A2

May 9, 2024

1 Computational Physics Blatt 04

Anne, Fabian und Asliddin

[ ]:
```
import numpy as np
from numpy.linalg import norm
from functools import partial
import matplotlib.pyplot as plt
import matplotlib
from scipy.optimize import curve_fit
from scipy.stats import linregress

matplotlib.rcParams["figure.figsize"] = [12, 8]
%matplotlib inline

T_MAX = 100 # Increased for nicer results
H = 0.001 # decreased for nicer results
SIGMA = 10
B = 8/3
R = 28
```

1.1 a)

Implementation of Runge-Kutta 5

[ ]:
```
def RKDP(f, y0, t0, tmax, h, digits=None, **kwargs):
    dim = len(y0)
    coef = {  # Coefficients for the Dormand-Prince method, padded with zeros␣
 ↪for the 6th order
        "k2": np.array([1 / 5, 0, 0, 0, 0, 0]).reshape(-1, 1),
        "k3": np.array([3 / 40, 9 / 40, 0, 0, 0, 0]).reshape(-1, 1),
        "k4": np.array([44 / 45, -56 / 15, 32 / 9, 0, 0, 0]).reshape(-1, 1),
        "k5": np.array(
            [19372 / 6561, -25360 / 2187, 64448 / 6561, -212 / 729, 0, 0]
        ).reshape(-1, 1),
        "k6": np.array(
            [9017 / 3168, -355 / 33, 46732 / 5247, 49 / 176, -5103 / 18656, 0]
        ).reshape(
```

```python
            -1, 1
        ),  # called a6 in the literature
        "y": np.array(
            [35 / 384, 0, 500 / 1113, 125 / 192, -2187 / 6784, 11 / 84]
        ).reshape(
            -1, 1
        ),  # called b in the literature
        "t": np.array(
            [0, 1 / 5, 3 / 10, 4 / 5, 8 / 9, 1]
        ),  # called c in the literature
    }

    t = t0
    y = y0
    fp = partial(f, **kwargs)  # set sigma, r and b
    while t < tmax:
        ks = np.zeros((6, dim))
        ks[0] = h * fp(t, y)
        ks[1] = h * fp(t + h * coef["t"][1], y + np.sum(ks * coef["k2"]))
        ks[2] = h * fp(t + h * coef["t"][2], y + np.sum(ks * coef["k3"]))
        ks[3] = h * fp(t + h * coef["t"][3], y + np.sum(ks * coef["k4"]))
        ks[4] = h * fp(t + h * coef["t"][4], y + np.sum(ks * coef["k5"]))
        ks[5] = h * fp(t + h * coef["t"][5], y + np.sum(ks * coef["k6"]))
        y += np.sum(coef["y"] * ks, axis=0)
        if digits:
            y = np.round(y, digits)
        t += h
        yield t, *y
```

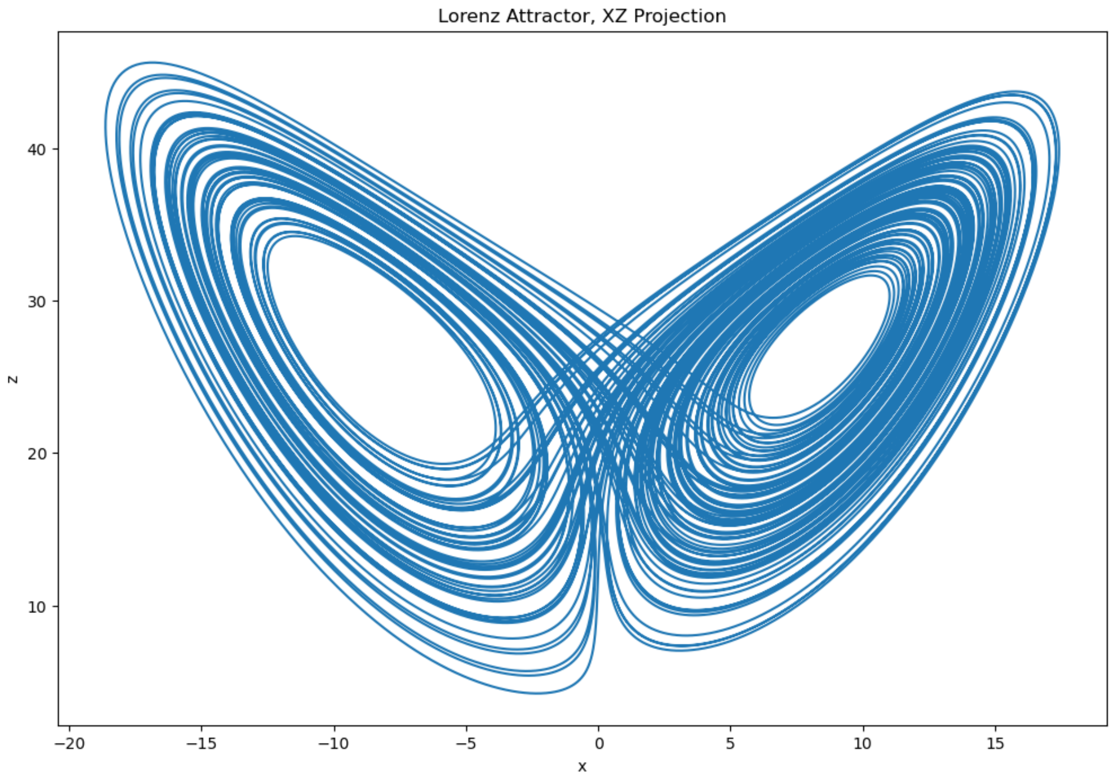
1.2 b)

Solve the Lorenz equation with $y_0 = (2, 3, 14)$, the parameters $\sigma = 10$, $b = \frac{3}{5}$ and $r = 28$ in the time frame $t_0 = 0$ to $t_e = 50$ with a stepsize of $h = 0.01$. Plot the XZ-projection.

