Institut für Theoretische Physik
Universität zu Köln

Prof. Dr. Simon Trebst

M. Sc. Carsten Bauer

Computerphysik
Vorlesung — Programmiertechniken 5

Sommersemester 2019

Website: [http://www.thp.uni-koeln.de/trebst/Lectures/2019-CompPhys.shtml (http://www.thp.uni-koeln.de/trebst/Lectures/2019-CompPhys.shtml)](http://www.thp.uni-koeln.de/trebst/Lectures/2019-CompPhys.shtml)

Wir werden heute ein größeres Paket benötigen, dessen Installation einige Minuten in Anspruch nimmt. Es ist daher empfehlenswert, die Installation jetzt gleich zu starten.

In [ ]:
```
] add DifferentialEquations
```

In [ ]:
```
using DifferentialEquations
```

0. Erinnerung letzte Vorlesung

Komplexität und O-Notation

In [3]:
```
@time 4+4
```

```
  0.000009 seconds (5 allocations: 208 bytes)
```

Out[3]: 8

In [4]:
```
@elapsed 4+4
```

Out[4]: 4.499e-6

Bubblesort ist ein $O(N^2)$ Sortierverfahren, d.h. die Laufzeit skaliert (asymptotisch) wie N^2.

Julias internes `sort!` ist fast (aber nicht ganz) $O(N)$. Welches [Sortierverfahren (https://de.wikipedia.org/wiki/Sortierverfahren)](https://de.wikipedia.org/wiki/Sortierverfahren) verwendet Julia intern?

In [5]:
```julia
@which sort!(rand(100))
```

Out[5]: sort!(v::**AbstractArray{T,1} where T**) in Base.Sort at sort.jl:682 (https://github.com/JuliaLang/julia/tree/80516ca20297a67b996caa08c38786332379b6a5/base/sort.jl#L682)

Externe Pakete

Beispiel: Measurement.jl (https://juliaphysics.github.io/Measurements.jl/stable/)

In [6]:
```julia
using Measurements
```

In [8]:
```julia
x = measurement(3.2, 0.1)
```

Out[8]: 3.2 ± 0.1

In [9]:
```julia
y = 1.0 ± 0.05
```

Out[9]: 1.0 ± 0.05

In [10]:
```julia
x * y
```

Out[10]: 3.2 ± 0.19

Standard Libraries (Pakete, die mit Julia ausgeliefert werden)

In [11]:
```julia
using Statistics
```

Links: Statistics (https://docs.julialang.org/en/latest/stdlib/Statistics/)

In [12]:
```julia
x = rand(200)
```

Out[12]:
```
200-element Array{Float64,1}:
 0.994393662094202
 0.08779193604037738
 0.4184891525018517
 0.20333703984953422
 0.22415779968999727
 0.6415330519581091
 0.753688389237934
 0.5510187022206741
 0.8592424169890989
 0.9209370314077547
 0.33554665430324637
 0.053051085859611824
 0.23007672383810607
 ⋮
 0.03987067588488258
 0.5518873407005032
 0.3266460992053608
 0.419723851287102
 0.8402890086451413
 0.8536712543934948
 0.70572241852739
 0.3040947312861695
 0.5486130835841527
 0.428641856217469
 0.047712351203718306
 0.30439792792016984
```

In [13]:
```julia
mean(x)
```

Out[13]:
```
0.4693033828560531
```

In [14]:
```julia
var(x)
```

Out[14]:
```
0.0787363873080993
```

1. Differentialgleichung lösen

In [15]:
```julia
using DifferentialEquations
```

Links: [DifferentialEquations.jl (https://github.com/JuliaDiffEq/DifferentialEquations.jl)](https://github.com/JuliaDiffEq/DifferentialEquations.jl) ([Dokumentation (https://docs.juliadiffeq.org/latest/index.html)](https://docs.juliadiffeq.org/latest/index.html))

Beispiel Problem: Kohlenstoffzerfall

Der Zerfall von Kohlenstoff-14 folgt der folgenden gewöhnlichen Differentialgleichung erster Ordnung

$$\frac{du(t)}{dt} = -cu(t) = f(u, p, t)$$

wobei $u(t)$ die Kohlenstoffkonzentration darstellt und $c = 5730$ Jahre die Halbwertszeit von C14 ist.

