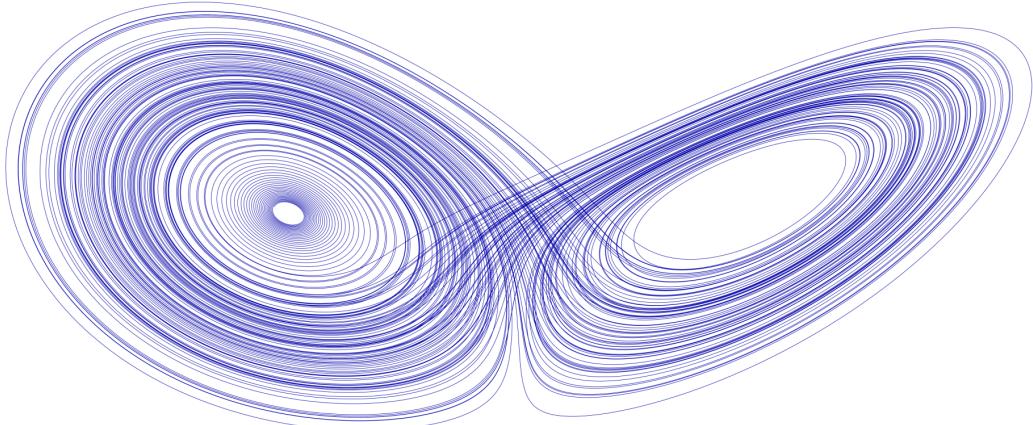


Practicum differentiaalvergelijkingen: De Lorenz Attractor

Pieter Luyten - r0708257

November 2018



1 Inleiding

In dit practicum bestuderen we het gedrag van de Lorenz attractor voor specifieke waarden voor de parameters. De Lorenz attractor wordt beschreven door een stelsel van 3 differentiaalvergelijkingen:

$$\begin{aligned}x'(t) &= -\sigma x + \sigma y \\y'(t) &= rx - y - xz \\z'(t) &= -bx + xy\end{aligned}\tag{1}$$

Hierin zijn σ, r, b 3 reële parameters. We stellen deze voor de rest van dit document gelijk aan:

$$\sigma = 10 \quad r = 28 \quad b = 8/3$$

Dit is een sterk vereenvoudigd model van het weer, een zogenaamd "speelgoedmodel". Dit werd voor het eerst bestudeerd door Edward Lorenz. Het is uit dit model dat de term vlindereffect is ontstaan. De oplossingen zijn over het algemeen heel chaotisch en een heel kleine aanpassing aan de beginvoorwaarde heeft na lange tijd een heel groot effect. De afbeelding in de header toont een pad op de Lorenz attractor, het oppervlak waar de oplossingen naar toe worden getrokken.

Deze differentiaalvergelijking werd bestudeerd met numerieke methoden. Ik heb IJulia notebooks gebruikt voor de numerieke methoden om de vergelijking op te lossen en verdere analyse. Voor de analyse van de kritieke punten heb ik het package sympy gebruikt om met symbolische vergelijkingen te werken. Ik heb ook een aantal scripts gebruikt met meer efficiënte methodes om beginwaardes te zoeken. De gebruikte code staat in de bijlage, de notbook vind je ook hier online.

2 Kritieke punten

We berekenen eerst de kritieke punten van het systeem van differentiaalvergelijkingen. Dit zijn punten waarin elke vergelijking van het stelsel gelijk is aan 0. In dit geval zijn er 3 kritieke punten: de oorsprong (O), een punt met negatieve x -coördinaat (A) en een punt met positieve x -coördinaat (B). De coördinaten van de kritieke punten zijn:

$$\begin{aligned}O &= (0, \quad 0, \quad 0) \\A &= \left(-6\sqrt{2}, \quad -6\sqrt{2}, \quad 27 \right) \\B &= \left(6\sqrt{2}, \quad 6\sqrt{2}, \quad 27 \right)\end{aligned}$$

Voor de analyse van de kritieke punten bepalen we de eigenwaarden en bijhorende eigenvectoren van de Jacobiaan in deze punten. Dit geeft:

$$\begin{aligned}
 A : \lambda_1 &= -13.85, \vec{v}_1 = \begin{bmatrix} 0.85 \\ -0.32 \\ 0.39 \end{bmatrix}; \lambda_2 = 0.0939 + 10.1945i, \vec{v}_2 = \begin{bmatrix} -0.26 - 0.29i \\ 0.032 - 0.56i \\ 0.719 \end{bmatrix}; \\
 \lambda_3 &= 0.0939 - 10.1945i, \vec{v}_2 = \begin{bmatrix} -0.26 + 0.29i \\ 0.032 + 0.56i \\ 0.719 \end{bmatrix} \\
 B : \lambda_1 &= -13.85, \vec{v}_1 = \begin{bmatrix} 0.85 \\ -0.32 \\ -0.39 \end{bmatrix}; \lambda_2 = 0.0939 + 10.1945i, \vec{v}_2 = \begin{bmatrix} -0.26 - 0.29i \\ 0.032 - 0.56i \\ -0.719 \end{bmatrix}; \\
 \lambda_3 &= 0.0939 - 10.1945i, \vec{v}_2 = \begin{bmatrix} -0.26 + 0.29i \\ 0.032 + 0.56i \\ -0.719 \end{bmatrix} \\
 O : \lambda_1 &= -22.827, \vec{v}_1 = \begin{bmatrix} -0.614 \\ 0.788 \\ 0 \end{bmatrix}; \lambda_2 = 11.827, \vec{v}_2 = \begin{bmatrix} -0.416 \\ -0.909 \\ 0 \end{bmatrix}; \lambda_3 = -\frac{8}{3}, \vec{v}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}
 \end{aligned}$$

Bij de punten A en B zien we dat er 1 negatieve en 2 complexe eigenwaarden zijn. De eigenwaarde λ_1 is negatief, de oplossingen zullen naar de punten getrokken worden in de richting van \vec{v}_1 . De eigenwaarden bij de andere 2 eigenvectoren zijn complex, in het vlak opgespannen door (de reële delen van) deze eigenvectoren hebben we een spiraalpunt. Dit is een onstabiel spiraalpunt omdat het reële deel van de eigenwaarden positief is.

In het punt O zijn er 3 reële eigenwaarden, 1 negatief en 2 positief, de component van de richtingsvector volgens \vec{v}_1 zal dus negatief zijn en de oplossingen naar het vlak door O opgespannen door \vec{v}_2 en \vec{v}_3 trekken. In dit vlak hebben we een bron want de eigenwaarden bij de 2 eigenvectoren in dit vlak zijn positief. Dit kritiek punt is dus niet stabiel en zeker niet asymptotisch stabiel.