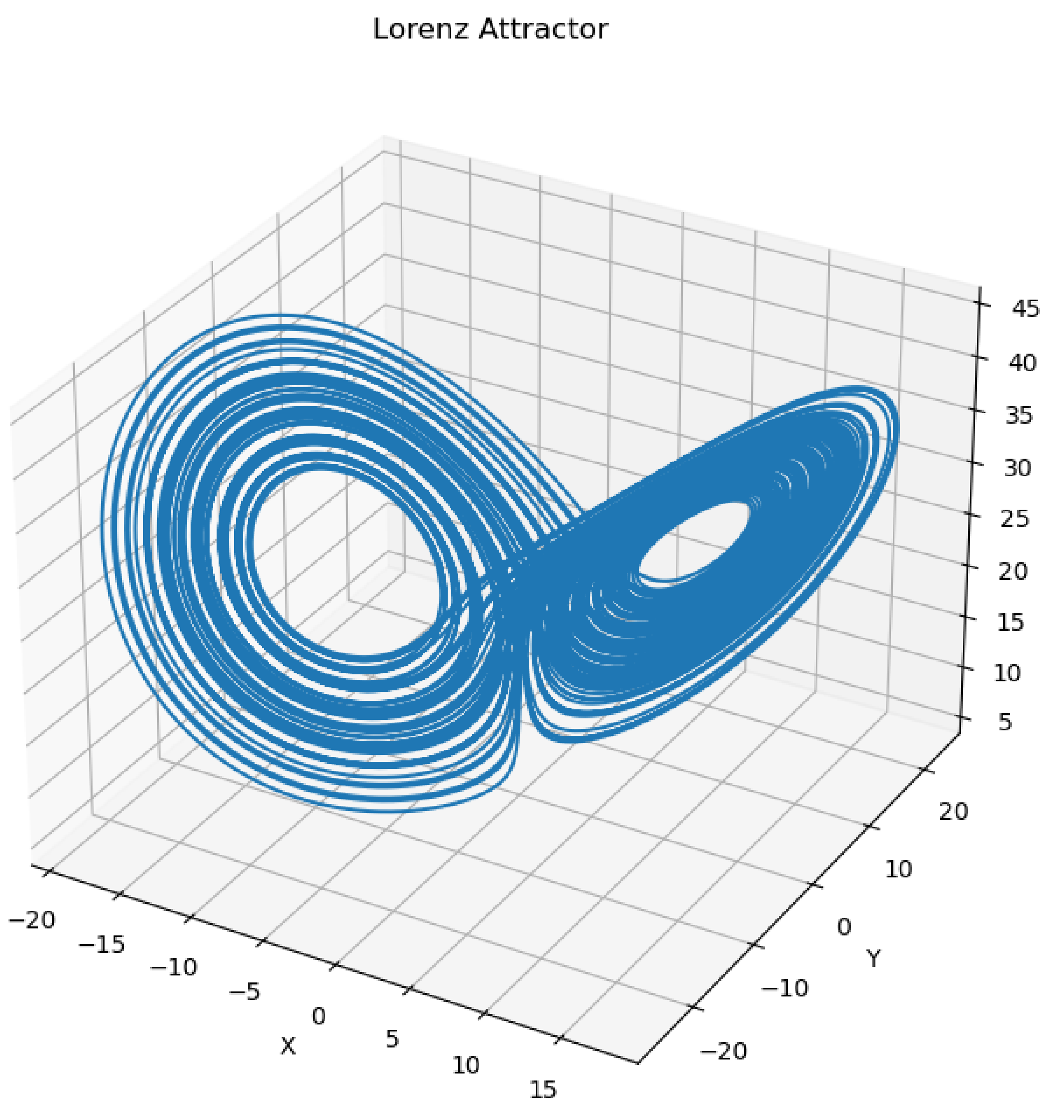
[ ]:
```python
def lorentz(t, y, sigma, r, b):
    x_dot = sigma * (y[1] - y[0])
    y_dot = y[0] * (r - y[2]) - y[1]
    z_dot = y[0] * y[1] - b * y[2]
    return np.array([x_dot, y_dot, z_dot])
```

[ ]:
```python
y0 = np.array([2, 3, 14], dtype=float)  # prevent type-casting errors
rkdp = RKDP(lorentz, y0, t0=0, tmax=T_MAX, h=H, sigma=SIGMA, r=R, b=B)
system = np.array(list(rkdp))
time = system[:, 0]
trajectory = system[:, 1:]
```

[ ]:
```python
plt.plot(trajectory[:, 0], trajectory[:, 2])
plt.xlabel("x")
plt.ylabel("z")
plt.title("Lorenz Attractor, XZ Projection")
None  # prevent ugly output
```



[ ]:
```python
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, projection="3d")

ax.plot(trajectory[:, 0], trajectory[:, 1], trajectory[:, 2])
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")  # strangly missing
plt.title("Lorenz Attractor")
None  # prevent ugly output
```