Allgemeine Schritte um das Problem in Julia zu lösen

1. Problem definieren
2. Problem lösen
3. Lösung analysieren/visualisieren

Schritt 1: Problem definieren

Das **Anfangswertproblem** ist charakteresiert durch

- einen Anfangswert u_0,
- eine Zeitspanne t_{span} und
- die Differentialgleichung $\frac{du}{dt} = f(u, p, t)$.

In [16]:
```
# Anfangswert & Zeitspanne
u0 = 1.0
tspan = (0.0, 1.0)

# Differentialgleichung
f(u, p, t) = - 5.730 * u

# ODE Problem
problem = ODEProblem(f, u0, tspan)
```

Out[16]:
```
ODEProblem with uType Float64 and tType Float64. In-place: false
timespan: (0.0, 1.0)
u0: 1.0
```

Schritt 2: Problem lösen

In [17]:
```
sol = solve(problem)
```

Out[17]:
```
retcode: Success
Interpolation: Automatic order switching interpolation
t: 12-element Array{Float64,1}:
 0.0
 0.0497536522894885
 0.10705970310084381
 0.17811921361859132
 0.25811384814284366
 0.34843185700982904
 0.4465160102060756
 0.552035751198497
 0.6634798881930261
 0.7801928441938321
 0.9013306881395161
 1.0
u: 12-element Array{Float64,1}:
 1.0
 0.7519478110084717
 0.5414785397148167
 0.36037071375687324
 0.2278678607690813
 0.13580882743232403
 0.07741915037207427
 0.04229292859083828
 0.022333004681040325
 0.011442440679109465
 0.005715875714947843
 0.0032474850624780484
```

Schritt 3: Problem analysieren/visualisieren

In [18]:
```
sol.t
```

Out[18]:
```
12-element Array{Float64,1}:
 0.0
 0.0497536522894885
 0.10705970310084381
 0.17811921361859132
 0.25811384814284366
 0.34843185700982904
 0.4465160102060756
 0.552035751198497
 0.6634798881930261
 0.7801928441938321
 0.9013306881395161
 1.0
```

In [19]:
```
sol.u
```

Out[19]:
```
12-element Array{Float64,1}:
 1.0
 0.7519478110084717
 0.5414785397148167
 0.36037071375687324
 0.2278678607690813
 0.13580882743232403
 0.07741915037207427
 0.04229292859083828
 0.022333004681040325
 0.011442440679109465
 0.005715875714947843
 0.0032474850624780484
```

Das Lösungsobjekt `sol` ist eine Funktion, die automatisch zwischen den berechneten Funktionswerten interpoliert:

In [20]:
```
sol(0.5432)
```

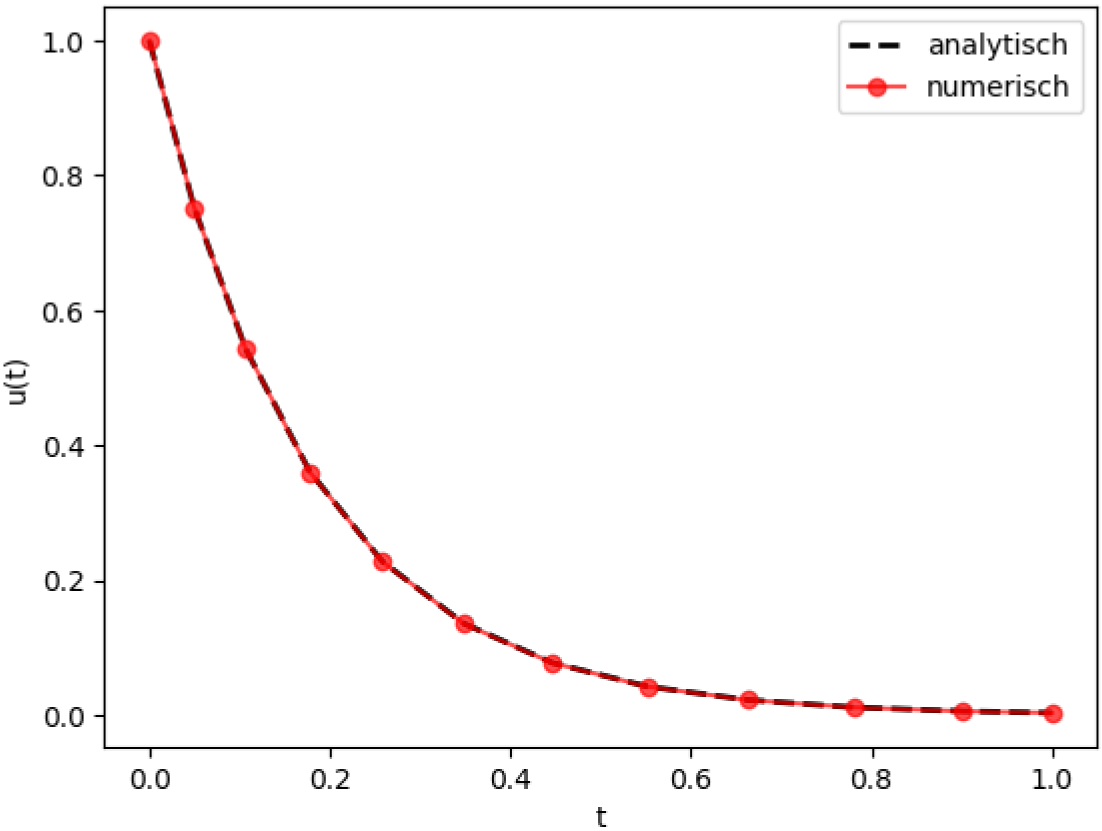
Out[20]:
```
0.0444891486453217
```

In [23]:
```
u = @. u0 * exp(-5.730 * sol.t)
```

Out[23]:
```
12-element Array{Float64,1}:
 1.0
 0.7519477584296727
 0.5414784020389519
 0.36037032560224214
 0.22786718313851745
 0.13580781187404758
 0.077417940295537
 0.04229168203913233
 0.02233188593263336
 0.01144153644711691
 0.005715206536379227
 0.0032470772336105863
```

In [25]:
```
using PyPlot
```

In [24]:
```
# Visualisierung
plot(sol.t, u, "--", color="black", linewidth=2, label="analytisch")
plot(sol.t, sol.u, "o-", color="red", alpha=.7, label="numerisch")
legend()
ylabel("u(t)")
xlabel("t");
```



Einschub: DifferentialEquations.jl + Measurements.jl

In [26]:
```
# Anfangswert & Zeitspanne
u0 = 1.0 ± 0.1
tspan = (0.0 ± 0.0, 1.0 ± 0.0)

# Differentialgleichung
f(u,p,t) = - (5.730 ± 2) * u

# ODE Problem definieren
problem = ODEProblem(f, u0, tspan)
```