We zien dus dat de oplossingen naar de vlakken opgespannen door de complexe eigenvectoren van de Jacobiaan in A en B worden getrokken. In de buurt van het onstabiele kritieke punt O kunnen ze van vlak wisselen.

3 Periodische oplossingen

Hoewel het gedrag van de oplossingen van vergelijking [1] in het algemeen chaotisch is, zijn er oplossingen met specifieke beginvoorwaarden die een periodiek gedrag vertonen.

Dit periodiek gedrag duiden we aan met een sequentie van A's en B's die zegt hoe vaak en in welke volgorde de oplossing rond elk punt draait.

3.1 Oplossing AB

Voor de beginvoorwaarde $\mathbf{x}(0)$:

$$x(0) = 13,763610682134, \quad y(0) = 19,578751942452, \quad z(0) = 27$$

vinden we een periodieke oplossing AB. Deze draait dus een keer rond A, een keer rond B en komt dan terug op de beginwaarde uit. Deze beginwaarde kunnen we gebruiken om de nauwkeurigheid van numerieke methodes te testen. We benaderen de oplossing met 3 numerieke methodes: de methode van Euler, de verbeterde methode van Euler en Runge-Kutta van orde 4. Voor elke methode zoeken we dan de stapgrootte die nodig is om de fout op de eindwaarde kleiner dan 10^{-5} te krijgen.

De functies om de oplossing numeriek te benaderen geven als return value een lijst met opeenvolgende punten van de oplossing. In deze lijst zoeken we het punt dat het dichtst bij de beginwaarde ligt, noem dit punt D , het punt ervoor C en het punt na D noemen we E . Deze afstand is echter niet altijd de kleinste afstand tot de beginwaarde, hiervoor moeten we de afstand van de beginwaarde tot de lijnstukken $[CD]$ en $[DE]$ ook bepalen. Als deze afstand kleiner is, en de loodrechte projectie van de beginwaarde op dit lijnstuk ligt, is dit de kleinste afstand tot de beginwaarde. Als de 2 loodrechte projecties niet op het lijnstuk liggen, gebruiken we de tijd tot het dichtste punt als benadering van de periode. In het andere geval projecteren we $\mathbf{x}(0)$ op het lijnstuk waar het het dichtste bij ligt, noem deze projectie P . We berekenen dan de verhouding $-\frac{|DP|}{|CD|}$ of $\frac{|DP|}{|DE|}$. Dit vermenigvuldigen we met de stapgrootte h en tellen we op bij de tijd tot het punt D om een schatting voor de periode T van de periodische baan te krijgen. De min bij $-\frac{|DP|}{|CD|}$ staat er omdat dit punt "voor" het punt D ligt als je de oplossingskromme doorloopt

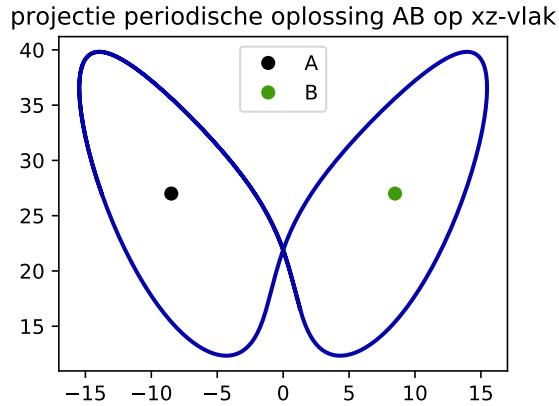
Deze methode geeft volgende waarden voor h en T zodat $|\mathbf{x}(0) - \mathbf{x}(T)| < 10^{-5}$

$$\text{methode van Euler : } h = 3 \cdot 10^{-7} \quad T = 1,5586, \text{afwijking} = 7,26 \cdot 10^{-6}$$

$$\text{verbeterde methode van Euler : } h = 8 \cdot 10^{-4} \quad T = 1,55864188, \text{afwijking} = 8,77 \cdot 10^{-6}$$

$$\text{Runge-Kutta van orde 4 : } h = 3 \cdot 10^{-4} \quad T = 1,55865233, \text{afwijking} = 8,48 \cdot 10^{-6}$$

De differentiaalvergelijking heel exact benaderen met Runge-Kutta van orde 4 en stapgrootte 10^{-7} geeft een periode van $T = 1.558652210716$. In figuur [1] zie je de projectie van de periodische oplossing op het xz -vlak.



Figuur 1: De projectie in het xz-vlak van een periodische oplossing AB

3.2 Oplossing AAAAB

Er is gegeven dat voor de beginwaarden

$$x(0) = 11,998523280062, \quad y(0) \in [-16, -15] \quad z(0) = 27 \quad (2)$$

er een periodische oplossing is van de vorm AAAAB. Het gedrag van de oplossingen is chaotisch maar we kunnen wel de continuïteit in de beginvoorwaarde gebruiken. Dit wordt analoog bewezen als de continuïteit in de beginvoorwaarde van 1 differentiaalvergelijking.

Stel dat we een stelsel hebben van de vorm $\mathbf{x}'(t) = \mathbf{f}(t, \mathbf{x})$ met (\mathbf{f}) continu op $[a, A] \times D$ en uniform Lipschitz continu op D . Stel dat \mathbf{x}_1 en \mathbf{x}_2 oplossingen zijn van de differentiaalvergelijking zodat voor alle $t \in [a, A]$ geldt dat $y_1, y_2 \in D$. Dan bestaat er voor elke $\epsilon > 0$ een $\delta > 0$ zodat $\|\mathbf{x}_1(t) - \mathbf{x}_2(t)\| < \epsilon$ voor alle $t \in [a, A]$ volgt uit $\|\mathbf{x}_1(a) - \mathbf{x}_2(a)\| < \delta$.

Bewijs. Stel $\phi(x) = \|\mathbf{x}_1(t) - \mathbf{x}_2(t)\|^2$. voor de afgeleide geldt als $\mathbf{x}_1(t) \neq \mathbf{x}_2(t)$ dat:

$$\begin{aligned} \phi'(t) &\leq 2\|\mathbf{x}_1(t) - \mathbf{x}_2(t)\| \|\mathbf{x}'_1(t) - \mathbf{x}'_2(t)\| \\ &= 2\|\mathbf{x}_1(t) - \mathbf{x}_2(t)\| \|\mathbf{f}(t, \mathbf{x}_1) - \mathbf{f}(t, \mathbf{x}_2)\| \\ &\leq 2K\|\mathbf{x}_1(t) - \mathbf{x}_2(t)\|^2 && = 2K\phi(x) \end{aligned}$$

