practicum - Julia

November 25, 2018

1 Practicum differentiaalvergelijkingen: De Lorenz Attractor

Link naar notebookviewer: https://nbviewer.jupyter.org/github/Cubedsheep/practicum-diff-Julia/blob/master/practicum%20-%20Julia.ipynb?fbclid=IwAR0w5uD6396VRiiF5jM1AxzXG7EB4HG5WAsDR

Link naar Github: https://github.com/Cubedsheep/practicum-diff-Julia

1.1 Importeren packages

```
In [1]: using Plots
        using Distributed
        using PyCall
        using PyPlot
        using DelimitedFiles
        const plt = PyPlot
        ioff()
        using3D()
        #pygui(true)
```

Voeg extra 'workers' toe om berekeningen parallel te kunnen runnen

```
In [2]: addprocs(9);
        @everywhere begin
            using BenchmarkTools
            using LinearAlgebra
            using SharedArrays
            using Printf
        end
```

1.1.1 Importeren python packages

```
In [3]: @pyimport sympy as sp;
        @pyimport scipy.linalg as lin;
        @pyimport numpy as np;
        @pyimport matplotlib as mpl;
        const col = mpl.colors;
        #plt.xkcd()
```

1.2 definieren functies en variabelen

Gradiënt implementeert de differentiaalvergelijking. Merk op dat alles binnen een @everywhere enviroment gewraped zit zodat de functies beschikbaar zijn voor alle workers

1.2.1 dynamische functies (args meegeven aan gradient)

```
In [7]: # definieer functies en variabelen voor alle workers
        @everywhere begin
            function gradient(t0::Float64, X::Array{Float64, 1}, arg::Array{Float64})
                # de functie die de afeleide van de te zoeken functie geeft
                # in het punt (x, y, z)
                (x, y, z) = X
                (sigma, r, b) = arg

                dx = -sigma*x+sigma*y
                dy = r*x-y-x*z
                dz = -b*z+x*y

                return [dx, dy, dz]
            end

            function euler(f, x0, h, n, arg)
                # implementatie van de methode van Euler om numeriek de oplossing van
                # een differentiaalvergelijking te benaderen.
                # f is het rechterlid van de ODE, x0 de beginwaarde, h de stapgrootte, n
                # het aantal stappen en arg de extra argumenten voor f
                x = zeros(Float64, 3, n)
                x[:, 1] = x0

                for i in 2:n
                    x[:, i] = x[:, i-1] + h .* f(0., x[:, i-1], arg)
                end
                return x
            end


            function RK4(f, x0, h, n, arg)
                # gebruikt de methode van Runge-Kata van orde 4 om een benaderende
                # oplossing te geven van het autonoom stelsel X'=f(X), met beginvoorwaarde x0
                # stapgrootte h en n stappen
                x = zeros(Float64, 3, n)
                x[:, 1] = x0

                for i in 2:n
                    k1 = f(0., x[:, i-1], arg)
                    k2 = f(0., x[:, i-1] + h * k1/2, arg)
                    k3 = f(0., x[:, i-1] + h * k2/2, arg)
```

```
            k4 = f(0., x[:, i-1] + h * k3, arg)

            x[:, i] = x[:, i-1] + h/6 * (k1+2*k2+2*k3+k4)
        end
        return x
    end
end
```

1.2.2 statische varianten (enkel de waarden uit de opgave in gradient)

In [8]:
```
# definieer functies en variabelen voor alle workers
# sigma = 10, r = 28, b = 8/3
@everywhere begin
    function gradient_s(t0::Float64, X::Array{Float64, 1})
        # de functie die de afeleide van de te zoeken functie geeft
        # in het punt (x, y, z)
        (x, y, z) = X

        dx = -10.0*x+10.0*y
        dy = 28.0*x-y-x*z
        dz = -8/3*z+x*y

        return [dx, dy, dz]
    end

    function euler_s(f, X0, h, n)
        # implementatie van de methode van Euler om numeriek de oplossing van
        # een differentiaalvergelijking te benaderen.
        # f is het rechterlid van de ODE, x0 de beginwaarde, h de stapgrootte, n
        # het aantal stappen en arg de extra argumenten voor f
        x = zeros(Float64, 3, n)
        x[:, 1] = X0

        for i in 2:n
            x[:, i] = x[:, i-1] + h .* f(0., x[:, i-1])
        end
        return x
    end

    function euler2_s(f, X0, h, n)
        # implementatie van de verbeterde methode van Euler om numeriek de oplossing v
        # een differentiaalvergelijking te benaderen.
        # f is het rechterlid van de ODE, x0 de beginwaarde, h de stapgrootte, n
        # het aantal stappen en arg de extra argumenten voor f
        x = zeros(Float64, 3, n)
        x[:, 1] = X0

        for i in 2:n
```

```
            k1 = f(0., x[:, i-1])
            k2 = f(0., x[:, i-1] + h*k1)
            x[:, i] = x[:, i-1] + h*(k1+k2)/2
        end
        return x
    end

    # gebruikt de methode van Runge-Kata van orde 4 om een benaderende oplossing te ge
    # stapgrootte h en n stappen
    function RK4_s(f, X0, h, n)
        x = zeros(Float64, 3, n)
        x[:, 1] = X0

        for i in 2:n
            k1 = f(0., x[:, i-1])
            k2 = f(0., x[:, i-1] + h*k1/2)
            k3 = f(0., x[:, i-1] + h*k2/2)
            k4 = f(0., x[:, i-1] + h*k3)

            x[:, i] = x[:, i-1] + h/6 .* (k1+2*k2+2*k3+k4)
        end
        return x
    end
end
```

1.2.3 Andere functies

Functie om afstand van lijn tot punt te berekenen

In [9]:
```
@everywhere begin
    #=
    berekent de afstand van de rechte opgespannen door de punten beg
    tot het punt point
    begin, end en point moeten 1d vectoren van dezelfde dimensie zij
    returns: Float64
    =#
    function dist_line_point(beginP::Array{Float64, 1}, endP::Array{Float64, 1}, point
        return norm(cross(endP-beginP, beginP-point))/ norm((endP-beginP))
    end

    #=
    berekent de orthogonale projectie van het punt point op de recht
    beginP en endP
    =#
    function orth_projection(beginP, endP, point)
        P = point
        P0= beginP
        v = P-P0
```

```
            s = endP - beginP
            I = fill(1., 3, 3)
            projectie = (dot(v, s) / dot(s,s))*s
            return projectie + P0
        end
    end
```

1.2.4 initializeren variabelen

```
In [10]: @everywhere begin
             # initiële condities en parameters voor elke methode
             x0 = [-13.763610682134, -19.578751942452, 27]
             he = 10^-5
             he2 = 5*10^-4
             hr = 10^-4
             # het tijdsinterval om over te integreren
             d=2
             # het aantal stappen om te zetten
             ne = ceil(Int64, d/he)
             ne2 = ceil(Int64, d/he2)
             nr = ceil(Int64, d/hr)
             args = [10., 28., 8/3]
         end
```

1.3 Oefening 2

```
In [9]: # initializeer de symbolen om symbolisch mee te rekenen.
        t, x, y, z = sp.symbols("t, x, y, z")
        #sp.init_printing()

        # definieer de componenten van de ODE
        sigma=10
        r = 28
        b = sp.S("8/3")
        dx = -sigma*x+sigma*y
        dy = r*x-y-x*z
        dz = -b*z+x*y
```

```
Out[9]: PyObject x*y - 8*z/3
```

Bepaal de kritieke punten

```
In [10]: kritieke_punten = sp.solve([dx, dy, dz], (x, y, z))
         vars = [x, y, z]
         funcs = [dx, dy, dz]
         (O, B, A) = (kritieke_punten[1], kritieke_punten[3], kritieke_punten[2])
         # print de latex code van de kritieke punten voor in het verslag
         println(sp.latex(A))
```

```
        println(sp.latex(B))
        println(sp.latex(O))
```

```
\left ( - 6 \sqrt{2}, \quad - 6 \sqrt{2}, \quad 27\right )
\left ( 6 \sqrt{2}, \quad 6 \sqrt{2}, \quad 27\right )
\left ( 0, \quad 0, \quad 0\right )
```

Functie om de Jacobiaan in een punt sub uit te rekenen

```
In [11]: function num_jacobian(funcs, args, sub)
             return [[sp.diff(fun, arg)[:subs](sub) for arg in args] for fun in funcs]
         end;
```

Reken de Jacobiaan uit in de kritieke punten

```
In [12]: subA = [vars[i] => A[i] for i in 1:3];
         subB = [vars[i] => B[i] for i in 1:3];
         subO = [vars[i] => O[i] for i in 1:3];
         JA = num_jacobian(funcs, vars, subA);
         JB = num_jacobian(funcs, vars, subB);
         JO = num_jacobian(funcs, vars, subO);
```