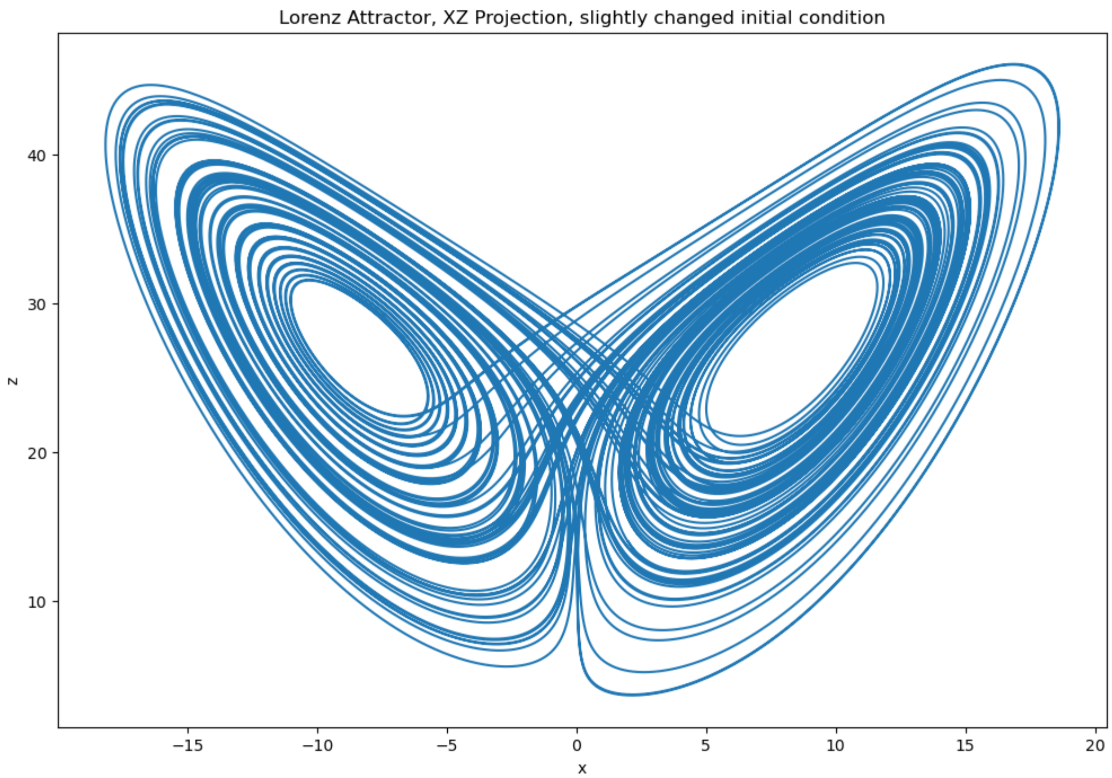
1.3 c)

Check the chaotic nature by using $y_0 = (2, 3, 14 + 10^{-9})$ and compare the trajectories.

[ ]:
```
y0_eps = np.array([2, 3, 14 + 1e-9], dtype=np.float64)
rkdp2 = RKDP(lorentz, y0_eps, t0=0, tmax=T_MAX, h=H, sigma=SIGMA, r=R, b=B)
system2 = np.array(list(rkdp2))
time2 = system2[:, 0]
trajectory2 = system2[:, 1:]
```

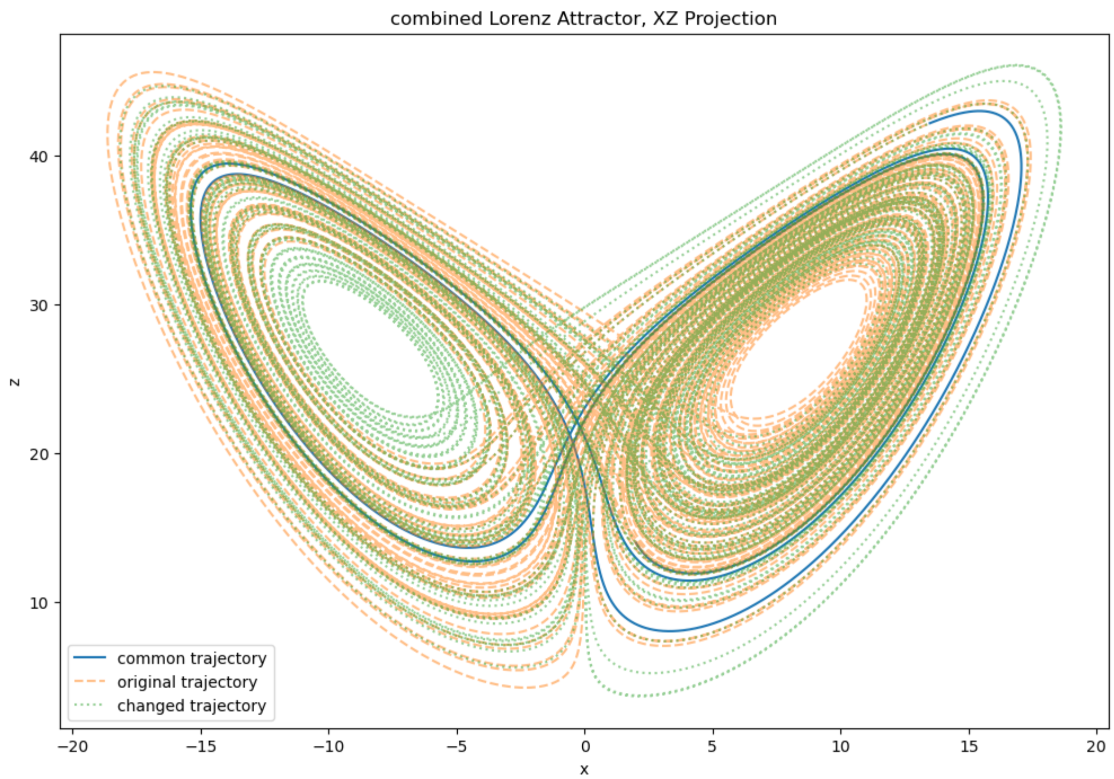
[ ]:
```
plt.plot(trajectory2[:, 0], trajectory2[:, 2])
plt.xlabel("x")
```

```
plt.ylabel("z")
plt.title("Lorenz Attractor, XZ Projection, slightly changed initial condition")
None  # prevent ugly output
```