Als norm gebruiken we de euclidische norm, deze is overal afleidbaar behalve in 0, maar omdat we $\mathbf{x}_1(t) \neq \mathbf{x}_2(t)$ veronderstellen kunnen we de kettingregel toepassen. In de laatste stap gebruiken we de Lipschitz-continuïteit van \mathbf{f} op D en is K de Lipschitz-constante. We kunnen nu het Lemma van Gronwall gebruiken op de functies $\phi(x)$ en $g(x) = 2K$. Dit geeft dan dat:

$$0 \leq \phi(x) \leq \phi(a) \exp(2K(t-a)).$$

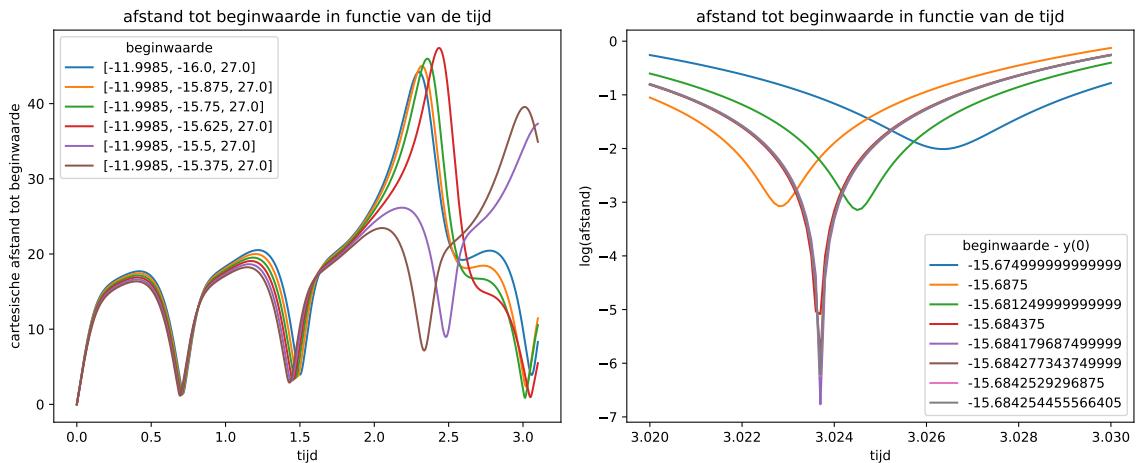
Hieruit volgt dan dat:

$$\|\mathbf{x}_1(t) - \mathbf{x}_2(t)\| \leq \|\mathbf{x}_1(a) - \mathbf{x}_2(a)\| \exp(K(A-a))$$

stel $\delta = \epsilon \exp(-K(a-A))$, dan volgt het gevraagde voor $\mathbf{x}_1 \neq \mathbf{x}_2$. In het geval dat $\mathbf{x}_1(t) = \mathbf{x}_2(t)$ voor minstens 1 t volgt uit de uniciteit van de oplossing dat $\mathbf{x}_1 = \mathbf{x}_2$, in dit geval geldt zeker het gevraagde. \square

De differentiaalvergelijking van het model van Lorenz heeft een continue afgeleide over heel \mathbb{R}^3 . Als we $t \in [0, T]$ bekijken zal de oplossing begrensd blijven. Deze ligt dus in een gesloten gebied D . Omdat D gesloten is zal in elk punt elke richtingsafgeleide begrensd blijven, en kunnen we deze afschatten met 1 K . Dan volgt dat \mathbf{f} Lipschitz continu is door voor elke \mathbf{x}, \mathbf{y} de rechte tussen \mathbf{x}, \mathbf{y} te bekijken en de richtingsafgeleide langs deze rechte.

Dit kunnen we gebruiken om de periodische oplossing te zoeken. Eerst berekenen we voor 10 beginwaarden in het interval [2] de oplossing en de afstand tot de beginwaarde in functie van de tijd. Dit kan je zien in de linkse grafiek van figuur [2]. Het is duidelijk dat vooral de rode, groene en oranje grafiek de beginwaarde terug dicht naderen rond $T = 3$ na 2 lussen rond A .



Figuur 2: Links: afstand tot de beginwaarde in functie van de tijd voor een aantal beginwaarden met $y(0) \in [-16, -15]$. Rechts: de afstand tot de beginwaarde in functie van de tijd, rond het tijdstip waarop de oplossing het dichtst komt voor opeenvolgende iteraties van de verfijningsmethode

We zetten nu een iteratieproces op waarbij we steeds 5 oplossingen bekijken. Nummer deze \mathbf{x}_i zodat als $i < j$ dan $y_i(0) < y_j(0)$. Van deze 5 oplossingen zoeken we de oplossing die rond $T = 3$ het dichtst (zijn) beginwaarde terug benadert, stel dat dit \mathbf{x}_i

is. In de volgende iteratiestap stellen we $y_1(0) = y_{i-1}(0)$, $y_3(0) = y_i(0)$, $y_5(0) = y_{i+1}(0)$. Indien $i \in \{1, 5\}$ stel dan i gelijk aan 2 respectievelijk 4. We berekenen nu steeds 2 nieuwe oplossingen met als y -waarde voor de beginwaarde

$$y_2(0) = \frac{y_3(0) - y_1(0)}{2}, \quad y_4(0) = \frac{y_5(0) - y_3(0)}{2}$$

Via dit iteratieproces vinden we de opeenvolgende y -waarden in de rechtse grafiek van figuur [2] en dat de periode iets minder dan 3,024 is.

Met behulp van een verbeterde methode proberen we de precisie te verhogen. In plaats van 5 oplossingen in een interval te beschouwen, worden er nu 7 gebruikt. Voor de rest werkt de procedure op dezelfde manier. In elke iteratiestap moeten dus 4 nieuwe oplossingen berekend worden. De beginwaarden hiervoor zijn:

$$\begin{aligned} y_2(0) &= \frac{2y_1(0) + y_4(0)}{3}, & y_3(0) &= \frac{y_1(0) + 2y_4(0)}{3} \\ y_5(0) &= \frac{2y_4(0) + y_7(0)}{3}, & y_6(0) &= \frac{y_4(0) + 2y_7(0)}{3} \end{aligned}$$

De 4 nieuwe oplossingen worden parallel berekend zodat het script optimaal gebruikt maakt van de 4 processorkernen. De code die hiervoor gebruikt is staat niet in de notebook maar zijn een aantal scripts die volledig achteraan dit document in sectie 5.2 toegevoegd als bijlage zijn.

$$y_0 = -15,684\,254\,096\,883$$

Met 100% zekerheid ligt de beginwaarde voor y in volgend interval, namelijk het laatste interval van de iteratieprocedure:

$$[-15,684\,254\,096\,881\,666; -15,684\,254\,096\,884\,999].$$

De bijhorende periode is:

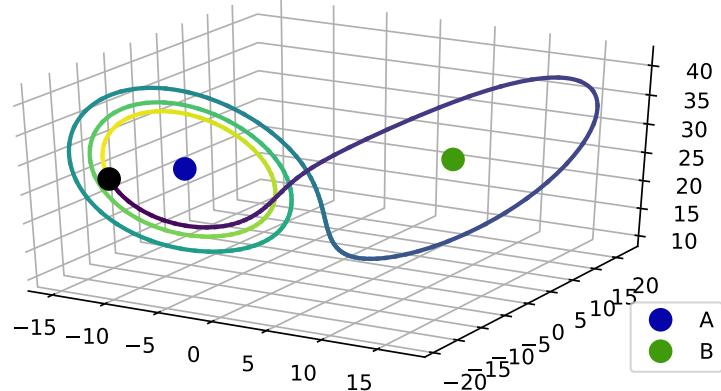
$$T = 3,023\,583\,703\,434$$

Figuur [3] toont een plot van de periodische oplossing.