Bereken de eigenvectoren en bijhorende eigenwaarden van de Jacobiaan in de kritieke punten

```
In [13]: Ja = sp.Matrix(JA, dtype=np.float64);
         eigA = Ja[:eigenvects]();
         write("eigvecsA.txt", sp.latex(eigA))
```

```
Out[13]: 7038
```

```
In [14]: Jb = sp.Matrix(JB, dtype=np.float64);
         eigB = Jb[:eigenvects]();
         write("eigvecsB.txt", sp.latex(eigB))
```

```
Out[14]: 7046
```

```
In [15]: Jo = sp.Matrix(JO, dtype=np.float64);
         eigO = Jo[:eigenvects]();
         write("eigvecsO.txt", sp.latex(eigO))
```

```
Out[15]: 504
```

Numerieke waarde van de eigenwaarden en vectoren in het kritieke punt A

```
In [13]: eigA = eigen(np.matrix(JA, dtype=np.float64))
         print(eigA)
```

```
Eigen{Complex{Float64},Complex{Float64},Array{Complex{Float64},2},Array{Complex{Float64},1}}(C
```

Symbolisch berekenen eigenwaarden en eigenvectoren

1.4 Oefening 3

1.4.1 functie voor bepalen kortste afstand en tijdstip

```
In [14]: @everywhere begin
             #=
             zoekt het punt op curve dat het dichtst bij x0 ligt
             x0: punt, 1d array van grootte d
             curve: 2d array met size dxn, met n het aantal punten op de curve
             guess: (i1, i2) tuple met een gok tussen welke indexen het dicht
             h: de stapgrootte, nodig voor het berekenen van de tijd waarop h
             returns: tuple met tijd, afstand en index (T, d, index)
             =#
             function find_closest(x0, curve, guess, h)
                 # definieer nieuwe arrays om alleen binnen het opgegeven bereik te zoeken
                 (t1, t2) = guess;
                 len = t2-t1+1;

                 # ---------------------
                 # vind het punt in curve dat het dichtste ligt
                 # definieer array om afstanden in op te slaan
                 dists = zeros(Float64, len);
                 # bereken de afstanden
                 for i in t1:t2
                     dists[i-t1+1] = norm(curve[:,i]-x0);
                 end
                 smallest_distance = minimum(dists);
                 # index van het punt dat het dichts bij de x0 ligt
                 index = findfirst(isequal(smallest_distance), dists) + t1 - 1;

                 # ---------------------
                 # bereken de afstanden tot de 2 lijnstukken die naast dit punt liggen
                 d_line1 = dist_line_point(curve[:,index-1], curve[:,index], x0);
                 d_line2 = dist_line_point(curve[:,index], curve[:,index+1], x0);

                 # ---------------------
                 # kijk na of de projecties op de lijnen wel degelijk op de lijnstukken liggen
                 P1 = orth_projection(curve[:,index-1], curve[:,index], x0);
                 P2 = orth_projection(curve[:,index], curve[:,index+1], x0);
                 # P = curve_punt + v*di, als curve_punt het eerste punt van het lijnstuk is
                 # en 0<= di <= 1, dan ligt de projectie op het lijnstuk
                 di1 = (x0[1]-curve[1,index-1])/(curve[1,index]-curve[1,index-1]);
                 di2 = (x0[1]-curve[1,index])/(curve[1,index+1]-curve[1,index]);
                 on_curve1 = (0 <= di1) && (di1 <= 1);
                 # -for debugging- println("lijn1: $(on_curve1), $(d_line1)")
                 on_curve2 = (0 <= di2) && (di2 <= 1);
                 # -for debugging- println("lijn1: $(on_curve2), $(d_line2)")
```

```
            # bereken de tijd
            if (!on_curve1) && (!on_curve2)
                T = (index-1)*h;
                return (T, smallest_distance, index)
            else
                # een van de 2 projecties ligt zeker op het lijnstuk
                # op het lijnstuk, dus afstand tot lijn kleiner dan of gelijk aan afstand
                if on_curve1 && (!on_curve2 || d_line1 < d_line2)
                    # projectie ligt op lijn 1 en afstand kleiner dan projectie2, of proj
                    # ligt niet op de lijn: return afstand tot lijn1
                    T = (index-2+di1) * h;
                    return (T, d_line1, index)
                else
                    # projectie ligt op lijn 2 en afstand is kleiner dan die tot lijn1
                    # of de projectie ligt niet op lijn 1
                    T = (index-1+di2) * h;
                    return (T, d_line2, index)
                end
            end
            return
        end


        #=
        berekent de afstand in de x-richting tussen het dichtste punt op
        gegeven punt x0.
        index is de index van het dichtste punt, T het de periode
        returns: index
        =#
        function x_dist(x0, curve, index, T, h)

            point = curve[:,index];

            if (T < h*(index-1))
                point = orth_projection(curve[:, index-1], curve[:, index], x0)
            elseif (T > h*(index-1))
                point = orth_projection(curve[:, index], curve[:, index+1], x0)
            end

            return abs(point[1] - x0[1])
        end
    end
```

1.4.2 bereken heel exact de periode met Runge-Kutta

Deze periode wordt gebruikt om een eerste gok te doen waar het punt het dichtst bij de oplossing zich bevindt

```
In [17]: # definieer bereik stapgroottes om te testen
         # (arrays compleet overbodig, gewoon copy-pasta van hieronder  )
         num = 1
         iter = 1:num
         h = iter .* 1e-7
         n = ceil.(Int64, 1.6 ./ h)
         # los de vergelijkingen op
         sol = pmap(RK4_s, fill(gradient_s,num), fill(x0,num), h, n);
         guess = [ceil.(Int64, (1.54, 1.57)./h[i]) for i in iter]
         # zoek de waarde voor t waar de oplossing het dichtst bij x0 komt
         closest = pmap(find_closest, fill(x0, num), sol, guess, h)
         T = closest[1][1]

Out[17]: 1.5586522107160146
```

1.4.3 Runge-Kutta

Bereken een aantal oplossingen met variërende stapgroottes

```
In [23]: # definieer bereik stapgroottes om te testen
         num = 9
         iter = 1:num
         h = iter .* 1e-4
         n = ceil.(Int64, 1.6 ./ h)
         # los de vergelijking op met de verschillende beginwaarden, gebruik pmap om deze verg
         # parallel op te lossen
         sol = pmap(RK4_s, fill(gradient_s,num), fill(x0,num), h, n);
```

Bereken voor elk van deze oplossingen de kleinste afstand tot de beginwaarde en het tijdstip waarop deze bereikt wordt

```
In [24]: guess = [(floor(Int64, T/h[i])-10, ceil(Int64, T/h[i])+10) for i in iter];
         closest = pmap(find_closest, fill(x0, num), sol, guess, h);
         index = [closest[i][3] for i in iter]; Ts = [closest[i][1] for i in iter];
         x_dists = pmap(x_dist, fill(x0, num), sol, index, Ts, h);

In [25]: for i in iter
             @printf("Runge-Kutta4 - periode: %.8f, x-afwijking: %.2e, stapgrootte: %.2e \n", 
         end

Runge-Kutta4 - periode: 1.55865222, x-afwijking: 9.40e-07, stapgrootte: 1.00e-04
Runge-Kutta4 - periode: 1.55865225, x-afwijking: 2.91e-06, stapgrootte: 2.00e-04
Runge-Kutta4 - periode: 1.55865233, x-afwijking: 8.48e-06, stapgrootte: 3.00e-04
Runge-Kutta4 - periode: 1.55865242, x-afwijking: 1.40e-05, stapgrootte: 4.00e-04
Runge-Kutta4 - periode: 1.55865250, x-afwijking: 2.00e-05, stapgrootte: 5.00e-04
Runge-Kutta4 - periode: 1.55865258, x-afwijking: 2.51e-05, stapgrootte: 6.00e-04
Runge-Kutta4 - periode: 1.55865283, x-afwijking: 4.22e-05, stapgrootte: 7.00e-04
Runge-Kutta4 - periode: 1.55865297, x-afwijking: 5.22e-05, stapgrootte: 8.00e-04
Runge-Kutta4 - periode: 1.55865282, x-afwijking: 4.17e-05, stapgrootte: 9.00e-04
```

1.4.4 Methode van Euler

Definieer functie die niet heel de oplossing onthoudt

```
In [21]: @everywhere begin
             function euler_l(f, x0, h, n)
                 x = x0
                 min_dist = 100
                 dist = 0
                 index = 0

                 for i in 2:n
                     x = x + h .* f(0., x)
                     dist = abs(x[1] - x0[1])
                     if (i > n/2) && (dist < min_dist)
                         min_dist = dist
                         index = i
                     end
                 end
                 return (min_dist, index)
             end
         end
```