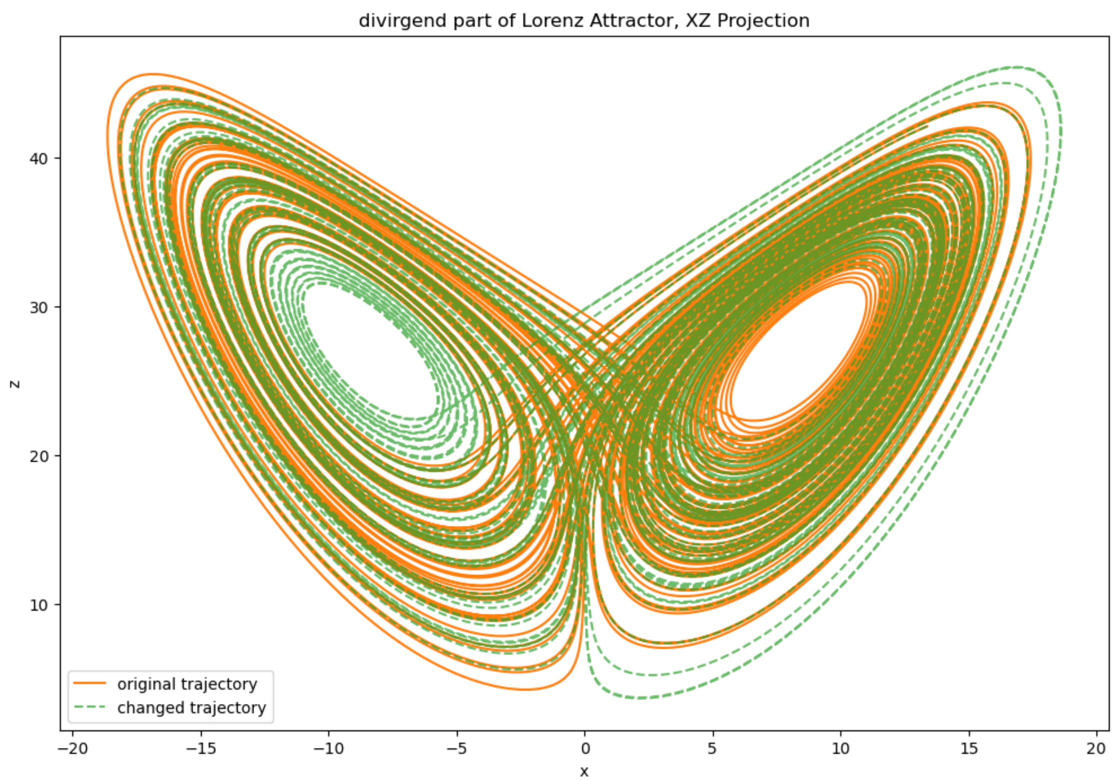
[ ]:
```
traj_mask = np.all(np.isclose(trajectory, trajectory2, rtol=1e-7), axis=1)
div_point = np.argmin(traj_mask)
plt.plot(
    trajectory[:div_point, 0],
    trajectory[:div_point, 2],
    ls="solid",
    label="common trajectory",
)
plt.plot(
    trajectory[div_point:, 0],
    trajectory[div_point:, 2],
    ls="dashed",
    alpha=0.5,
    label="original trajectory",
)
plt.plot(
    trajectory2[div_point:, 0],
    trajectory2[div_point:, 2],
```

```
    ls="dotted",
    alpha=0.5,
    label="changed trajectory",
)
plt.legend()
plt.xlabel("x")
plt.ylabel("z")
plt.title("combined Lorenz Attractor, XZ Projection")
None  # prevent ugly output
```



[ ]:
```
plt.plot(
    trajectory[div_point:, 0],
    trajectory[div_point:, 2],
    ls="solid",
    c="tab:orange",
    label="original trajectory",
)
plt.plot(
    trajectory2[div_point:, 0],
    trajectory2[div_point:, 2],
    ls="dashed",
    c="tab:green",
```

```python
    alpha=0.7,
    label="changed trajectory",
)
plt.legend()
plt.xlabel("x")
plt.ylabel("z")
plt.title("divirgend part of Lorenz Attractor, XZ Projection")
None  # prevent ugly output
```