4 Vorm van de Lorenz attractor

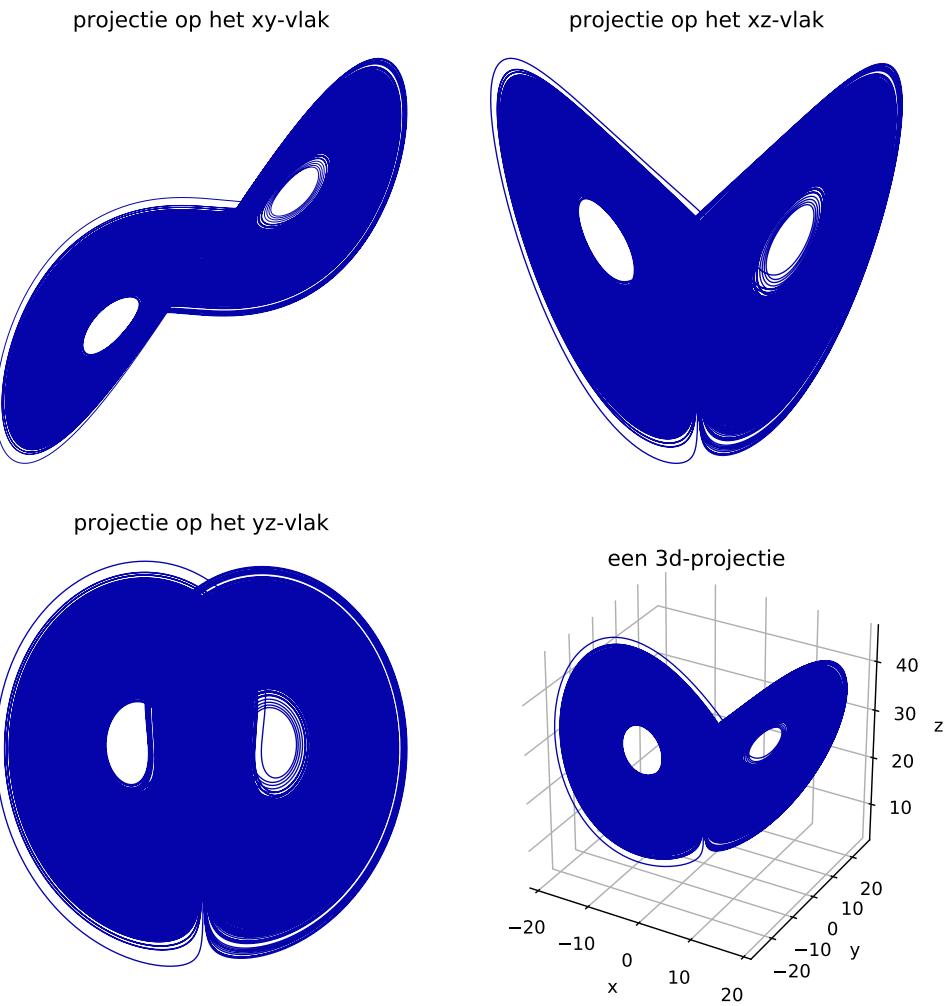
Nu plotten we een oplossing op de Lorenz attractor over een heel lange tijd om een idee te krijgen van de vorm van het oppervlak. In figuur [4] zie je een oplossing met

Periodische oplossing AAAB, doorlopen van licht naar donker



Figuur 3: Plot van een periodische baan AAAB. De kromme wordt doorlopen van geel naar paars (licht naar donker)

$t \in [0, 1500]$ geprojecteerd op de 3 coördinaatvlakken en een 3d-projectie. We zien duidelijk dat heel dicht bij de kritieke punten A en B , en verder van de punten vandaan de banen steeds minder dicht zijn. Ook valt op dat de banen in de vlakken rond de kritieke punten blijven en wisselen van vlak in het kritieke punt 0.



Figuur 4: Projecties van een pad op de Lorenz attractor voor $T \rightarrow \infty$

5 Bijlages

5.1 De Notebook

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  $x_0$ 
        # 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, arg)

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

```

    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 afgeleide 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 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

```

```

        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

```

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 begin en end tot het punt point
begin, end en point moeten 1d vectoren van dezelfde dimensie zijn
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 rechte 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
    # initiele 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(0))

\left( -6 \sqrt{2}, -6 \sqrt{2}, 27\right)
\left( 6 \sqrt{2}, 6 \sqrt{2}, 27\right)
\left( 0, 0, 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)[:,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];
sub0 = [vars[i] => 0[i] for i in 1:3];
JA = num_jacobian(funcs, vars, subA);
JB = num_jacobian(funcs, vars, subB);
J0 = num_jacobian(funcs, vars, sub0);
```

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

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

Out[13]: 7038

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

Out[14]: 7046

```
In [15]: Jo = sp.Matrix(J0, dtype=np.float64);
eig0 = Jo[:eigenvecs]();
write("eigvecs0.txt", sp.latex(eig0))
```

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 projectie
        # 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 vergelijken
    # 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, stapgrootte: %.2e\n",
            Ts[i], x_dists[i], h[i])
end
```