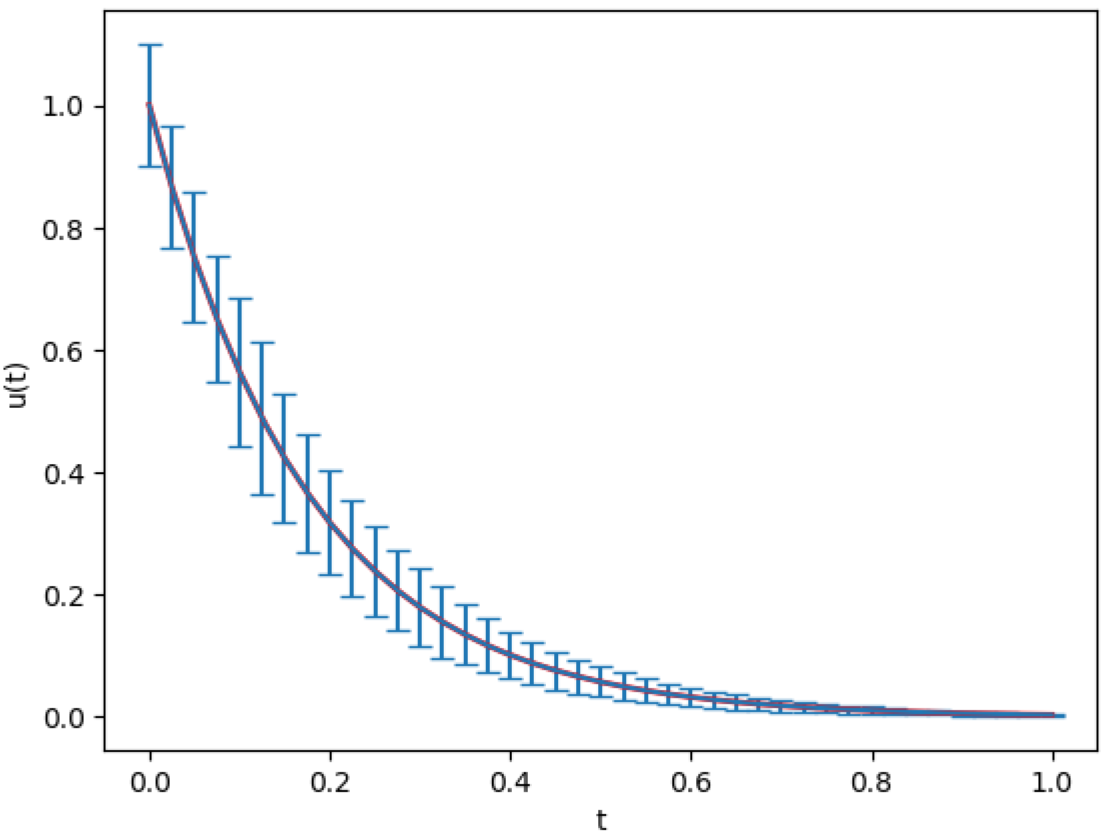
Out[26]:
```
ODEProblem with uType Measurement{Float64} and tType Measurement{Float64}. In-p
lace: false
timespan: (0.0 ± 0.0, 1.0 ± 0.0)
u0: 1.0 ± 0.1
```

In [27]:
```
sol = solve(problem, saveat=0.025)

# analytic solution
u = u0 .* exp.(-(5.730 ± 2) .* sol.t);

# plot solution
ts = getfield.(sol.t, :val)
solvals = getfield.(sol, :val)
solerrs = getfield.(sol, :err);

errorbar(ts, solvals, yerr=solerrs)
plot(ts, getfield.(u, :val), color="red", lw=2)
ylabel("u(t)")
xlabel("t");
```



Das Beeindruckende ist hier, dass die Autoren von DifferentialEquations.jl und Measurements.jl [nie zusammengearbeitet haben (https://discourse.julialang.org/t/differentialequations-jl-and-measurements-jl/6350)](https://discourse.julialang.org/t/differentialequations-jl-and-measurements-jl/6350). Das Feature wurde durch Julias Konstruktionsprinzip "automatisch" erzeugt. Mehr dazu im Workhop (siehe unten).

2. Differentialgleichungen höherer Ordnungen

Beispiel Problem: Klassisches Pendel

$$\ddot{\theta} + \frac{g}{L}\sin(\theta) = 0$$

Analytisch lösen wir diese Differentialgleichung typischerweise in der "Kleine Winkel Näherung", d.h. $sin(\theta) \approx \theta$, da wir sonst Elliptische Integrale erhalten die keine geschlossene analytische Lösung haben.

In Julia haben wir aber numerische Integratoren zur Hand! Wieso dann nicht die volle Differentialgleichung lösen?