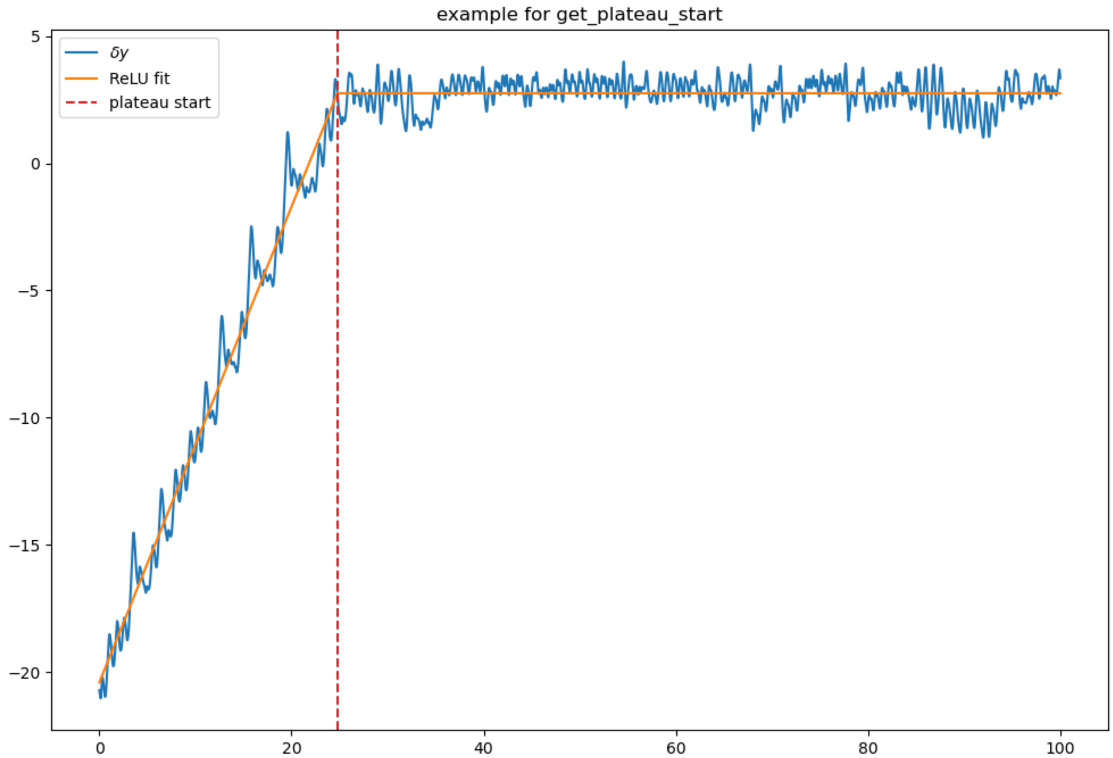
1.4 d)

Calculate the Liapunov exponent. Plot the result and create a fit to an exponential function.

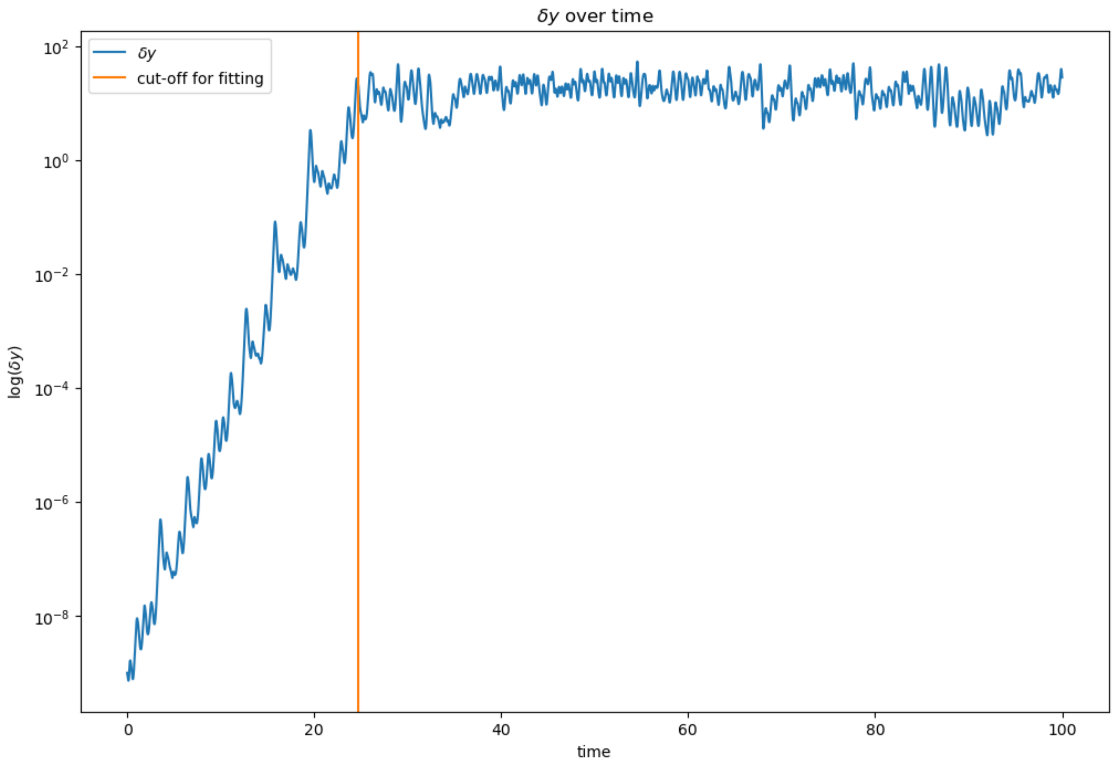
[ ]:
```python
delta_y = np.linalg.norm(trajectory2 - trajectory, axis=1)


def liapunov(t, lambda_, a):
    return np.exp(lambda_ * t) * a
```

[ ]:
```python

```

```python
# automation for the search of the plateau start by fitting a function to the␣
↪data. This only works if the data has the increas-plateau form. If not␣
↪return NaN. As the min and max value are already good estimations for b and␣
↪c, this fit should converge rather quickly.


def relu(x, a, b, c):
    return np.minimum(a * x + b, c)


def get_plateau_start(
    delta_y, time
):  # This was way more work than doing part f) by hand
    """Fitts a relu function to the data and returns the time at which the␣
↪function plateaus."""
    logY = np.log(delta_y)
    try:
        params, cov = curve_fit(
            relu,
            time[delta_y > 0],
            logY[delta_y > 0],
            p0=[1, np.min(logY), np.max(logY)],
        )
        t_cutoff = (params[2] - params[1]) / params[0]  # x = (c-b)/a
    except RuntimeError:  # if the fit fails, return nan
        t_cutoff = np.nan
    return t_cutoff


# example plot of what we are doing
logY = np.log(delta_y)
params, cov = curve_fit(relu, time, logY, p0=[1, np.min(logY), np.max(logY)])
plt.plot(time, np.log(delta_y), label="$\delta y$")
plt.plot(time, relu(time, *params), label="ReLU fit")
plt.axvline(
    get_plateau_start(delta_y, time), c="tab:red", ls="--", label="plateau␣
↪start"
)
plt.legend()
plt.title("example for get_plateau_start")
None  # prevent ugly output
```