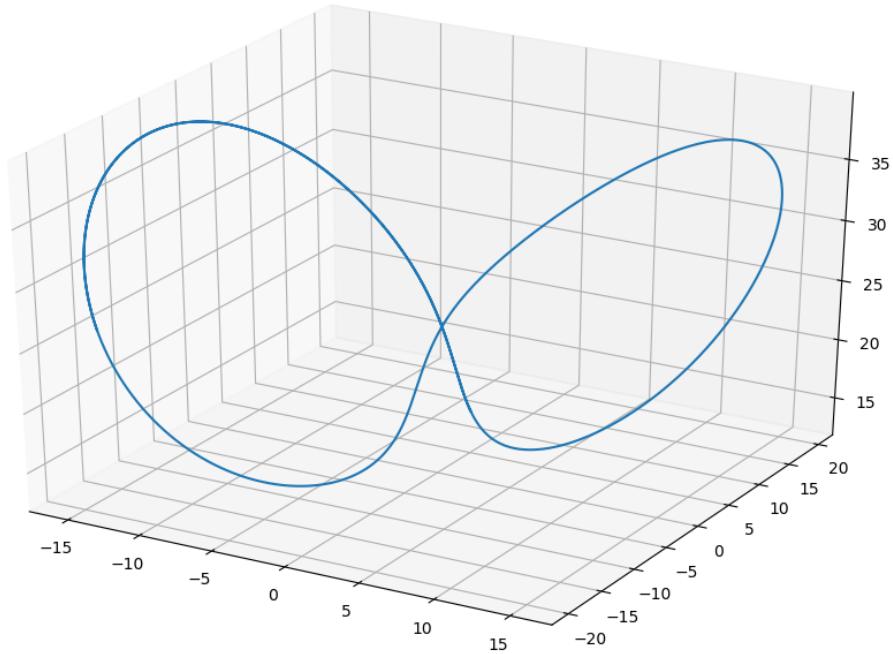
```
Verbeterde methode van Euler - periode: 1.55865157, x-afwijking: 5.60e-07, stapgrootte: 2.00e-04
Verbeterde methode van Euler - periode: 1.55864967, x-afwijking: 4.42e-06, stapgrootte: 4.00e-04
Verbeterde methode van Euler - periode: 1.55864638, x-afwijking: 3.00e-06, stapgrootte: 6.00e-04
Verbeterde methode van Euler - periode: 1.55864188, x-afwijking: 8.77e-06, stapgrootte: 8.00e-04
Verbeterde methode van Euler - periode: 1.55863611, x-afwijking: 1.86e-05, stapgrootte: 1.00e-03
Verbeterde methode van Euler - periode: 1.55862807, x-afwijking: 3.62e-05, stapgrootte: 1.20e-03
Verbeterde methode van Euler - periode: 1.55862020, x-afwijking: 1.12e-05, stapgrootte: 1.40e-03
Verbeterde methode van Euler - periode: 1.55860887, x-afwijking: 8.47e-05, stapgrootte: 1.60e-03
Verbeterde methode van Euler - periode: 1.55859686, x-afwijking: 1.35e-04, stapgrootte: 1.80e-03
```

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()
```

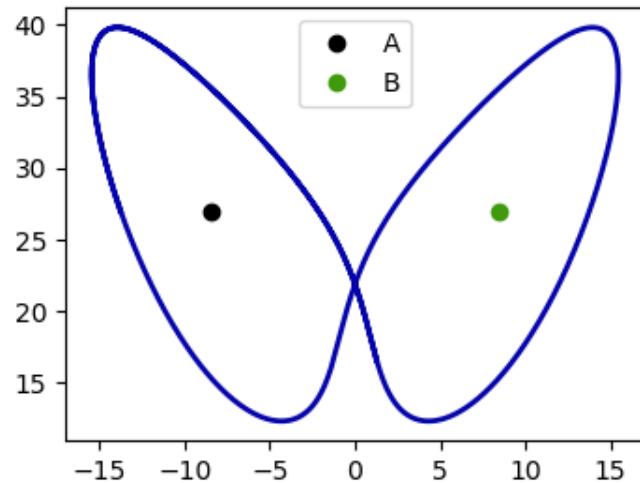


```
In [18]: fig, ax = plt.subplots(1,1,figsize=(4, 3))

