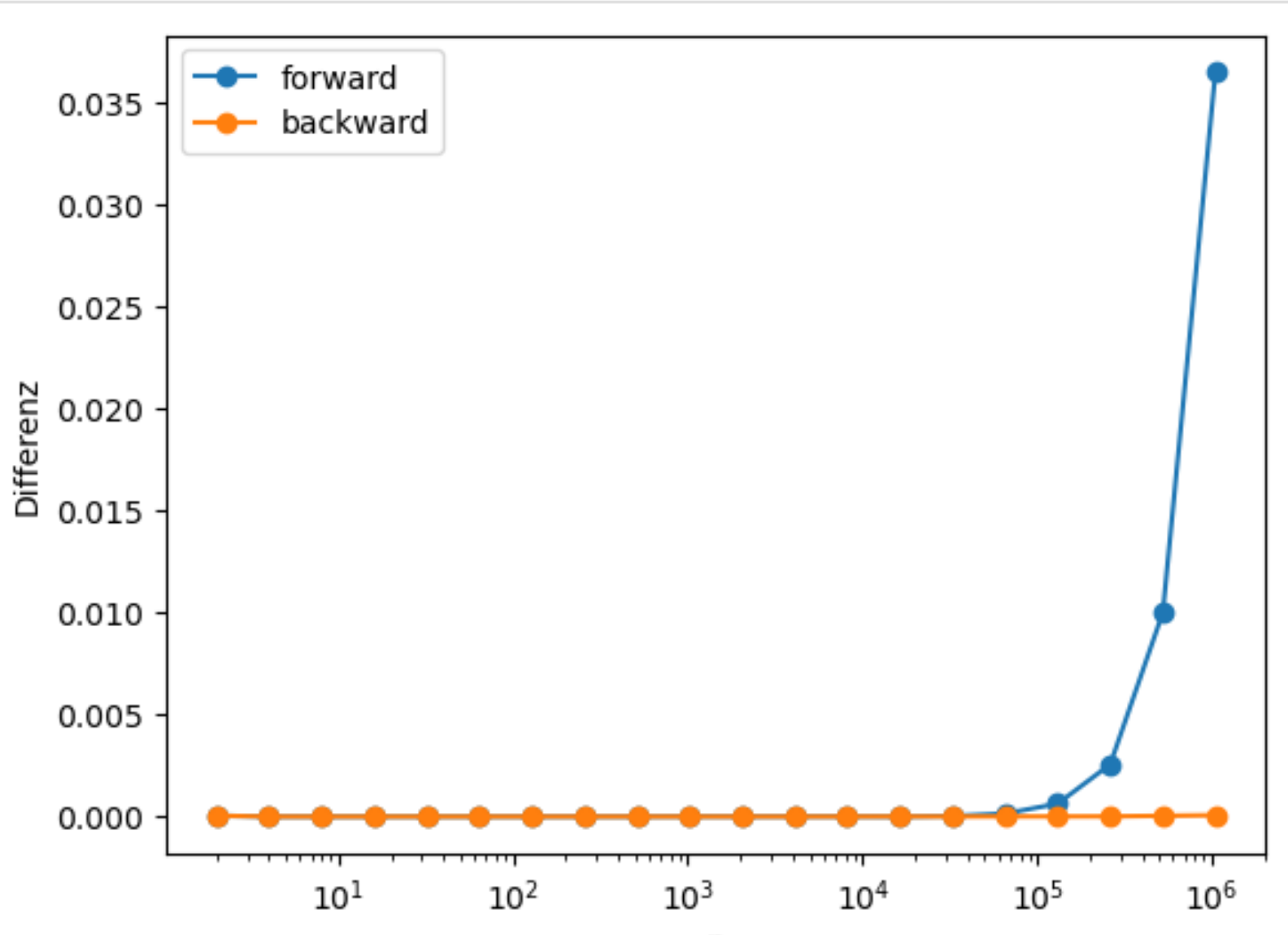
```
using PyPlot

function H_approx(n)
    n = Float64(n)
    return log(n)+Base.MathConstants.eulergamma+1.0/(2n)-1.0/(12n^2)+1.0/(120n^4)
end

n_werte = [2^n for n in 1:20]

Hn_fwd = [H_forward(2^n) for n in 1:20]
Hn_bwd = [H_backward(2^n) for n in 1:20]
Hn_approx = [H_approx(2^n) for n in 1:20]

semilogx(n_werte, abs.(Hn_fwd.-Hn_approx), "-o", label="forward")
semilogx(n_werte, abs.(Hn_bwd.-Hn_approx), "-o", label="backward")
xlabel("n")
ylabel("Differenz")
legend()
```