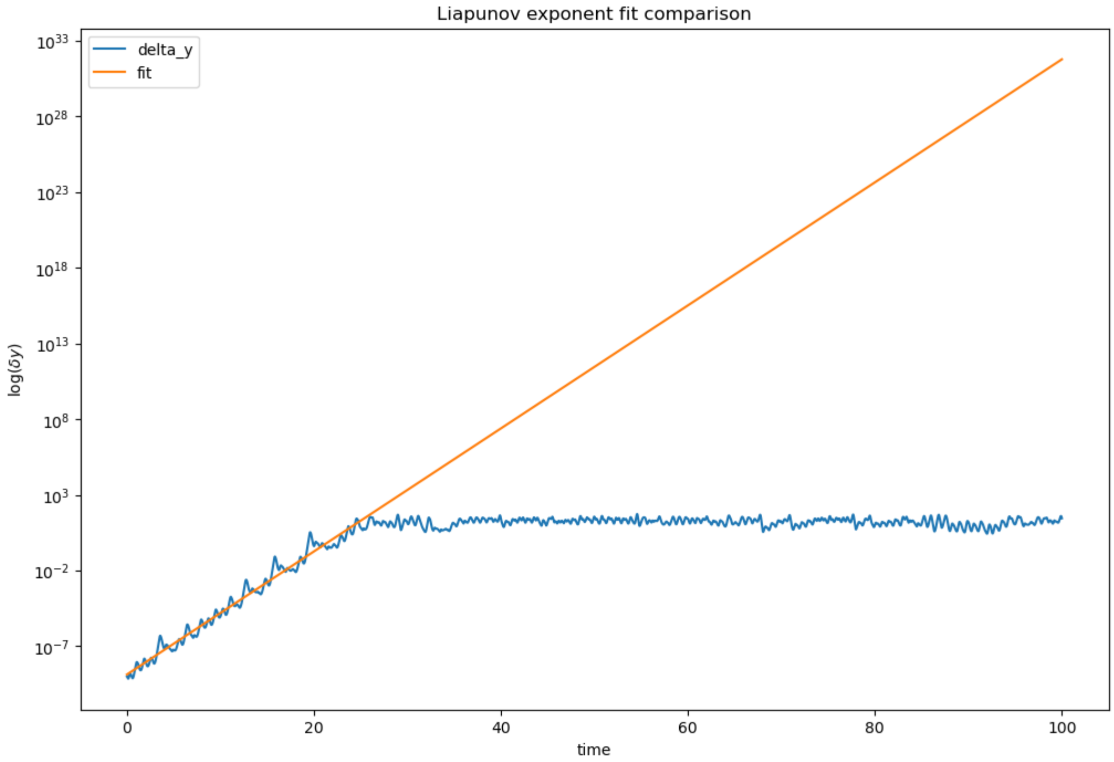
[ ]:
```
delta_y = np.linalg.norm(trajectory2 - trajectory, axis=1)
t_cutoff = get_plateau_start(delta_y, time)

plt.plot(time, delta_y, label="$\delta y$")
plt.yscale("log")
plt.ylabel(r"log($\delta y$)")
plt.axvline(x=t_cutoff, c="tab:orange", label="cut-off for fitting")
plt.xlabel("time")
plt.title("$\delta y$ over time")
plt.legend()
None  # prevent ugly output
```



[ ]:
```
linFit = linregress(
    time[time < t_cutoff], np.log(delta_y[time < t_cutoff])
)  # if a linear regression is possible, it performs better than an ordinary
 ↪minimizer
print(f"Lyapunov exponent: {linFit.slope}")
print(f"a value: {np.exp(linFit.intercept)}")
```

```
Lyapunov exponent: 0.9357312391505839
a value: 1.3669578951590576e-09
```

[ ]:
```
plt.plot(time, delta_y, label="delta_y")
plt.plot(time, liapunov(time, linFit.slope, np.exp(linFit.intercept)),
 ↪label="fit")
plt.yscale("log")
plt.ylabel(r"log($\delta y$)")
plt.xlabel("time")
plt.legend()
plt.title("Liapunov exponent fit comparison")
None  # prevent ugly output
```