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")
```

projectie periodische oplossing AB op xz-vlak



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 oplossing
    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 de
        @returns:
            lijst met steeds betere benaderingen van de beginwaarde + hoe die
            =#
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
    # 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]], GUI)
(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

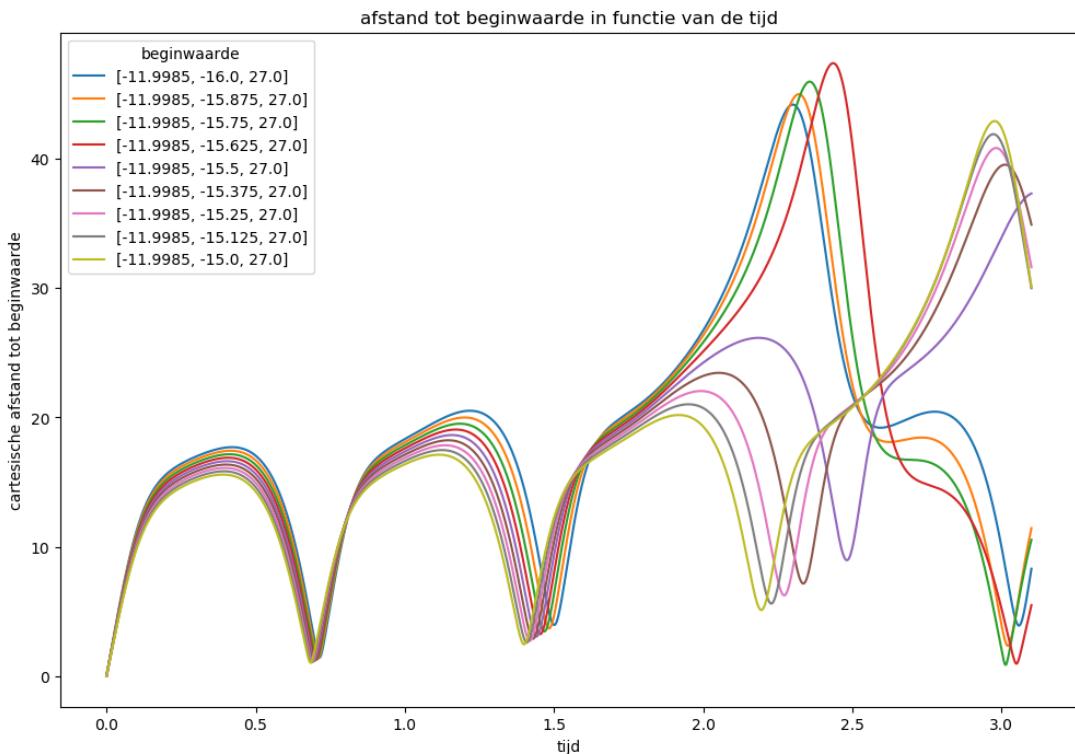
```

```

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)
```

```

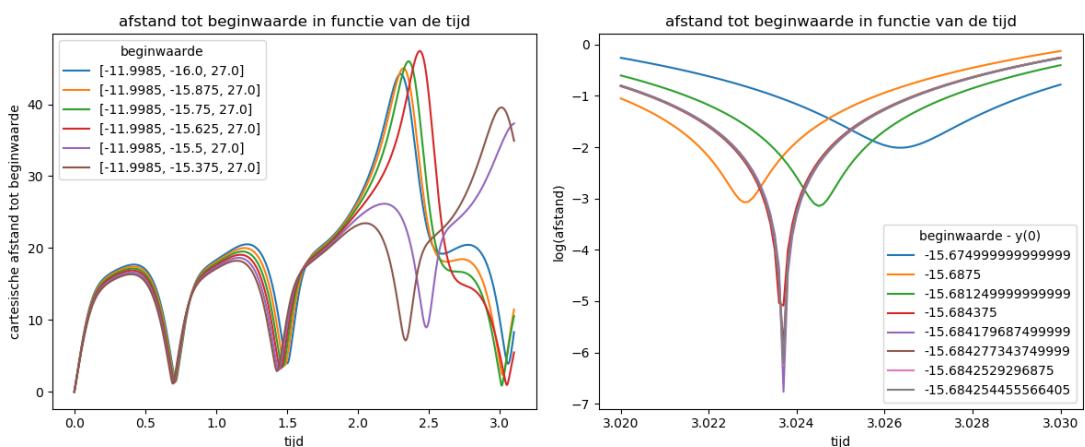
plot_range = 30200:30300
for i in 1:8
    ax[:plot](plot_range/1e4, log.(dists[i][plot_range]), label=results[i][1][2])
end

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="A")
ax[:plot3D]([B[1]], [B[2]], [B[3]], "o", markersize=10, color="xkcd:grass green", label="B")
#ax[:plot3D]([O[1]], [O[2]], [O[3]], "o", markersize=7, color="xkcd:maroon", label="O")

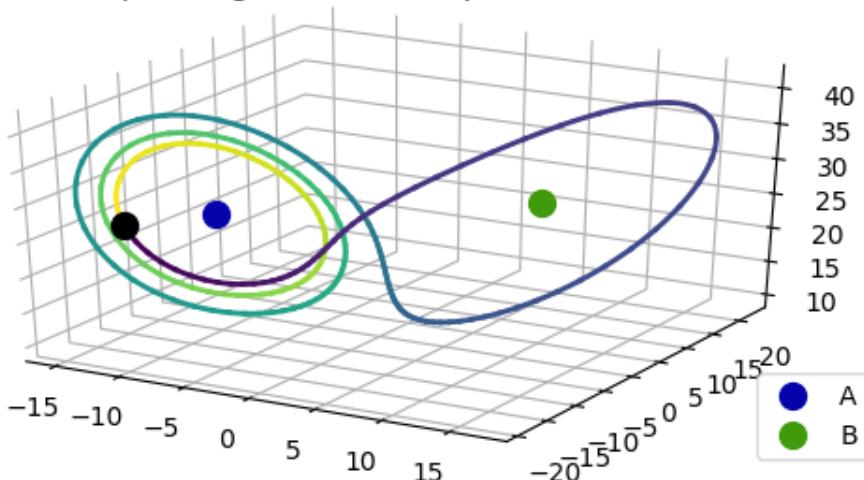
ax[:xaxis][:set_pane_color]((1,1,1,0))
ax[:yaxis][:set_pane_color]((1,1,1,0))
ax[:zaxis][:set_pane_color]((1,1,1,0))
plt.legend(loc=4)
plt.title("Periodische oplossing AAAB, doorlopen van licht naar donker")
fig[:tight_layout]()

plt.show()

fig[:savefig]("periodische_baan_AAA_B_opdracht4.pdf")

```

Periodische oplossing AAAB, doorlopen van licht naar donker



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

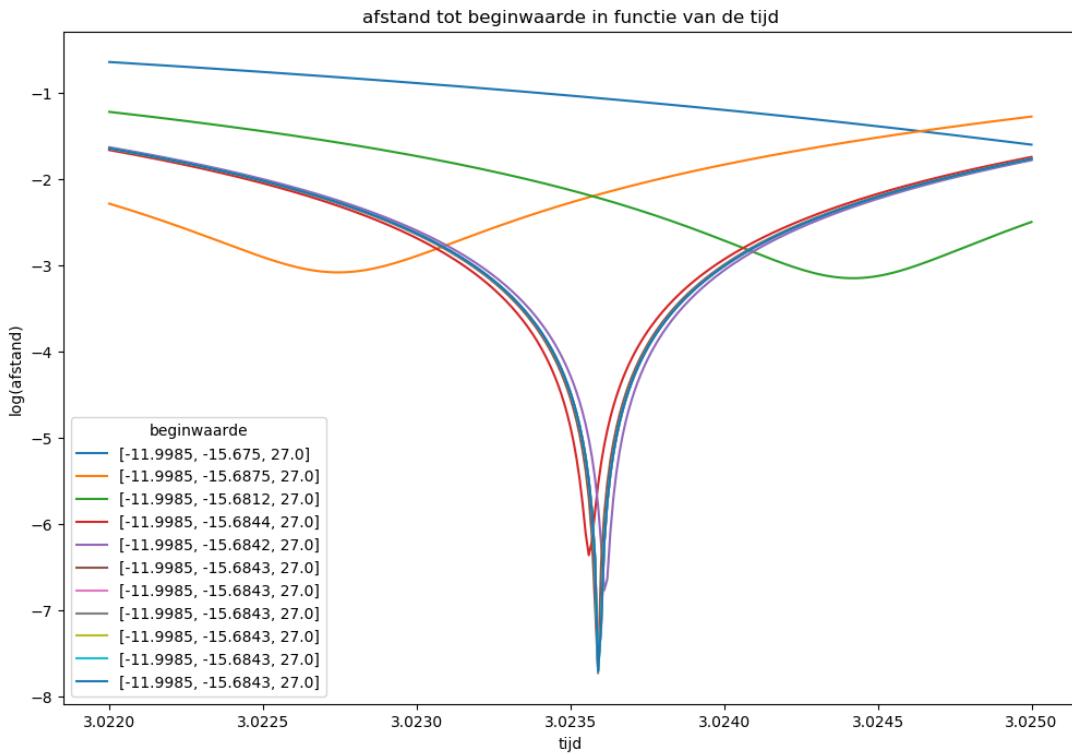
```
In [14]: # vind de opeenvolgende oplossingen
num = 11
iter = 1:num

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);
```

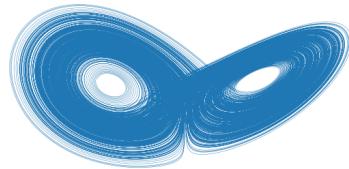
Kies een interessante plot

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

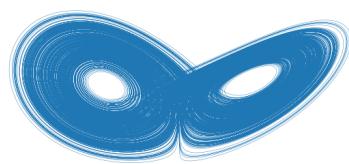
for i in 1:num
    ax = fig[:add_subplot](num, 1, i, projection = "3d")
    ax[:plot3D](sol[i][1,:], sol[i][2,:], sol[i][3,:], linewidth=0.3)
    ax[:set_title]("Plot $i")
    plt.axis("off")
end