In [28]:
```
# Konstanten
g = 9.81
L = 1.0

# Anfangsbedingungen & Zeitspanne
u0 = [pi/2, 0]
tspan = (0.0, 6.3)
params = [g, L]
```

Out[28]:
```
2-element Array{Float64,1}:
 9.81
 1.0
```

In [29]:
```
function pendulum!(du, u, p, t)
    g, L = p # Parameter "auspacken"
    θ = u[1]
    dθ = u[2]

    du[1] = dθ
    du[2] = - (g/L) * sin(θ)
end
```

Out[29]:
```
pendulum! (generic function with 1 method)
```

In [ ]:
```
# function pendulum(u, p, t)
#     g, L = p # Parameter "auspacken"
#     θ = u[1]
#     dθ = u[2]

#     return [dθ, - (g/L) * sin(θ)]
# end
```

In [30]:
```
prob = ODEProblem(pendulum!, u0, tspan, params)
```

Out[30]:
```
ODEProblem with uType Array{Float64,1} and tType Float64. In-place: true
timespan: (0.0, 6.3)
u0: [1.5708, 0.0]
```

In [33]:
```
sol = solve(prob, saveat=0.05);
```

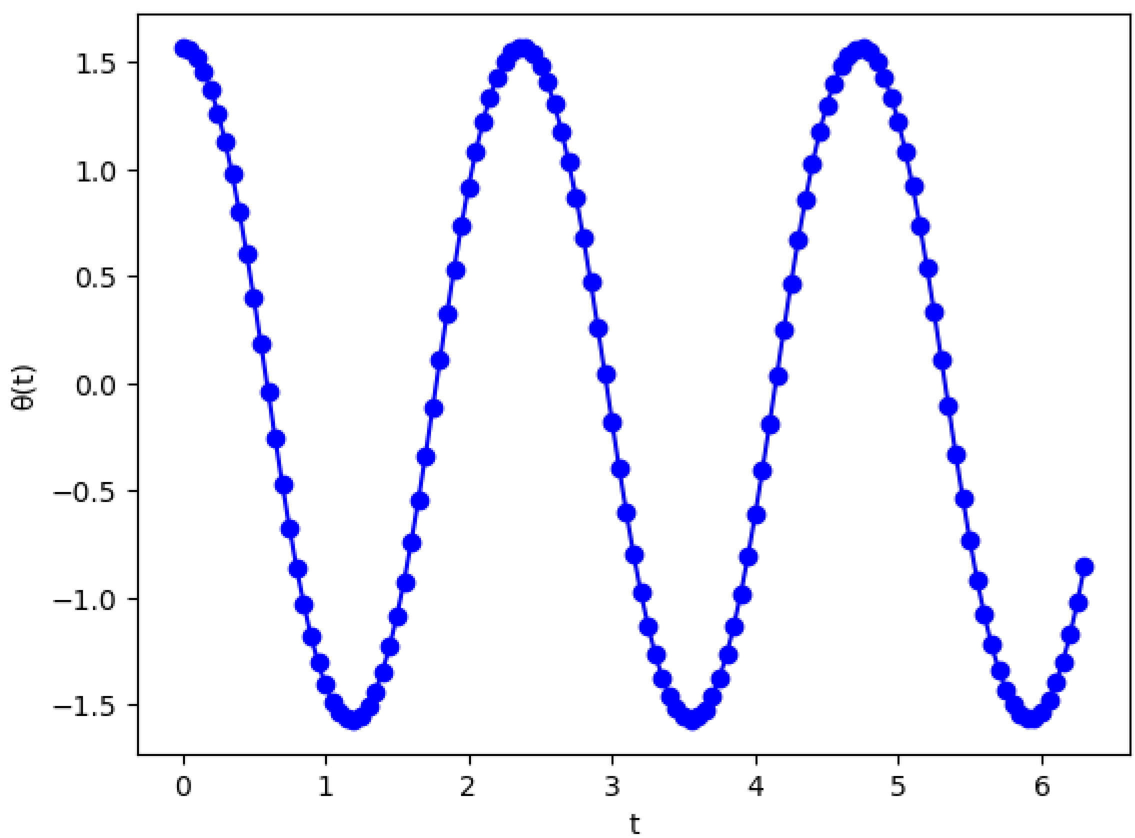
In [34]:
```
x = rand(10)
```

Out[34]:
```
10-element Array{Float64,1}:
 0.6443312186181427
 0.4192211355675344
 0.6118341117527704
 0.8334953761670723
 0.12330572423639397
 0.6411173842081075
 0.6371458802647416
 0.8410357474180892
 0.5326663036068471
 0.6992082405470461
```