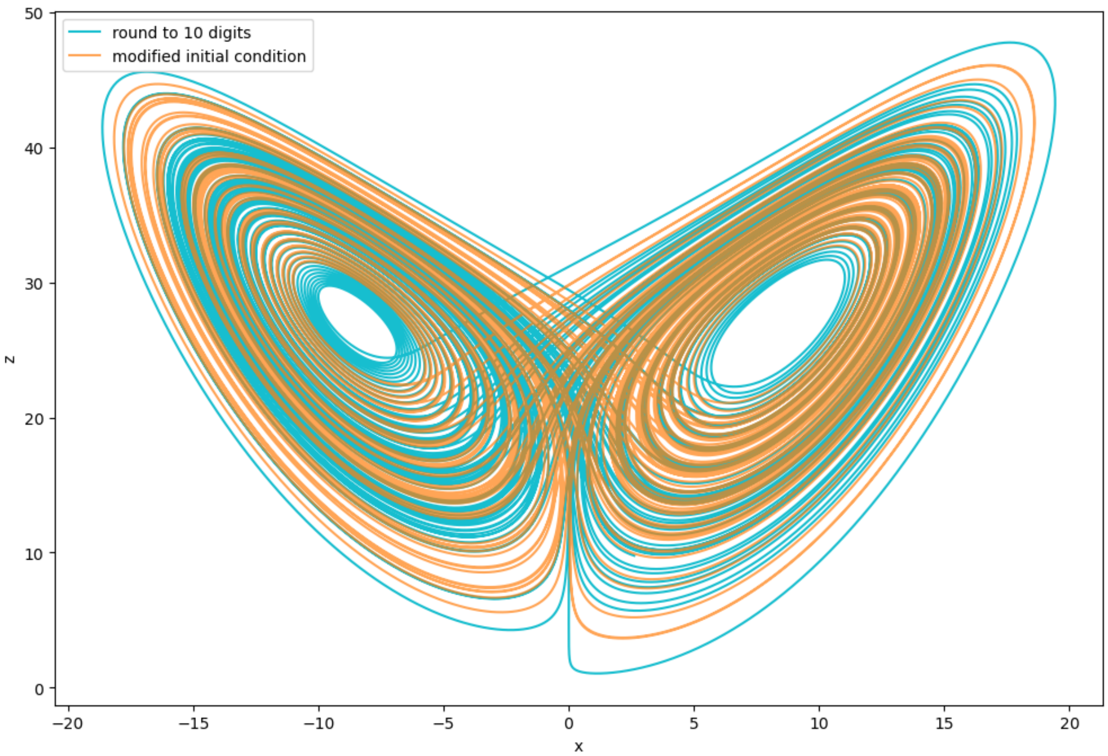
1.5 e)

Create errors by rounding the values to 10 digits. Compare them to the trajectory from b).

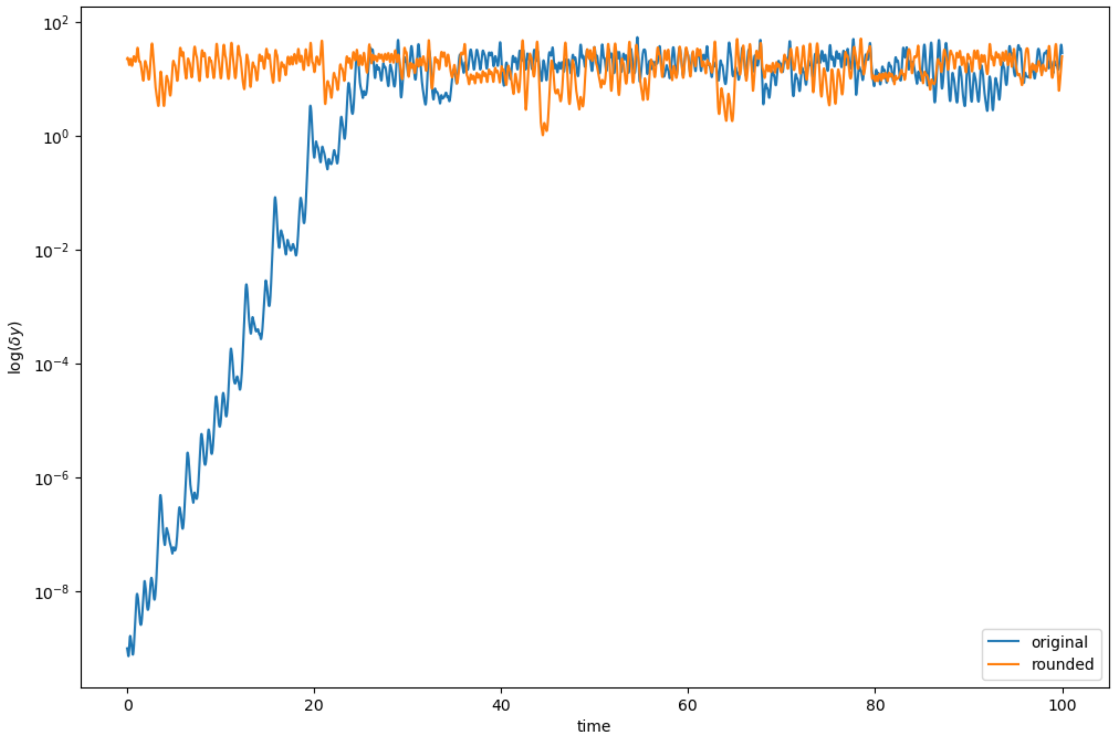
[ ]:
```
rkdp3 = RKDP(
    lorentz,
    y0,
    t0=0,
    tmax=T_MAX,
    h=H,
    digits=10,
    sigma=SIGMA,
    r=R,
    b=B,
)
system3 = np.array(list(rkdp3))
time3 = system3[:, 0]
trajectory3 = system3[:, 1:]
```

[ ]:
```
plt.plot(trajectory3[:, 0], trajectory3[:, 2], c="tab:cyan", label="round to 10␣
↪digits")
plt.plot(
    trajectory2[:, 0],
```

```
    trajectory2[:, 2],
    c="tab:orange",
    label="modified initial condition",
    alpha=0.7,
)
plt.legend()
plt.xlabel("x")
plt.ylabel("z")
None  # prevent ugly output
```



[ ]:
```
delta_y3 = np.linalg.norm(trajectory3 - trajectory, axis=1)

plt.plot(time, delta_y, label="original")
plt.plot(time, delta_y3, label="rounded")
plt.yscale("log")
plt.ylabel(r"log($\delta y$)")
plt.xlabel("time")
plt.legend()
None  # prevent ugly output
```



When introducing errors due to rounding, δy stays in the same (big) order of magnitude. The "natural" divergence has an exponential increasing error instead.