Bereken een aantal oplossingen met variërende stapgroottes

```
In [22]: # definieer bereik stapgroottes om te testen
         num = 5
         iter = 1:num
         h = iter .* 1e-7
         n = ceil.(Int64, 1.6 ./ h)
         #sol = pmap(euler_s, fill(gradient_s,num), fill(x0,num), h, n);

In [23]: sols = pmap(euler_l, fill(gradient_s, num), fill(x0, num), h, n);
```

Bereken voor elk van deze oplossingen de kleinste afstand tot de beginwaarde en het tijdstip waarop deze bereikt wordt

```
In [24]: for i in iter
             @printf("euler - periode: %.4f, afwijking: %.2e, stapgrootte: %.2e \n", (sols[i][1
         end
```

```
euler - periode: 1.5587, afwijking: 1.87e-06, stapgrootte: 1.00e-07
euler - periode: 1.5587, afwijking: 3.12e-06, stapgrootte: 2.00e-07
euler - periode: 1.5586, afwijking: 7.26e-06, stapgrootte: 3.00e-07
euler - periode: 1.5586, afwijking: 6.01e-06, stapgrootte: 4.00e-07
euler - periode: 1.5586, afwijking: 1.06e-05, stapgrootte: 5.00e-07
```

1.4.5 verbeterde methode van Euler

Bereken een aantal oplossingen met variërende stapgroottes

```
In [25]: # definieer bereik stapgroottes om te testen
         num = 9
         iter = 1:num
         h = iter .* 2e-4
         n = ceil.(Int64, 1.6 ./ h)
         sol = pmap(euler2_s, fill(gradient_s,num), fill(x0,num), h, n);
```

Bereken voor elk van deze oplossingen de kleinste afstand tot de beginwaarde en het tijdstip waarop deze bereikt wordt

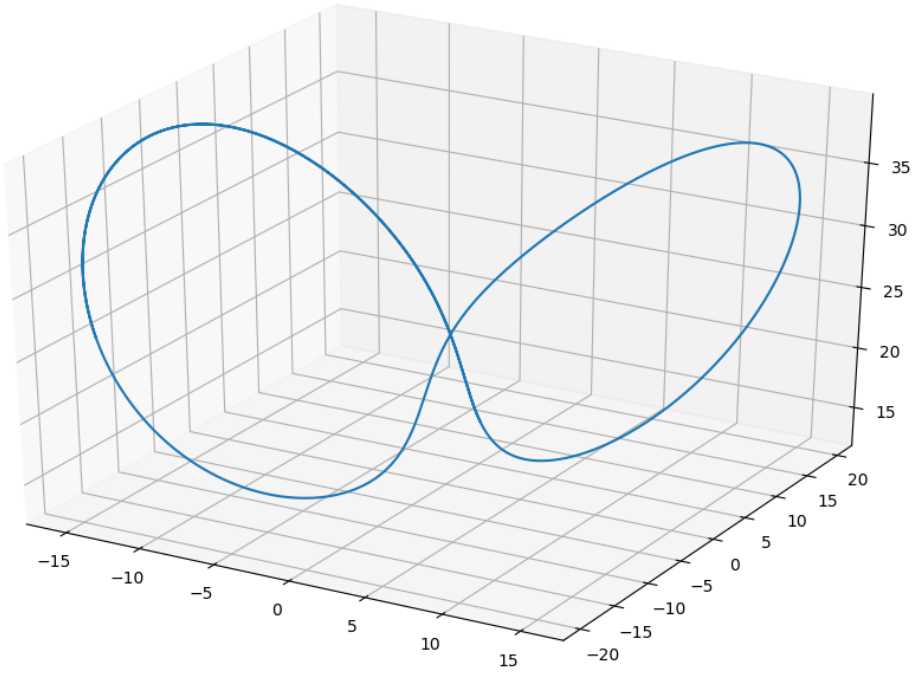
```
In [26]: guess = [(floor(Int64, T/h[i])-10, ceil(Int64, T/h[i])+10) for i in iter];
         closest = pmap(find_closest, fill(x0, num), sol, guess, h);
         index = [closest[i][3] for i in iter]; Ts = [closest[i][1] for i in iter];
         x_dists = pmap(x_dist, fill(x0, num), sol, index, Ts, h);
```

```
In [27]: for i in iter
             @printf("Verbeterde methode van Euler - periode: %.8f, x-afwijking: %.2e, stapgroo
         end
```

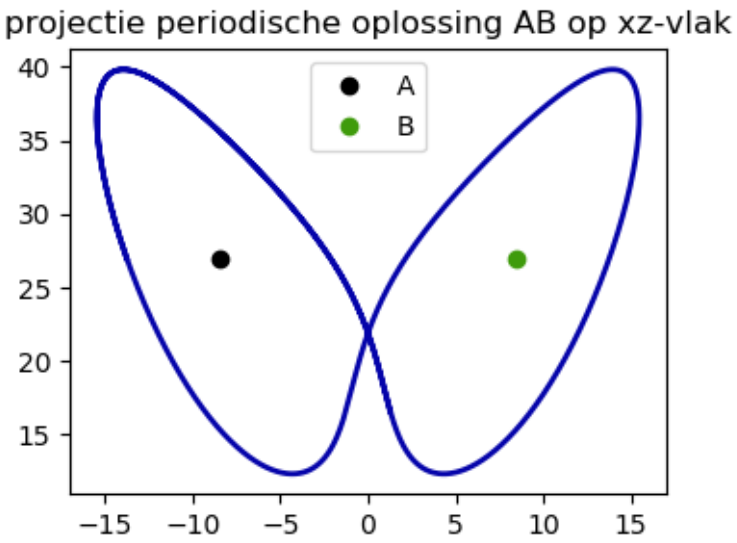
```
Verbeterde methode van Euler - periode: 1.55865157, x-afwijking: 5.60e-07, stapgrootte: 2.00e-0
Verbeterde methode van Euler - periode: 1.55864967, x-afwijking: 4.42e-06, stapgrootte: 4.00e-0
Verbeterde methode van Euler - periode: 1.55864638, x-afwijking: 3.00e-06, stapgrootte: 6.00e-0
Verbeterde methode van Euler - periode: 1.55864188, x-afwijking: 8.77e-06, stapgrootte: 8.00e-0
Verbeterde methode van Euler - periode: 1.55863611, x-afwijking: 1.86e-05, stapgrootte: 1.00e-0
Verbeterde methode van Euler - periode: 1.55862807, x-afwijking: 3.62e-05, stapgrootte: 1.20e-0
Verbeterde methode van Euler - periode: 1.55862020, x-afwijking: 1.12e-05, stapgrootte: 1.40e-0
Verbeterde methode van Euler - periode: 1.55860887, x-afwijking: 8.47e-05, stapgrootte: 1.60e-0
Verbeterde methode van Euler - periode: 1.55859686, x-afwijking: 1.35e-04, stapgrootte: 1.80e-0
```

1.4.6 plots periodische oplossing

```
In [17]: sol = RK4_s(gradient_s, x0, hr, nr)

         fig = figure("Lorenz attractor",figsize=(12,8))
         ax = fig[:add_subplot](1,1,1, projection = "3d")

         ax[:plot3D](sol[1,:], sol[2,:], sol[3,:])
         plt.show()
```



In [18]:
```
fig, ax = plt.subplots(1,1,figsize=(4, 3))

ax[:plot](sol[1,:], sol[3,:], "xkcd:royal blue", linewidth=2)
ax[:plot]([A[1]],[A[3]], "ok", label="A")
ax[:plot]([B[1]],[B[3]], "o", color="xkcd:grass green", label="B")
plt.title("projectie periodische oplossing AB op xz-vlak")
plt.legend()

plt.show()
fig[:savefig]("projectie_opdracht_3.pdf")
```