In [35]:
```
# x[3] = getindex(x, 3)
getindex.(x,3)
```

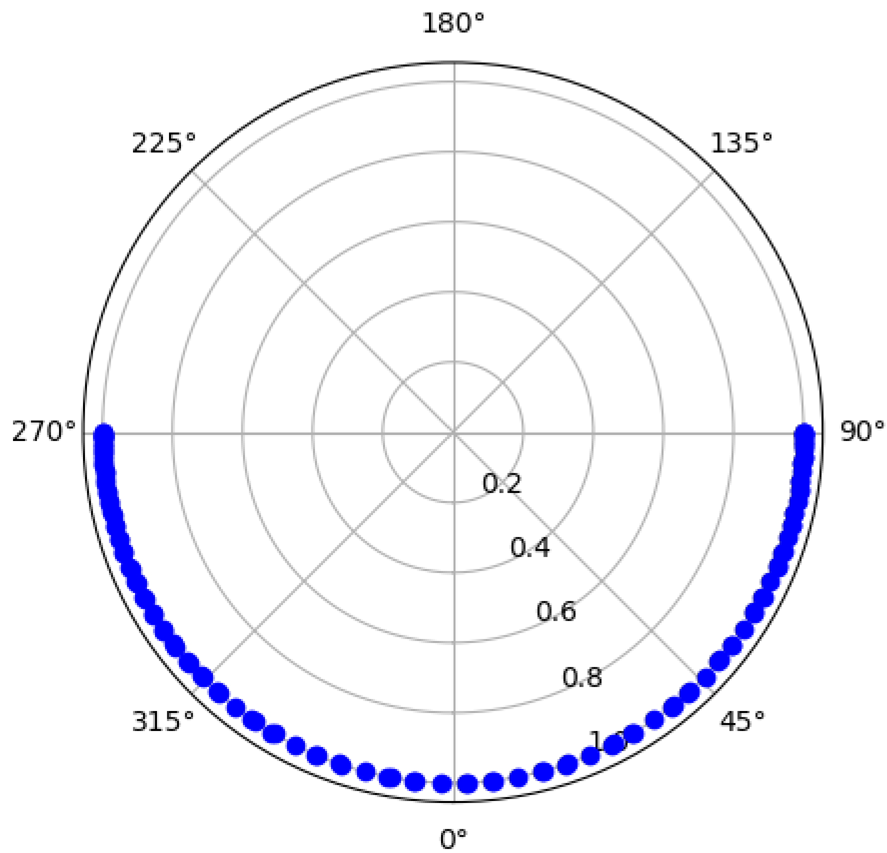
Out[35]:
```
0.6118341117527704
```

In [32]:
```
# Visualisierung
plot(sol.t, getindex.(sol.u, 1), "bo-",);
xlabel("t")
ylabel("θ(t)");
```



In [37]:
```
figure()
for u in sol.u
    polar(u[1], L, "bo")
end

# Rotate polar plot such that zero is "in the south"
gca().set_theta_zero_location("S")
```



3. Differentialgleichungssysteme: Der Lorenz-Attraktor

Für die Simulation eines einfachen **Wetter-Modells** hat der Meteorologe Edward Lorenz 1963 eine Beschreibung von Luftströmungen entworfen. Dazu hat er ein Gleichungssystem von drei gekoppelten Differentialgleichungen betrachtet:

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

Im folgenden wollen wir dieses Differentialgleichungssytem lösen.