Out[24]:

```
PyObject <matplotlib.legend.Legend object at 0x7fcb38e75ee0>
```

[E] Multiple dispatch

Durch **multiple dispatch** können wir Funktionen definieren, deren Verhalten vom *Typ* der Argumente abhängt. Dazu werden Funktionen mit identischem Namen definiert, bei denen der Typ der Argumente durch Anhängen von `::<Datentyp>` spezifiziert ist.

Beispiel:

In [25]:

```
function fun(x::Int64)
    println("Mein Argument ist vom Typ Int64.")
    println("     check: ", typeof(x))
end

function fun(x::Float64)
    println("Mein Argument ist vom Typ Float64.")
    println("     check: ", typeof(x))
end
```

Out[25]:

```
fun (generic function with 2 methods)
```

In [26]:

```
fun(2)
```

```
Mein Argument ist vom Typ Int64.
     check: Int64
```

In [27]:

```
fun(2.0)
```

```
Mein Argument ist vom Typ Float64.
     check: Float64
```

In [28]:

```
fun(true)
```

```
MethodError: no method matching fun(::Bool)
Closest candidates are:
  fun(::Int64) at In[25]:1
  fun(::Float64) at In[25]:6

Stacktrace:
 [1] top-level scope
   @ In[28]:1
 [2] eval
   @ ./boot.jl:360 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
   @ Base ./loading.jl:1094
```

Neben den konkreten Datentypen wie `Int64` oder `Float64` gibt es in Julia auch **abstrakte Datentypen**, durch die alle Datentypen hierarchisch strukturiert werden. Abstrakte Datentypen fassen die konkreten Datentypen in Gruppen zusammen. Beispiele sind `Integer` für ganze Zahlen oder `Number` für alle Zahlen. Ob ein konkreter Datentyp einem abstrakten Datentyp zugeordnet ist, kann man mit dem Operator `<:` überprüfen:

In [29]:

```
Int64 <: Integer
```

Out[29]:

```
true
```

In [30]:

```
Float64 <: Integer
```

Out[30]:

```
false
```

In [31]:

```
Float64 <: Number
```

Out[31]:

```
true
```

In [32]:

```
Integer <: Number
```

Out[32]:

```
true
```

Funktionen können auch für abstrakte Datentypen *spezialisiert* werden:

In [33]:

```
function more_fun(x::Integer)
    println("$x ist eine ganze Zahl vom Typ ", typeof(x))
end

more_fun(3)
more_fun(Int32(3))
```

```
3 ist eine ganze Zahl vom Typ Int64
3 ist eine ganze Zahl vom Typ Int32
```

So können wir z.B. unsere Funktion `maschinendarstellung()` von oben auch für den Datentyp `Float32` *spezialisieren*:

In [36]:

```
using Printf

function maschinendarstellung(x::Float32)

    bits = bitstring(x)
    sgn = bits[1]
    exponent = bits[2:9]    # 8 bits für Exponent
    mantissa = bits[10:32]  # 23 bits für Mantisse

    println("Dezimal       | Vorz.  Exponent  Mantisse")
    println(@sprintf("%.7e |   %s    %s  %s", x, sgn, exponent, mantissa))
    return nothing
end
```

Out[36]:

```
maschinendarstellung (generic function with 2 methods)
```

In [34]:

```
maschinendarstellung(123.4)
```

```
Dezimal               | Vorz.  Exponent     Mantisse
1.234000000000000e+02 |   0    10000000101  1110110110011001100110011001100110011001100110011010
```

In [37]:

```
maschinendarstellung(Float32(123.4))
```

```
Dezimal       | Vorz.  Exponent  Mantisse
1.2340000e+02 |   0    10000101  11101101100110011001101
```