1.5 Oefening 4

1.5.1 functie om beginwaarde te benaderen

```
In [9]: # definieer functie voor alle workers
        @everywhere begin
            #=
            deze functie zoekt de beginwaarde (y0) voor een periodische oplo
            voor gegeven waarden voor x0 en z0. het doet dit door oplossingen
            en zoekt naar de dichtste waarde op het interval guess_I voor de
            @Args:
            x0: x-waarde van de te zoeken beginwaarde
            y0: y-waarde van de te zoeken beginwaarde
            h: stapgrootte die gebruikt wordt om oplossingen te berekenen
            n: aantal iteratiestappen voor berekenen oplossing
            guess_y0: tuple(Float64), zoekt in dit interval naar de y-waarde
            guess_I: tuple, zoekt in de iteratiestappen in dit interval naar
            rounds: het aantal rondes gebruikt om een betere benadering van
            @returns:
            lijst met steeds betere benaderingen van de beginwaarde + hoe di
            =#
            function find_y0(x0, z0, h, n, guess_y0, guess_I, rounds)
                # initializeren variabelen
                # vul arrays met de stapgrootte, aantal stappen en rechterlid om parallel de v
                # op te lossen
                H = fill(h, 5);
                N = fill(n, 5);
                GRAD = fill(gradient_s, 5);
```

```
X0 = (0:4)/4 .* fill([0, guess_y0[2]-guess_y0[1], 0], 5) + fill([x0, guess_y0[
sols = pmap(RK4_s, GRAD, X0, H, N);

# in de for-loop worden slechts 2 oplossingen opnieuw berekend, pas de arrays
N = fill(n, 2);
GRAD = fill(gradient_s, 2);

# -------------------------------------
# bereken de kleinste afstanden
# vul arrays om dit parallel te doen
GUESS = fill(guess_I, 5);
dists = pmap(find_closest, X0, sols, GUESS, H);
GUESS = fill(guess_I, 2);
H = fill(h, 2);

dists = [d[2] for d in dists];
best = minimum(dists);
index = findfirst(isequal(best), dists);

# sla de beginwaarde en bijhorende afstand op
results = [];
push!(results, [X0[index], best]);

# -------------------------------------
# refine the best guess 'rounds' times
for i in 1:rounds
    # ---------------------------------
    # herbereken de beginwaarden
    if index == 1
        index = index+1;
    elseif index == 5
        index = index-1;
    end
    # verander de plaats van de nieuwe eindpunten en het midden in de arrays
    # begin
    tempx1 = X0[index-1];
    temps1 = sols[index-1];
    tempd1 = dists[index-1];
    # midden
    tempx3 = X0[index];
    temps3 = sols[index];
    tempd3 = dists[index];

    # einde
    tempx5 = X0[index+1];
    temps5 = sols[index+1];
    tempd5 = dists[index+1];
```

```
                (X0[1], sols[1], dists[1]) = (tempx1, temps1, tempd1)
                (X0[3], sols[3], dists[3]) = (tempx3, temps3, tempd3)
                (X0[5], sols[5], dists[5]) = (tempx5, temps5, tempd5)

                # bereken de data voor de 2 nieuwe punten
                # nieuwe beginwaarde
                (X0[2], X0[4]) = ((X0[1] + X0[3])/2, (X0[5] + X0[3])/2);
                # bereken de oplossingen
                (sols[2], sols[4]) = pmap(RK4_s, GRAD, [X0[2], X0[4]], H, N);
                # vind de dichtste punten
                (temp2, temp4) = pmap(find_closest, [X0[2], X0[4]], [sols[2], sols[4]], GU
                (dists[2], dists[4]) = (temp2[2], temp4[2]);

                # sla de nieuwe waarden op
                best = minimum(dists);
                index = findfirst(isequal(best), dists);
                # voeg de resultaten toe aan de array met resultaten
                push!(results, [X0[index], best]);
            end
            return results
        end
    end
```

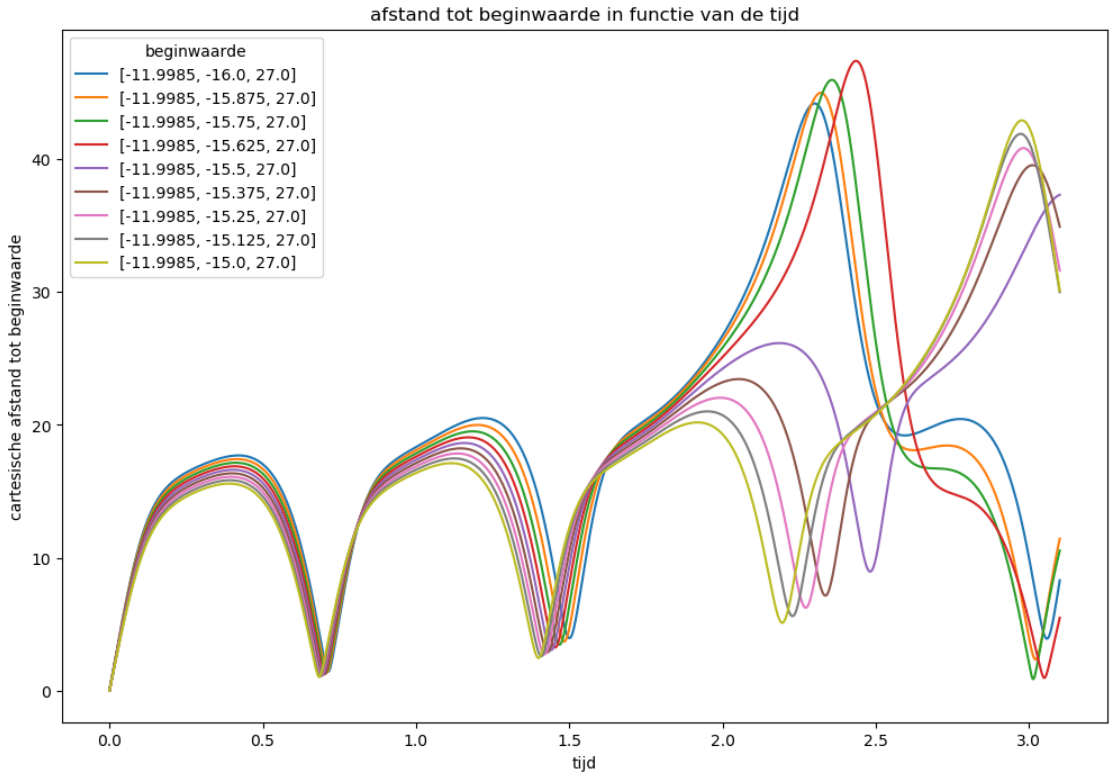
1.5.2 test

In [20]:
```
# definieer bereik stapgroottes om te testen
num = 9
iter = 1:num
h = fill(1e-4, num)
T = 4
n = ceil.(Int64, T ./ h)
# definieer de beginvoorwaarden
x1 = [-11.998523280062, -16, 27]
step = 1/8
X = fill([0, step, 0], num) .* (0:(num-1)) .+ fill(x1, num)
# los de differentiaalvergelijkingen op
sol = pmap(RK4_s, fill(gradient_s, num), X, h, n);

b0 = fill.(X, n)
sols = [[sol[i][:,j] for j in 1:n[i]] for i in iter]
dists = [pmap(norm, b0[i]-sols[i]) for i in iter];
```

In [21]:
```
dists_init = [dists[i][1:31000] for i in iter];
Y1 = X;
```

In [22]:
```
fig, ax = plt.subplots(1,1,figsize=(12,8))

plot_range = 1:31000
```

```
for i in iter
    ax[:plot](plot_range/1e4, dists_init[i][plot_range], label="$(Y1[i])")
end

plt.legend(title="beginwaarde")
plt.title("afstand tot beginwaarde in functie van de tijd")
ax[:set_xlabel]("tijd")
ax[:set_ylabel]("cartesische afstand tot beginwaarde")
fig[:savefig]("afstand_tot_beginwaarde_exploratie.pdf", dpi=400)
plt.show()
```



We zien dat de rode, groene en oranje curve het dichtst in de buurt komen van het gewenste gedrag. We zoeken dus verder in de buurt van deze beginwaarden

1.5.3 zoek de juiste beginwaarde

```
In [11]: # initialiseer de beginwaarden en parameters voor de functie
         x0 = -11.998523280062;
         z0 = 27;
         h = 1e-4;
         n = 32001;
         guess_y0 = [-15.7, -15.6];
         guess_I = [29000, 32000];
         rounds = 15;
```

```
In [24]: # zoek een zo goed mogelijke benadering, de fout zal met een stapgrootte van 1e-4
         # niet veel kleiner dan 1e-5 kunnen worden
         results = find_y0(x0, z0, h, n, guess_y0, guess_I, rounds);
         results[rounds][2]
```

```
Out[24]: 4.146571768813626e-6
```

Print de periode en beginwaarde met hoge precisie

```
In [25]: @printf("Beginwaarde: y(0) = %.8f, periode: T = %.8f", results[rounds][1][2], 5)
```

```
Beginwaarde: y(0) = -15.68425446, periode: T = 5.00000000
```