In [40]:
```julia
σ = 10.0
ρ = 28.0
β = 8/3

params = [σ, ρ, β]

function lorenz!(du,u,p,t)
    σ, ρ, β = p
    du[1] = σ * (u[2] - u[1])
    du[2] = u[1] * (ρ - u[3]) - u[2]
    du[3] = u[1] * u[2] - β * u[3]
end
```

Out[40]:
```
lorenz! (generic function with 1 method)
```

In [43]:
```julia
u0 = [1.0; 0.0; 0.0]

tspan = (0.0, 100.0)

prob = ODEProblem(lorenz!, u0, tspan, params)

sol = solve(prob, saveat=0.01)
```

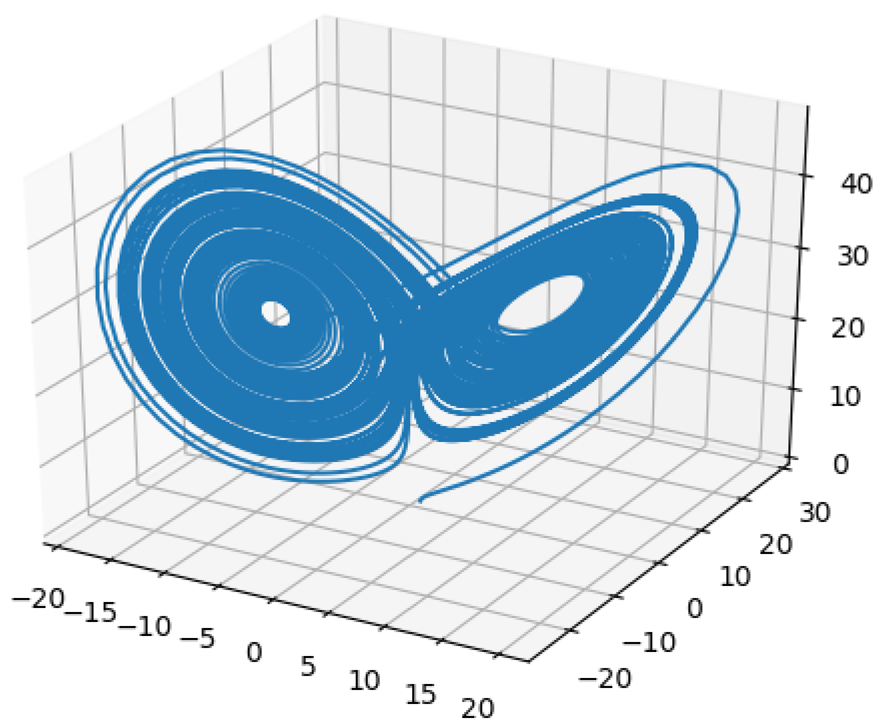
Out[43]:
```
retcode: Success
Interpolation: 1st order linear
t: 10001-element Array{Float64,1}:
   0.0
   0.01
   0.02
   0.03
   0.04
   0.05
   0.06
   0.07
   0.08
   0.09
   0.1
   0.11
   0.12
   ⋮
  99.89
  99.9
  99.91
  99.92
  99.93
  99.94
  99.95
  99.96
  99.97
  99.98
  99.99
 100.0
u: 10001-element Array{Array{Float64,1},1}:
 [1.0, 0.0, 0.0]
 [0.917924, 0.26634, 0.0012639]
 [0.867919, 0.511741, 0.00465545]
 [0.84536, 0.744654, 0.00983587]
 [0.846806, 0.972332, 0.0167345]
 [0.869786, 1.20113, 0.0254862]
 [0.912641, 1.43677, 0.0364012]
 [0.974385, 1.68452, 0.0499584]
 [1.05462, 1.9494, 0.0668157]
 [1.15346, 2.23634, 0.08784]
 [1.27146, 2.55025, 0.114138]
 [1.40961, 2.89616, 0.14713]
 [1.56929, 3.27935, 0.188616]
 ⋮
 [14.5376, 9.61993, 39.5324]
 [13.9841, 7.87141, 39.7244]
 [13.3213, 6.20466, 39.6301]
 [12.5693, 4.66392, 39.2822]
 [11.7492, 3.28145, 38.7203]
```

```
[10.8831, 2.07757, 37.9914]
[9.994, 1.06053, 37.1485]
[9.10211, 0.22393, 36.2307]
[8.22359, -0.446963, 35.2669]
[7.37208, -0.968807, 34.2835]
[6.55837, -1.36029, 33.3008]
[5.79036, -1.64216, 32.3329]
```

Da haben wir wohl einen Fehler gemacht. Der **Stacktrace** hilft uns ihn effizient zu finden.