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

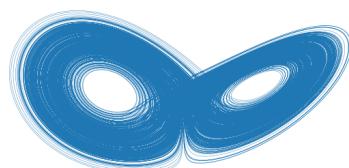
Plot 1



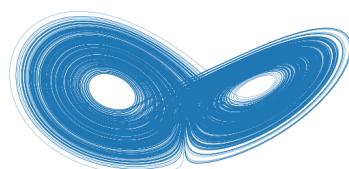
Plot 2



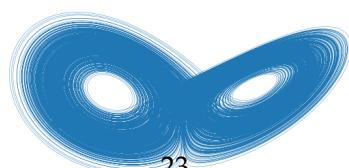
Plot 3



Plot 4



Plot 5



Maak een plot met de projecties op de coördinaatvlakken en de 3D-projectie
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")

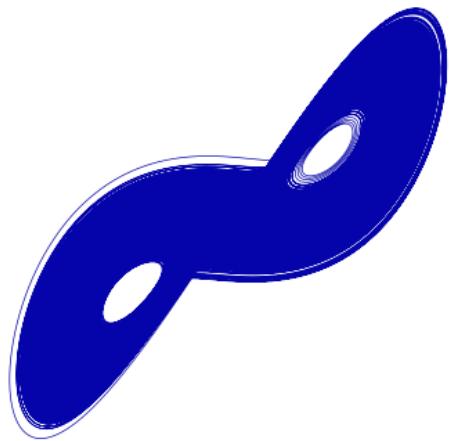
# projectie op yz-vlak
ax = fig[:add_subplot](2,2,3)
ax[:plot](Y, Z, linewidth=0.75, color="xkcd:royal blue")
ax[:set_title]("projectie op het yz-vlak")
plt.axis("off")

# 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")

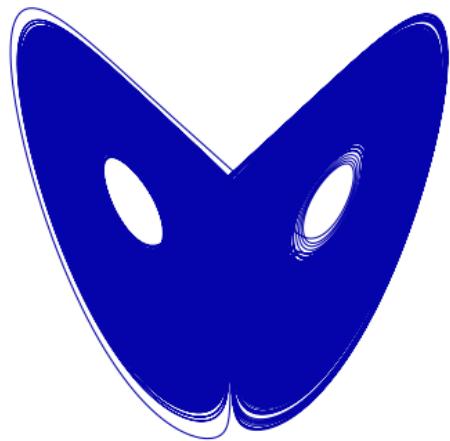
fig[:tight_layout](pad=1, w_pad=0, h_pad=0)
plt.show()

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

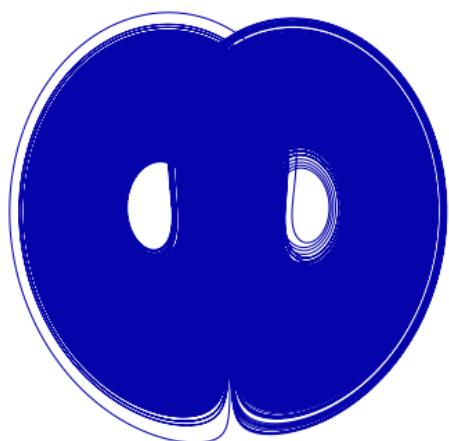
projectie op het xy-vlak



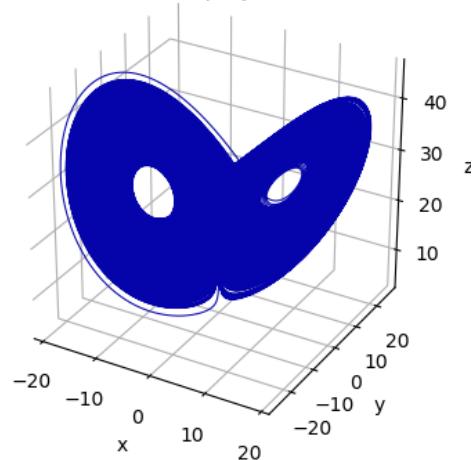
projectie op het xz-vlak



projectie op het yz-vlak



een 3d-projectie



geheugen vrijmaken

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

5.2 scripts opdracht 4

5.2.1 oef4-vind-y0.jl

De functie om vanaf command line of een ander script op te roepen om de beginwaarde voor y_0 te zoeken.

```
include("solvers/f.jl")
include("solvers/Runge-Kutta.jl")
include("oef4.jl")

function testoef4(guess_y0, rounds, h_index)
    x0 = -11.998523280062
    z0 = 27.0
    guess_y0 = guess_y0
    guess_T = [3.0, 3.1]
    rounds = rounds

    gues1 = findy0(f, x0, z0, guess_y0, guess_T, rounds, h_index)
    #print(gues1[end])

    return gues1
end
```

5.2.2 solvers/f.jl

```
function f(X::Array{Float64, 1})
    (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
```

5.2.3 solvers/Runge-Kutta.jl

```
using LinearAlgebra
```

```
"""
```

```

berekent de afstand van de rechte opgespannen door de punten beginP en endP
tot het punt point
begin , end en point moeten 1d vectoren van dezelfde dimensie zijn
returns: Float64
"""
function dist_line_point(beginP::Array{Float64, 1}, endP::Array{Float64, 1},
    point::Array{Float64, 1});
    return norm(cross(endP-beginP, beginP-point))/ norm((endP-beginP))
end

"""
berekent de orthogonale projectie van het punt point op de rechte opgespannen
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

"""
berekent de afstand van het punt point tot het lijnstuk tussen corner1 en
corner 2. Returns de afstand tot het lijnstuk , en hoe ver de projectie
op het lijnstuk ligt als fractie van de lengte , tellend vanaf corner1.
"""
function dist_point_segment(corner1::Array{Float64, 1},
                             corner2::Array{Float64, 1},
                             point::Array{Float64, 1})
    # bereken de afstand tot de 2 hoekpunten en de afstand tot de rechte
    d_corner1 = norm(corner1 - point);
    d_corner2 = norm(corner2 - point);
    d_line = dist_line_point(corner1, corner2, point);

    # check of de orthogonale projectie op het lijnstuk ligt
    projection = orth_projection(corner1, corner2, point);