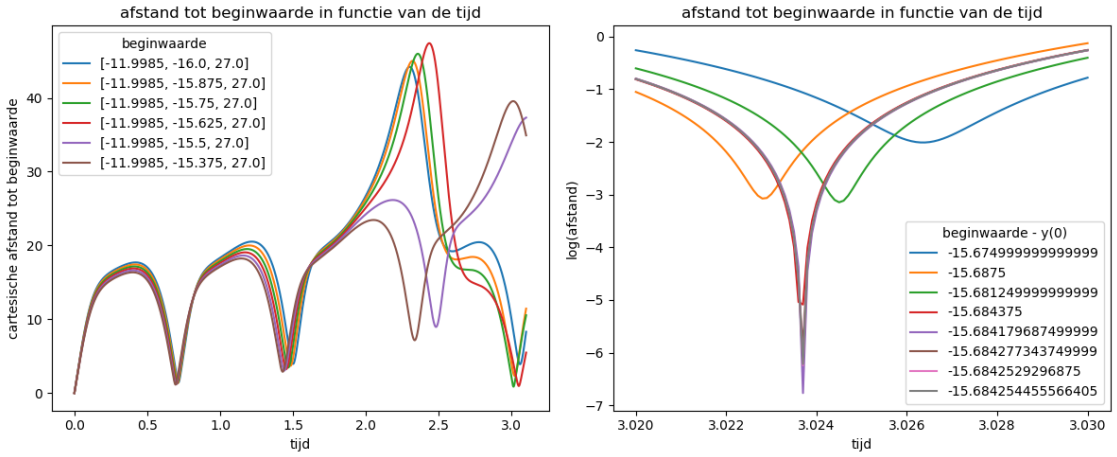
1.5.4 Plots voor verslag

```
In [26]: # verwijder de dubbels
         results = unique(results);
```

```
In [27]: # vind de opeenvolgende oplossingen
         num = 8
         iter = 1:num

         X = [result[1] for result in results]
         h = fill(1e-4, num)
         n = fill(32001, num)
         GRAD = fill(gradient_s, num)
         sol = pmap(RK4_s, GRAD, X, h, n);
         # vind de afstanden tot de beginwaarde
         b0 = fill.(X, n)
         sols = [[sol[i][:,j] for j in 1:n[i]] for i in iter]
         dists = [pmap(norm, b0[i]-sols[i]) for i in iter];
```

```
In [28]: fig = plt.figure("benaderen beginwaarde", figsize=(12,5))

         # eerste gokken verspreid over interval
         ax = fig[:add_subplot](1,2,1)
         plot_range = 1:31000
         for i in 1:6
             ax[:plot](plot_range/1e4, dists_init[i][plot_range], label="$(Y1[i])")
         end
         plt.legend(title="beginwaarde")
         plt.title("afstand tot beginwaarde in functie van de tijd")
         ax[:set_xlabel]("tijd")
         ax[:set_ylabel]("cartesische afstand tot beginwaarde")

         #-----------------------------------------------------------
         # fout op iteratiestappen rond T

         ax = fig[:add_subplot](1,2,2)
```

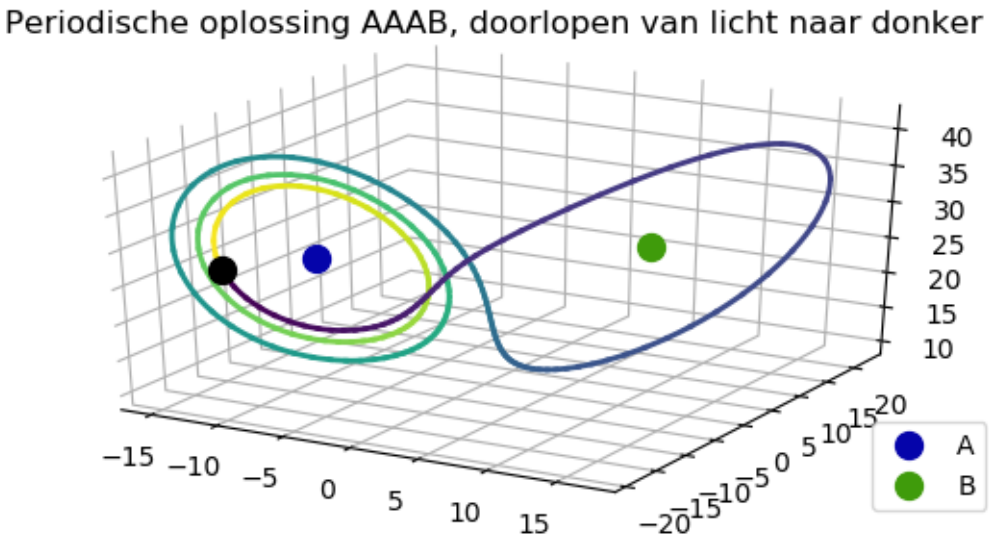
```
plot_range = 30200:30300
for i in 1:8
    ax[:plot](plot_range/1e4, log.(dists[i][plot_range]), label="$(results[i][1][2])")
end

plt.legend(title="beginwaarde - y(0)")
plt.title("afstand tot beginwaarde in functie van de tijd")
ax[:set_xlabel]("tijd")
ax[:set_ylabel]("log(afstand)")

plt.show()
fig[:tight_layout]()

fig[:savefig]("afstand_beginwaarde_opdracht4.pdf", dpi=400)
```



Plot oplossing

```
In [29]: X = sol[8][1,:];
         Y = sol[8][2,:];
         Z = sol[8][3,:];

         fig = figure("Lorenz attractor",figsize=(6, 3))
         ax = fig[:add_subplot](1,1,1, projection = "3d")

         len = 30200
         num = 200
         ran = 0:0.005:995
         stap = floor(Int64, len / num)
         c = mpl.cm[:viridis](ran)
         #c = mpl.cm[:plasma](ran)

         for i in 1:num
```

```
    ax[:plot3D](X[(i-1)*stap+1:i*stap], Y[(i-1)*stap+1:i*stap], Z[(i-1)*stap+1:i*stap]
end
ax[:plot3D]([X[1]], [Y[1]], [Z[1]], "ok", markersize=10)
ax[:plot3D]([A[1]], [A[2]], [A[3]], "o", markersize=10, color="xkcd:royal blue", label
ax[:plot3D]([B[1]], [B[2]], [B[3]], "o", markersize=10, color="xkcd:grass green", labe
#ax[:plot3D]([O[1]], [O[2]], [O[3]], "o", markersize=7, color="xkcd:maroon", label="O

ax[:xaxis][:set_pane_color]((1,1,1,0))
ax[:yaxis][:set_pane_color]((1,1,1,0))
ax[:zaxis][:set_pane_color]((1,1,1,0))
plt.legend(loc=4)
plt.title("Periodische oplossing AAAB, doorlopen van licht naar donker")
fig[:tight_layout]()

plt.show()

fig[:savefig]("periodische_baan_AAAB_opdracht4.pdf")
```



1.5.5 Exactere benadering met kleinere stapgrootte

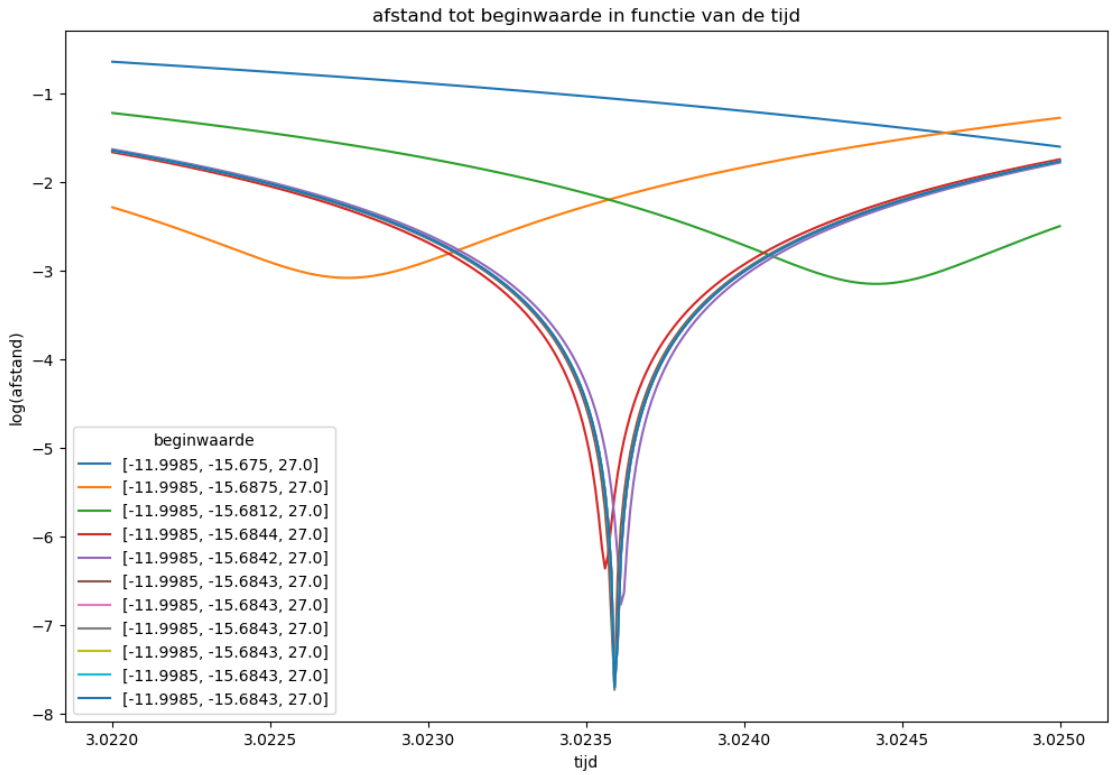
In [12]:
```
# zoek een betere benadering van de beginwaarde door een kleinere h te nemen
h = 1e-5
n = 320001
guess_I = [290000, 320000]
rounds = 25
results = find_y0(x0, z0, h, n, guess_y0, guess_I, rounds);
results[rounds][2]
```

