Formelzettel Julia (Vorderseite)

Stand: January 9, 2026

REPL / Pkg

- `?name` Hilfe/Docs, `]` Paketmodus, `;` Shell
- Umgebung: `] activate .` (lokales Project), `] instantiate`
- Pakete: `] add Plots`, `] status`

Typen / Konvert / Grenzen

```
typeof(x) # Typ von x (z.B. Int64, Float64, Vector{...})

Int(x) # Konvert: nach Int (Achtung: schneidet bei Float ab)
Float64(x) # Konvert: nach Float64
parse(Int, s) # String s -> Int (z.B. s="123" -> 123)

typemin(Int8) # kleinster Int8 Wert (-128)
typemax(Int8) # groesster Int8 Wert (127)
```

- `parse` fuer Strings; `typemin/max` fuer Overflow-Aufgaben.

Division / Modulo / Rational

```
a / b # "echte" Division -> oft Float (z.B. 1/2 = 0.5)
div(a,b) # ganzzahlige Division (Quotient ohne Rest)
a % b # Rest (modulo)

1//4 # Rational: exakt 1/4 (kein Rundungsfehler)
```

- `/` kann Float liefern; `//` ist exakt (wichtig bei Kettenbruechen).

Broadcasting (Punkt!) + Praezedenz

```
y = f.(x) # wendet f elementweise auf x an
y .= f.(x) # schreibt elementweise in y (in-place, keine neue
     Allocation)

A .+ 1 # addiert 1 zu JEDEM Eintrag von A (elementweise)
2 .^ (1:4) # elementweise Potenzen fuer Range 1:4 -> [2,4,8,16]
                    # Klammern bei Ranges vermeiden Parser-Fallen
```

- Punkt = elementweise (und oft schneller, weil “fused”).
- Bei Ranges fast immer Klammern setzen: `2 .^ (1:4)`.

Control Flow

```
if cond # falls cond == true
    ...
elseif cond2 # sonst falls cond2 == true
    ...
else # sonst
    ...
end

for k in 1:n # k laeuft ueber 1,2,...,n
    ...
end

while cond # wiederhole solange cond true ist
    ...
end

break # Schleife sofort verlassen
continue # naechster Schleifendurchlauf
```

Arrays / Indexing (1-basiert)

```
v = [1,2,3] # Vector{Int}
A = [1 2; 3 4] # Matrix: ; startet neue Zeile

v[1] # erstes Element (Julia ist 1-basiert!)
v[end] # letztes Element

A[i,j] # Element (i,j)
A[:,j] # ganze Spalte j
A[i,:] # ganze Zeile i

zeros(n) # Vector mit n Nullen (Float64)
ones(m,n) # m x n Matrix aus Einsen

xs = range(a, b; length=N) # N Stuetzstellen von a bis b (inkl.)
y = [f(x) for x in xs] # Comprehension: wende f auf jedes x an
```

- 1-basiert; `end` = letzter Index.
- `A[:,j]` ganze Spalte, `A[i,:]` ganze Zeile.

Iteratoren / Ranges (haeufig in Aufgaben)

```
r = 1:5 # Range (lazy), noch kein Array
v = collect(r) # macht daraus Vector [1,2,3,4,5]

for (i,val) in enumerate(v) # i=Index, val=Wert
    ...
end

for (a,b) in zip(1:3, 10:12) # Paare (1,10), (2,11), (3,12)
    ...
end

for i in eachindex(v) # sicherer Index-Iterator fuer Arrays
    ...
end
```

Funktionen (default + keyword) + Tuples

```
f(x) = x^2 + 1 # Kurzform: eine Zeile

function g(x, a=2; mu=0.0) # a: Default-Arg, mu: Keyword-Arg (nach
     ;)
    return a*x + mu # Rueckgabewert (return optional, aber klar)
end

mn, mx = minimum(v), maximum(v) # Mehrfachzuweisung (Tuple)
```

- Keywords nach `;` (z.B. `mu=`).
- Mehrere Rueckgaben via Tuple/Mehrfachzuweisung.

I/O (Konsole + Datei)

```
println("x=", x) # Ausgabe auf Konsole mit Zeilenumbruch
s = readline() # liest eine Zeile (String) von stdin
n = parse(Int, s) # String -> Int

open("out.txt","w") do io # Datei oeffnen (write), io ist Handle
    println(io, "hi") # schreibt in Datei (nicht Konsole)
end # Datei wird automatisch geschlossen

txt = read("out.txt", String) # liest ganze Datei als String
```

- `open(... )do io ... end` schliesst automatisch.

Debug / Inspection

```
@show x # druckt "x = <wert>" und gibt x zurueck
methods(f) # listet alle Methoden (Overloads) von f
@which f(args...) # zeigt, welche Methode fuer args gewaehlt wird
```

Mini-Fallen (merken!)