In [46]:
```
ux = getindex.(sol.u, 1) # getindex(x, i) == x[i]
uy = getindex.(sol.u, 2)
uz = getindex.(sol.u, 3)

pygui(true)
plot3D(ux, uy, uz);
```



In [45]:
```
pygui(false);
```

4. Endliche Machinenpräzision

Wie Objekte im Computer repräsentiert werden, d.h. wie sie in Nullen und Einsen kodiert werden, ist durch ihren Typ bestimmt. Wir können den Typ einer Variable mit Hilfer der Funktion `typeof` herausfinden.

In [47]:
```
a = 5
```

Out[47]: 5

In [48]:
```julia
typeof(a)
```

Out[48]:
```
Int64
```

In [49]:
```julia
c = "5"
```

Out[49]:
```
"5"
```

In [50]:
```julia
typeof(c)
```

Out[50]:
```
String
```

In [51]:
```julia
a = 5.0
```

Out[51]:
```
5.0
```

In [52]:
```julia
typeof(a)
```

Out[52]:
```
Float64
```

In [54]:
```julia
5 == 5.0 # Werte-äquivalent
```

Out[54]:
```
true
```

In [56]:
```julia
5 === 5.0 # aber nicht identisch
```

Out[56]:
```
false
```

Julia ist eine sogenannte **dynamisch typisierte Sprache**, was bedeutet, dass Objekte ihren Typ dynamisch je nach Kontext verändern können. Zu jedem Zeitpunkt hat jedes Objekt jedoch einen bestimmten Typ.

In [57]:
```julia
a = 3
println(typeof(a))

a = "test"
println(typeof(a))
```

```
Int64
String
```

Die herkömmlichen Datentypen wie `Int64` und `Float64` stellen Zahlen durch eine endliche Anzahl von 64 Bits (Einsen und Nullen) dar. Daraus folgt direkt, dass z.B. nicht jede Fließkommazahl exakt repräsentiert werden kann.

In [58]:
```julia
1.2 + 0.12
```

Out[58]:
```
1.3199999999999998
```

In [ ]:
```julia
x - 0.0 < 1e-12
```

Ein weiterer Effekt der endlichen Repräsentation von Zahlen ist der sogenannt ***Integer-Overflow***.

In [59]:
```julia
x = typemax(Int64)
```

Out[59]:
```
9223372036854775807
```

In [60]:
```julia
x + 1
```

Out[60]:
```
-9223372036854775808
```

In den allermeisten Fällen ist die endliche Machinenpräzision kein Problem. Falls doch kann man, auf Kosten der Laufzeit, Datentypen mit beliebiger Präzision verwenden:

In [61]:
```julia
x = big"1.2"
```

Out[61]:
```
1.200000000000000000000000000000000000000000000000000000000000000000000000000000
7
```

In [62]:
```julia
typeof(x)
```

Out[62]:
```
BigFloat
```

In [63]:
```julia
big"1.2" + big"0.12"
```

Out[63]:
```
1.320000000000000000000000000000000000000000000000000000000000000000000000000000
4
```

5. Offene Fragerunde

Beispiele:

- Was sind Methoden (im Unterschied zu Funktionen)?
- Wo bekomme ich Hilfe, wenn ich Julia-Fragen habe?
- Warum ist der erste Funktionsaufrauf besonders (langsam)?
- Wie update/deinstalliere ich Pakete?
- ....

6. Ankündigung: Fortgeschrittener Julia Workshop

In den Semesterferien wird ein auf der Vorlesung aufbauender **fortgeschrittener Julia Workshop** stattfinden. Weitere Informationen und eine Umfrage bzgl. Teilnahme und Themen werden gegen Ende des Semesters folgen.

the julia programming language