Out[12]:
```
2.4886188244557686e-9
```

```
In [13]: # verwijder de dubbels
         results = unique(results);
```

Maak een plot van de opeenvolgende benaderingen, de cel hieronder wil je niet opnieuw runnen :p

```
In [14]: # vind de opeenvolgende oplossingen
         num = 11
         iter = 1:num

         X = [result[1] for result in results]
         h = fill(1e-5, num)
         n = fill(320001, num)
         GRAD = fill(gradient_s, num)
         sol = pmap(RK4_s, GRAD, X, h, n);
         # vind de afstanden tot de beginwaarde
         b0 = fill.(X, n)
         sols = [[sol[i][:,j] for j in 1:n[i]] for i in iter]
         dists = [pmap(norm, b0[i]-sols[i]) for i in iter];

In [15]: fig, ax = plt.subplots(1,1,figsize=(12,8))

         plot_range = 302200:302500
         for i in iter
             ax[:plot](plot_range/1e5, log.(dists[i][plot_range]), label="$(results[i][1])")
         end

         plt.legend(title="beginwaarde")
         plt.title("afstand tot beginwaarde in functie van de tijd")
         ax[:set_xlabel]("tijd")
         ax[:set_ylabel]("log(afstand)")
         fig[:savefig]("afstand_tot_beginwaarde_nauwkeurig.pdf", dpi=400)
         plt.show()
```



De heel exacte periode en beginwaarde

```
In [40]: find_closest(X[11], sol[11], (302000, 302500), 1e-5)

Out[40]: (3.023583703106596, 2.4886188244557686e-9, 302359)

In [39]: X[11]

Out[39]: 3-element Array{Float64,1}:
          -11.998523280062
          -15.684254097938537
           27.0
```

1.6 Oefening 5

Bereken een beginwaarde die in het vlak ligt waarnaar de oplossingen rond A worden getrokken. De vergelijking van dit vlak kunnen we schrijven als:

$$(\vec{x} - \vec{A}) \cdot \vec{v_1} = 0$$

Als we z i.f.v. x en y schrijven wordt dit:

$$z = \frac{-v_x(x - a_x) - v_y(y - a_y)}{v_z} + a_z$$

```
In [17]: v = eigA.vectors[:,1]
         a = np.matrix(A, dtype=np.float64)
         z_val(x, y) = (-v[1]*(x-a[1])-v[2]*(y-a[2]))/v[3] + a[3]

Out[17]: z_val (generic function with 1 method)
```

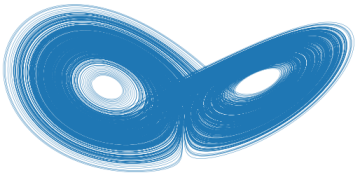
1.6.1 plot over lange tijd

```
In [22]: sp.pprint(A)
         # kies een beginwaarde in de buurt van A
         num=5
         x0 = [[-10 - i, -10 - i, z_val(-10 - i, -10 - i)] for i in 0:num];

In [23]: h = fill(1e-4, num+1)
         T = 500
         n = ceil.(Int64, T./h)
         sol = pmap(RK4_s, fill(gradient_s, num+1), x0, h, n);
```

Kies een interessante plot

```
In [49]: fig = figure("Lorenz attractor",figsize=(10, 7*num))
         ax = fig[:add_subplot](num,1,num, projection = "3d")

         for i in 1:num
             ax = fig[:add_subplot](num,1,i, projection = "3d")
             ax[:plot3D](sol[i][1,:], sol[i][2,:], sol[i][3,:], linewidth=0.3)
             ax[:set_title]("Plot $i")
             plt.axis("off")
         end

         plt.show()
         #fig[:savefig]("10 Lorenz Attractors.pdf", dpi=600)
```

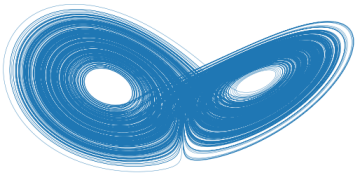
Plot 1



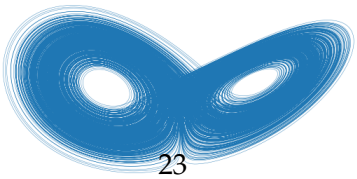
Plot 2

Plot 3

Plot 4

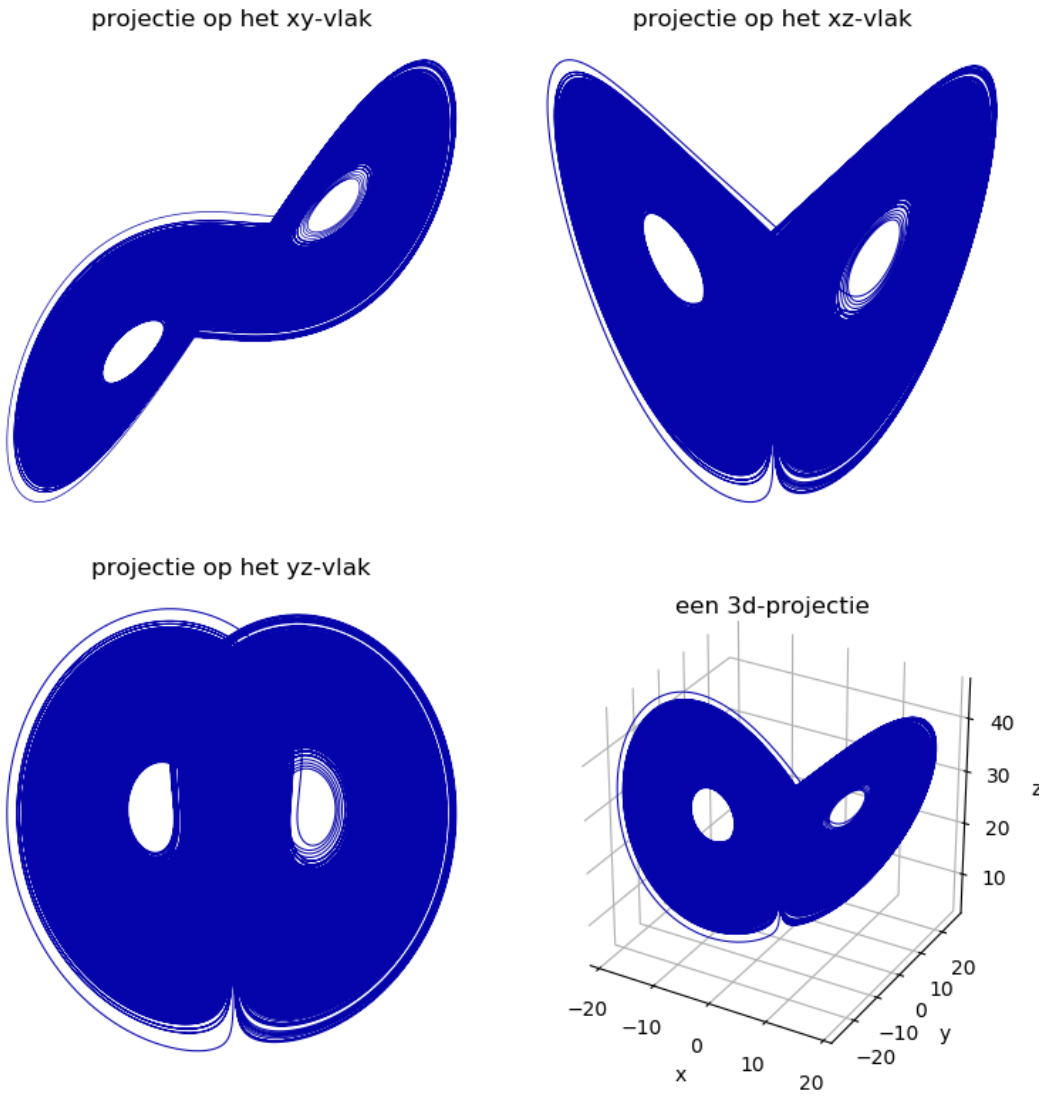


Plot 5



Maak een plot met de projecties op de coördinaatvlakken en de 3D-prjectie
Plot voor in verslag

```
In [37]: fig =figure("Lorenz Attractor", figsize=(8, 8))

         sol = RK4_s(gradient_s, x0[4], h[4], 3*n[4])

         X = sol[1,:]
         Y = sol[2,:]
         Z = sol[3,:]

         # projectie op xy-vlak
         ax = fig[:add_subplot](2,2,1)
         ax[:plot](X, Y, linewidth=0.5, color="xkcd:royal blue")
         ax[:set_title]("projectie op het xy-vlak")
         plt.axis("off")

         # projectie op xz-vlak
         ax = fig[:add_subplot](2,2,2)
         ax[:plot](X, Z, linewidth=0.75, color="xkcd:royal blue")
         ax[:set_title]("projectie op het xz-vlak")
         plt.axis("off")

         # projectie op yz-vlak
         ax = fig[:add_subplot](2,2,3)
         ax[:plot](Y, Z, linewidth=0.75, color="xkcd:royal blue")
         ax[:set_title]("projectie op het yz-vlak")
         plt.axis("off")

         # een 3D-projectie
         ax = fig[:add_subplot](2,2,4, projection = "3d")
         ax[:plot3D](X, Y, Z, linewidth=0.75, color="xkcd:royal blue")
         ax[:set_title]("een 3d-projectie")
         ax[:set_xlabel]("x")
         ax[:set_ylabel]("y")
         ax[:set_zlabel]("z")
         ax[:xaxis][:set_pane_color]((1,1,1,0))
         ax[:yaxis][:set_pane_color]((1,1,1,0))
         ax[:zaxis][:set_pane_color]((1,1,1,0))
         #plt.axis("off")

         fig[:tight_layout](pad=1, w_pad=0, h_pad=0)
         plt.show()

         fig[:savefig]("projecties_opdracht_5.pdf")
```