```

```

# bereken de verhoudingen
fractions = (corner1 - projection) ./ (corner1 - corner2);
# test of alle verhoudingen tussen 0 en 1 liggen
on_segment = all(x -> (x >= 0) && (x <= 1), fractions);
fraction = fractions[1];
if on_segment
    return (d_line, fraction);
elseif fraction < 0
    return (d_corner1, 0.0);
else
    return (d_corner2, 1);
end
end

"""
berekent de volgende iteratiestap voor de methode van Runge-Kutta van orde
om de oplossing van de diff. vgl.  $X' = f(x)$  te benaderen met stapgrootte  $h$ 
"""

function RK4_step(f::Function, Xi::Array{Float64, 1}, h::Float64)
    k1 = f(Xi);
    k2 = f(Xi + h*k1/2);
    k3 = f(Xi + h*k2/2);
    k4 = f(Xi + h*k3);
    return Xi + h/6 * (k1 + 2*k2 + 2*k3 + k4);
end

"""
berekent de eerste  $n$  iteratiestappen van de Runge-Kutta methode van orde
en returnt het tijdstip waarop, de kleinste afstand tot de beginwaarde be-
werd in het interval  $[n1, n2]$  en deze afstand
"""

function RK4dists(f::Function, X0::Array{Float64, 1}, h::Float64, n1::Int,
                   n2::Int64)
    # voeg de eerste 2 punten toe aan de array met punten
    X1 = X0;
    X2 = RK4_step(f, X1, h);

    # bereken de eerste  $n1$  iteratiestappen
    for i = 2:n1
        X1 = X2;

```

```

X2 = RK4_step(f, X1, h);
end

# bereken voor de volgende stappen ook steeds de afstand
dist, t = dist_point_segment(X1, X2, X0)
best = (dist, h*(n1+t-1))

# itereer over de resterende stappen
for i = (n1+1):n2
    X1 = X2;
    X2 = RK4_step(f, X1, h);
    dist, t = dist_point_segment(X1, X2, X0)
    if dist < best[1]
        best = (dist, h*(i+t-1))
    end
end
return best
end

```

5.2.4 oef4.jl

```

@everywhere begin
    include("solvers/Runge-Kutta.jl")
    include("solvers/f.jl")

end

"""
initialiseerd de arrays die aanduiden in welk gebied de kleinste
afstand gezocht wordt, in termen van de index
"""

function init_N(h, guess_T, num)
    H = fill(h, num);
    n1 = floor(Int64, guess_T[1]/h); n2 = ceil(Int64, guess_T[2]/h);
    N1 = fill(n1, num); N2 = fill(n2, num);
    return (H, n1, n2, N1, N2)
end

"""
functie om beginvoorwaarden te zoeken die een periodieke oplossing geven.
Zoekt in het interval guess_y0 naar een waarde y0 zodat (x0, y0, z0) een

```

beginvoorwaarde van een periodieke oplossing is met periode T in guess_T.
Het verfijnt deze benadering rounds keer.

```
"""
function findy0(f, x0, z0, guess_y0, guess_T, rounds, h_index)
    # arrays die info bijhouden over de haalbare precisie met
    # een bepaalde stapgrootte , voor verkleinen stapgrootte indien
    # nodig .
    num_h = 4;
    delta_arr = [1e-5, 1e-7, 1e-9, 1e-11];
    h_arr = [1e-4, 1e-5, 1e-6, 1e-7];
    h = h_arr[h_index];

    # initializeren variabelen
    # vul arrays met de stapgrootte , aantal stappen en rechterlid
    # om parallel de vergelijkingen
    # op te lossen
    H, n1, n2, N1, N2 = init_N(h, guess_T, 7);
    F = fill(f, 7);

    # los de vergelijkingen op en zoek de beginvoorwaarde die de beste
    # benadering geeft
    X = [[x0, guess_y0[1] + i*(guess_y0[2]-guess_y0[1])/6, z0] for i=0:6]
    dists = pmap(RK4dists, F, X, H, N1, N2);
    # array met afstanden
    d = [dist[1] for dist in dists];
    # zoek de index van de kleinste afstand
    mini = minimum(d);
    index = findfirst(isequal(mini), d);

    # als de beste waarde een randpunt was , neem de waarde naast
    # het randpunt
    if index == 1
        index = 2
    elseif index == 7
        index = 6
    end

    # sla deze beginwaarde , afstand en periode op
    results = ["Xi" => [X[i] for i = (index-1):(index+1)], "Ei" => [d[i] for i = (index-1):(index+1)],
```

```

    "Ti" => [ dists[i][2] for i = (index-1):(index+1)];;

# test of de precisie nog niet om een kleinere stapgrootte vraagt
if mini < delta_arr[h_index]
    h_index += 1;
# we kunnen niet precieser binnen redelijke tijd , return de gevonden
# waarden
    if h_index > num_h
        return results
    end
# we kunnen precieser
    h = h_arr[h_index];
    H, n1, n2, N1, N2 = init_N(h, guess_T, 4);
end

# in de for-loop worden slechts 4 oplossingen opnieuw berekend ,
# pas de arrays voor pmap aan
H = fill(h, 4);
N1 = fill(n1, 4);
N2 = fill(n2, 4);
F = fill(f, 4);

# verbeter de waarde voor y0 rounds keer
for i=2:rounds
    # kopieer de waarden van de vorige iteratie naar hun nieuwe locaties
    X[1], X[4], X[7] = (X[index-1], X[index], X[index+1])
    dists[1], dists[4], dists[7] = (dists[index-1], dists[index], dists[index+1])
    # bereken de nieuwe X-waarden
    X[2], X[3] = ((2*X[1] + X[4])/3, (X[1] + 2*X[4])/3)
    X[5], X[6] = ((2*X[4] + X[7])/3, (X[4] + 2*X[7])/3)
    # bereken de nieuwe afstanden en periodes
    dists[2], dists[3], dists[5], dists[6] = pmap(RK4dists,
        F, [X[2], X[3], X[5], X[6]], H, N1, N2)

    # zoek de beste beginvoorwaarde en sla deze op, samen met de
    # afstand en periode
    d = [dist[1] for dist in dists];
    # zoek de index van de kleinste afstand
    mini = minimum(d);

```

```

index = findfirst(isequal(mini), d);

# als de beste waarde een randpunt was, neem de waarde naast
# het randpunt
if index == 1
    index = 2
elseif index == 7
    index = 6
end
# sla deze beginwaarde, afstand en periode op
push!(results , "Xi" => [X[i] for i = (index-1):(index+1)],
      "Ei" => [d[i] for i = (index-1):(index+1)],
      "Ti" => [dists[i][2] for i = (index-1):(index+1)]]

# test of de precisie nog niet om een kleinere stapgrootte vraagt
if mini < delta_arr[h_index]
    h_index += 1;
    # we kunnen niet precieser binnen redelijke tijd, return de g
    # waarden
    if h_index > num_h
        return results
    end
    # we kunnen precieser
    h = h_arr[h_index];
    H, n1, n2, N1, N2 = init_N(h, guess_T, 4);
end
end
return results
end

```