- Tuple immutable: `t[2]=...` geht nicht.
- Aliasing: `b=a` teilt Referenz; Mutation wirkt auf beide.
- Kopie: `copy(a)` (flach), `deepcopy(a)` (tief) falls verschachtelt.
- `for i in [1:4]` iteriert ueber *ein* Range-Objekt; richtig: `for i in 1:4`.

A) Baum-Rekursion: Knotenanzahl + Tiefe

```
function rec(B)
    N, tmax = 1, 0 # N: Knotenanzahl, tmax: max. Kind-Tiefe

    for kind in B # iteriere ueber alle Kinder-Teilbaeume
        Nk, tk = rec(kind) # rekursiv: (Knoten, Tiefe) des Kindes
        N += Nk # addiere Knoten des Kindes
        tmax = max(tmax, tk) # merke groesste Tiefe unter den
             Kindern
    end

    return N, 1 + tmax # Tiefe = 1 (aktueller) + max Kind-Tiefe
end
```

- Muster: "aktueller Knoten" + rekursiv ueber Kinder.
- Tiefe = `1 + max(Kinder-Tiefen)`.

B) Periode/Zyklus finden (Dict "seen")

```
function finde_periode(f, w)
    seen = Dict{Int,Int}() # map: Zustand w -> erster Index i
    i = 0 # Schrittzaehler

    while !haskey(seen, w) # solange Zustand w noch nicht gesehen
         wurde
        seen[w] = i # merke den ersten Zeitpunkt i fuer diesen
             Zustand
        w = f(w) # gehe zum naechsten Zustand
        i += 1 # ein Schritt weiter
    end

    N = seen[w] # Startindex des Zyklus (erstes Auftreten von w)
    d = i - N # Periodenlaenge (Abstand bis Wiederholung)
    return d, N # (Periode, Start)
end
```

- Speichere Zustand → Index; bei Wiederholung: Zyklusstart `N`, Periode `d`.

C) Polynom aus Koeff.-Vektor + Ableitung

```
werte(c, z) = sum([c[i]*z^(i-1) for i in 1:length(c)])
# c[i] ist Koeffizient von z^(i-1): i=1 -> z^0 (konstant), i=2 -> z
     ^1, ...

ableit(c) = [k*c[k+1] for k in 1:length(c)-1]
# Ableitung: (c[k+1]*z^k)' = k*c[k+1]*z^(k-1)
```

- Interpretation: `c[i]` ist Koeff. von z^{i-1}.
- Ableitung: aus $c_{k+1}z^k$ wird $kc_{k+1}z^{k-1}$.

D) Plotting (Plots.jl): Line + Scatter + Save

```
using Plots

x = range(0, 2pi; length=400) # viele Punkte im Intervall
y = sin.(x) # elementweise sin

plot(x, y; label="sin", xlabel="x", ylabel="y",
     title="Line plot") # Linienplot
plot!(x, cos.(x); label="cos") # zweite Kurve im selben Plot

scatter!(x[1:20:end], y[1:20:end]; label="samples") # Punkte oben
     drauf (optional)

savefig("fig.pdf") # Plot speichern (pdf/png)
```

- `plot!`/`scatter!` fuegt zum aktuellen Plot hinzu; ohne `!` neuer Plot.
- Labels/Achsen: `xlabel`, `ylabel`, `title`, `label`.

E) Kettenbruch: auswerten / erzeugen (exakt)

```
function berechne_kettenbruch(a)::Rational
    v = a[end] # starte hinten: v = a_n
    for alpha in a[end-1:-1:1] # gehe rueckwaerts: a_{n-1},...,a_1
        v = alpha + 1//v # v = alpha + 1/v (exakt rational!)
    end
    return v
end

function erzeuge_kettenbruch(w::Rational)
    a = Int[] # sammelt die alpha-Werte (Ganzteile)
    while true
        alpha = floor(Int, w) # ganzzahliger Anteil von w
        push!(a, alpha) # alpha ans Ende der Liste
        d = w - alpha # Restteil
        d == 0 && break # wenn Rest 0: fertig
        w = 1//d # sonst invertieren (exakt) und weiter
    end
    return a
end
```

- `1//v` haelt alles rational (kein Float-Drift).
- Rueckwaerts auswerten: $v = a_n$, dann $v = a_k + 1/v$.

F) Bits / Interpretation

```
bitstring(x) # gibt Bitdarstellung als String zurueck
reinterpret(Int8, 0b11111100)# interpretiert dieselben Bits als
     Int8
```

- `bitstring`: Binaerdarstellung; `reinterpret`: gleiche Bits, anderer Typ.

G) Numerik/Stats Mini-Toolbox

```
using LinearAlgebra
x = A \ b # loest lineares Gleichungssystem A*x = b
dot(v,w) # Skalarprodukt
norm(v) # euklidische Norm

using Random, Statistics
Random.seed!(1) # macht Zufall reproduzierbar
x = randn(N) # N normalverteilte Zufallszahlen
mean(x); std(x) # Mittelwert und Standardabweichung
```