geheugen vrijmaken

```
In [34]: sol = nothing;
         sols = nothing;
         dists = nothing;
         dists_init = nothing;
         b0 = nothing;
         c = nothing;
         X = nothing;
         Y = nothing;
         Z = nothing;
```

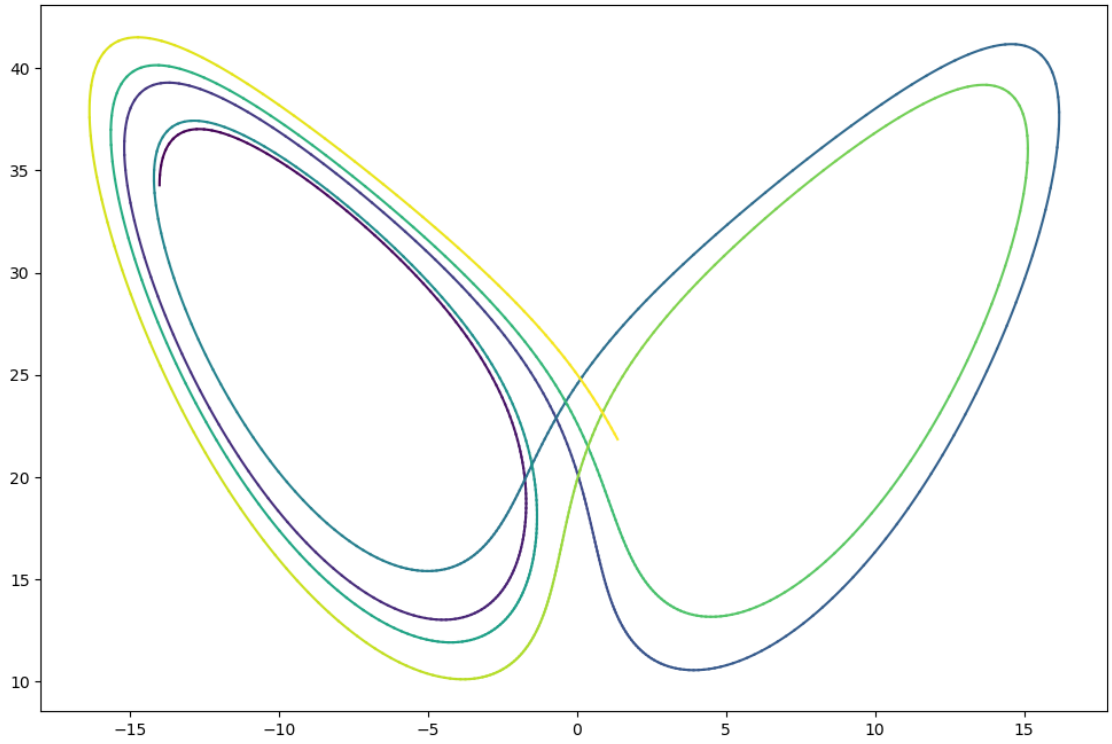
1.6.2 plot met colorgradient

```
In [21]: sp.pprint(A)
         # kies een beginwaarde in de buurt van A
         num=5
         x0 = [[-10 - i, -10 - i, z_val(-10 - i, -10 - i)] for i in 0:num];

In [55]: h = fill(1e-4, num+1)
         T = 5
         n = ceil.(Int64, T./h)
         sol = pmap(RK4_s, fill(gradient_s, num+1), x0, h, n);
```

Kies een interessante oplossing

Maak een plot met kleurgradiënt ifv de tijd, waar de kleur snel verandert, is de afgeleide klein en omgekeerd

```
In [56]: curve_num = 5
         X = sol[curve_num][1,:]
         Y = sol[curve_num][3,:]
         t = 1:n[curve_num]
         len = n[curve_num]
         num = 1000
         ran = 0:0.001:999
         stap = floor(Int64, len / num)

         c = mpl.cm[:viridis](ran)
         #c = mpl.cm[:plasma](ran)

         fig, ax = plt.subplots(1,1, figsize=(12, 8))

         for i in 1:num
             ax[:plot](X[(i-1)*stap+1:i*stap], Y[(i-1)*stap+1:i*stap], c=c[i,:])
         end

         plt.show()
```



1.7 Fancy plot header

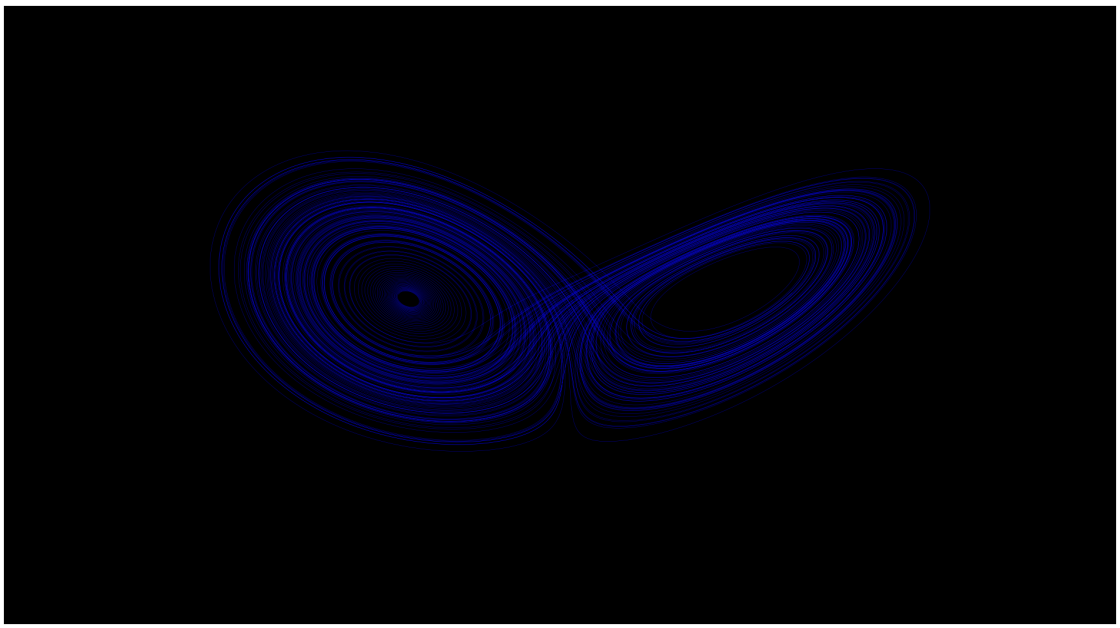
```
In [57]: x0 = [-9, -9, z_val(-9, -9)]
         h = 1e-4
         T = 100
         n = ceil(Int64, T/h)
         sol = RK4_s(gradient_s, x0, h, n);
```

Header notebook

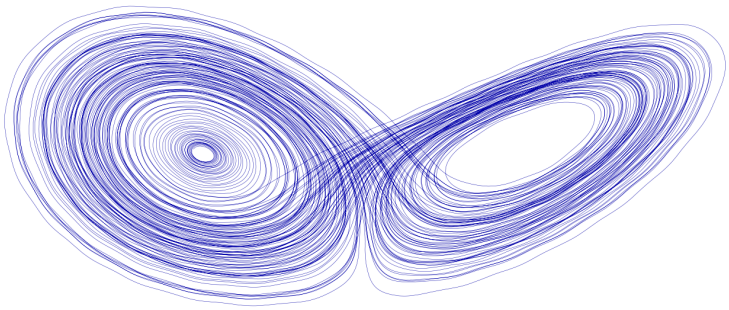
1.8 Wallpaper render

```
In [18]: x0 = [-9, -9, z_val(-9, -9)]
         h = 5e-5
         T = 151
         n = ceil(Int64, T/h)
         sol = RK4_s(gradient_s, x0, h, n);

In [18]: fig = plt.figure("blabla", figsize=(19.2, 10.8))

         ax = fig[:add_subplot](1,1,1, projection="3d")

         plot_interval = 1:floor(Int64, 3*10^6)
```

```
ax[:plot](sol[1,:][plot_interval], sol[2,:][plot_interval], sol[3,:][plot_interval], c
ax[:set_facecolor]("k")

plt.axis("off")
fig[:tight_layout]()
plt.show()

fig[:savefig]("wallpaper_render.png", dpi=400)
```



```
In [20]: fig = plt.figure("blabla", figsize=(19.2, 10.8))

         ax = fig[:add_subplot](1,1,1, projection="3d")

         plot_interval = 1:floor(Int64, 3*10^6)
         ax[:plot](sol[1,:][plot_interval], sol[2,:][plot_interval], sol[3,:][plot_interval], c
         #ax[:set_facecolor]("k")

         plt.axis("off")
         fig[:tight_layout]()
         plt.show()

         #fig[:savefig]("header_verslag_big.png", dpi=300)
         #fig[:savefig]("header_verslag_big.png", dpi=300)
```



1.9 Tests snelheid methodes

```
In [4]: # definieer een aantal beginvoorwaarden
        num_vgl = 3;
        X0 = [[-8, -8, -8] .- i for i in 1:num_vgl];
        # test met een stapgrootte van 10^-4 over een tijd van 5
        h = 1e-4;
        n = 5*10^4;
```

test de statische variant van Runge-Kutta en de tijdswinst door te paralleliseren

```
In [5]: function serial_test(X0)
            for x0 in X0
                RK4_s(gradient_s, x0, h, n)
            end
        end

Out[5]: serial_test (generic function with 1 method)

In [11]: # test de tijd om 1 vgl op te lossen
         @benchmark RK4_s(gradient_s, X0[1], h, n)

Out[11]: BenchmarkTools.Trial:
           memory estimate:  136.18 MiB
           allocs estimate:  1949963
           --------------
           minimum time:     113.006 ms (11.72% GC)
```

```
  median time:      113.946 ms (11.74% GC)
  mean time:        119.568 ms (14.54% GC)
  maximum time:     190.860 ms (43.07% GC)
  --------------
  samples:          42
  evals/sample:     1
```

```
In [12]: # test de tijd om num_vgl vergelijkingen na elkaar op te lossen
         @benchmark serial_test(X0)
```

```
Out[12]: BenchmarkTools.Trial:
  memory estimate:  408.55 MiB
  allocs estimate:  5849889
  --------------
  minimum time:     339.620 ms (11.95% GC)
  median time:      342.407 ms (12.62% GC)
  mean time:        352.716 ms (14.62% GC)
  maximum time:     416.955 ms (26.68% GC)
  --------------
  samples:          15
  evals/sample:     1
```

```
In [13]: # initialiseer arrays voor pmap
         GRAD = fill(gradient_s, num_vgl);
         H = fill(h, num_vgl);
         N = fill(n, num_vgl);
```

```
In [14]: @benchmark pmap(RK4_s, GRAD, X0, H, N)
```

```
Out[14]: BenchmarkTools.Trial:
  memory estimate:  3.49 MiB
  allocs estimate:  509
  --------------
  minimum time:     158.207 ms (0.00% GC)
  median time:      166.649 ms (0.00% GC)
  mean time:        193.798 ms (1.61% GC)
  maximum time:     280.081 ms (0.00% GC)
  --------------
  samples:          26
  evals/sample:     1
```

1.9.1 Herhaal test over langere tijd

```
In [46]: # definieer een aantal beginvoorwaarden
         num_vgl = 3;
         X0 = [[-8, -8, -8] .- i for i in 1:num_vgl];
         # test met een stapgrootte van 10^-4 over een tijd van 5
         h = 1e-4;
         n = 5*10^5;
```

test de statische variant van Runge-Kutta en de tijdswinst door te paralleliseren

```
In [47]: function serial_test(X0)
             for x0 in X0
                 RK4_s(gradient_s, x0, h, n)
             end
         end

Out[47]: serial_test (generic function with 1 method)

In [48]: # test de tijd om 1 vgl op te lossen
         @benchmark RK4_s(gradient_s, X0[1], h, n)

Out[48]: BenchmarkTools.Trial:
           memory estimate:  1.33 GiB
           allocs estimate:  19499963
           --------------
           minimum time:     1.250 s (14.45% GC)
           median time:      1.273 s (14.29% GC)
           mean time:        1.289 s (15.54% GC)
           maximum time:     1.362 s (18.87% GC)
           --------------
           samples:          4
           evals/sample:     1

In [49]: # test de tijd om num_vgl vergelijkingen na elkaar op te lossen
         @benchmark serial_test(X0)

Out[49]: BenchmarkTools.Trial:
           memory estimate:  3.99 GiB
           allocs estimate:  58499889
           --------------
           minimum time:     3.981 s (14.86% GC)
           median time:      4.140 s (15.18% GC)
           mean time:        4.140 s (15.18% GC)
           maximum time:     4.298 s (15.47% GC)
           --------------
           samples:          2
           evals/sample:     1

In [50]: # initialiseer arrays voor pmap
         GRAD = fill(gradient_s, num_vgl);
         H = fill(h, num_vgl);
         N = fill(n, num_vgl);

In [51]: @benchmark pmap(RK4_s, GRAD, X0, H, N)

Out[51]: BenchmarkTools.Trial:
           memory estimate:  34.42 MiB
```

```
  allocs estimate:  946
  --------------
  minimum time:     1.839 s (0.10% GC)
  median time:      1.904 s (0.09% GC)
  mean time:        1.962 s (0.11% GC)
  maximum time:     2.142 s (0.08% GC)
  --------------
  samples:          3
  evals/sample:     1
```

1.10 vergelijk met python functie

```
In [15]: py"""
         import sympy as sp
         import numpy as np
         # gebruikt de methode van Runge-Kata van orde 4 om een benaderende oplossing te geven
         # stapgrootte h en n stappen
         def RK4(f, x0, h, n):
             x = np.zeros((n, x0.size))
             x[0] = x0

             for i in range(1, n):
                 k1 = f(0, x[i-1])
                 k2 = f(0, x[i-1] + h*k1/2)
                 k3 = f(0, x[i-1] + h*k2/2)
                 k4 = f(0, x[i-1] + h*k3)

                 x[i] = x[i-1] + h/6*(k1+2*k2+2*k3+k4)
             return x

         t, x, y, z = sp.symbols('t, x, y, z')
         sigma=10
         r = 28
         b = 8/3
         dx = -sigma*x+sigma*y
         dy = r*x-y-x*z
         dz = -b*z+x*y

         # definieer de componentsfuncties
         fx = sp.lambdify([x, y, z], dx, 'numpy')
         fy = sp.lambdify([x, y, z], dy, 'numpy')
         fz = sp.lambdify([x, y, z], dz, 'numpy')

         # evalueert f van het autonoom stelsel in x
         def f(t, x):
             grad = np.zeros(3)
             grad[0] = fx(*x)
             grad[1] = fy(*x)
```

```
        grad[2] = fz(*x)
        return grad
    """

    RK4_py = py"RK4"
    f_py = py"f"
```

```
Out[15]: PyObject <function f at 0x7fab69704488>
```

```
In [10]: # definieer een aantal beginvoorwaarden
         num_vgl = 3;
         X0 = [[-8, -8, -8] .- i for i in 1:num_vgl];
         # test met een stapgrootte van 10^-4 over een tijd van 5
         h = 1e-4;
         n = 5*10^4;
```

```
In [17]: @benchmark RK4_py(f_py, X0[1], h, n)
```

```
Out[17]: BenchmarkTools.Trial:
           memory estimate:  1.15 MiB
           allocs estimate:  82
           --------------
           minimum time:     1.896 s (0.00% GC)
           median time:      1.985 s (0.00% GC)
           mean time:        2.056 s (0.00% GC)
           maximum time:     2.286 s (0.00% GC)
           --------------
           samples:          3
           evals/sample:     1
```

```
In [18]: # test de tijd om 1 vgl op te lossen
         @benchmark RK4_s(gradient_s, X0[1], h, n)
```

```
Out[18]: BenchmarkTools.Trial:
           memory estimate:  136.18 MiB
           allocs estimate:  1949963
           --------------
           minimum time:     114.618 ms (12.17% GC)
           median time:      122.952 ms (12.04% GC)
           mean time:        126.119 ms (14.82% GC)
           maximum time:     199.170 ms (44.67% GC)
           --------------
           samples:          40
           evals/sample:     1
```

```
In [19]: function py_serial_test(X0)
             for x0 in X0
                 RK4_s(f_py, x0, h, n)
             end
         end
```

```
Out[19]: py_serial_test (generic function with 1 method)
```

```
In [ ]: @benchmark py_serial_test(X0)
```