1.6 f)

Lower r to find a r_{crit} where the system is no longer chaotic.

[ ]:
```
def liapFromLorePara(r, sigma, b, t0, tmax, h, digits=None):
    """calculates the liapunov exponent of the lorenz system for given␣
 ↪parameters."""
    y0 = np.array([2, 3, 14], dtype=float)  # prevent type-casting errors
    rkdp0 = RKDP(lorentz, y0, t0, tmax, h, digits=digits, sigma=sigma, r=r, b=b)
    system0 = np.array(list(rkdp0))
    time0 = system0[:, 0]
    trajectory0 = system0[:, 1:]

    y1 = np.array([2, 3, 14 + 1e-9], dtype=float)  # prevent type-casting errors
    rkdp1 = RKDP(lorentz, y1, t0, tmax, h, digits=digits, sigma=sigma, r=r, b=b)
    system1 = np.array(list(rkdp1))
    time1 = system1[:, 0]
    trajectory1 = system1[:, 1:]

    delta_y = np.linalg.norm(trajectory0 - trajectory1, axis=1)
    t_cutoff = get_plateau_start(delta_y, time0)
```

```python
    if np.isnan(t_cutoff):
        return np.NaN
    if len(time0[time0 < t_cutoff]) < 2 or len(time0[time0 < t_cutoff]) == len(
        time0
    ):  # check if plateau start is in the time frame
        return 0
    linFit = linregress(time0[time0 < t_cutoff], np.log(delta_y[time0 <
 ↪t_cutoff]))
    return linFit.slope
```

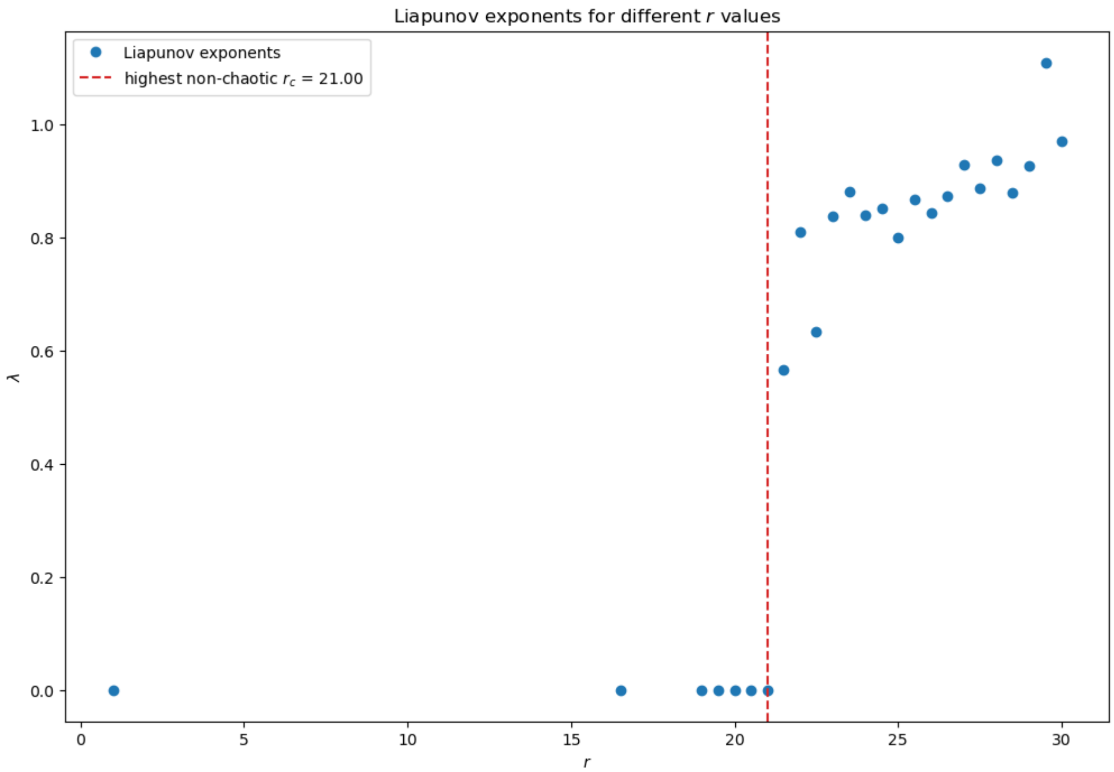
[ ]:
```python
from multiprocessing import Pool, cpu_count


rs = np.linspace(1, 30, 59)

liapF = partial(liapFromLorePara, sigma=SIGMA, b=B, t0=0, tmax=T_MAX, h=H)


with Pool(cpu_count() - 2) as p:  # use all but 2 cores
    liap = np.array(p.map(liapF, rs))
```

```
/tmp/ipykernel_16137/1903625063.py:12: RuntimeWarning: divide by zero
encountered in log
  logY = np.log(delta_y)
/tmp/ipykernel_16137/1903625063.py:12: RuntimeWarning: divide by zero
encountered in log
  logY = np.log(delta_y)
/tmp/ipykernel_16137/1903625063.py:12: RuntimeWarning: divide by zero
encountered in log
  logY = np.log(delta_y)
/tmp/ipykernel_16137/1903625063.py:12: RuntimeWarning: divide by zero
encountered in log
  logY = np.log(delta_y)
/tmp/ipykernel_16137/1903625063.py:12: RuntimeWarning: divide by zero
encountered in log
  logY = np.log(delta_y)
/tmp/ipykernel_16137/1903625063.py:14: OptimizeWarning: Covariance of the
parameters could not be estimated
  params, cov = curve_fit(
/tmp/ipykernel_16137/1903625063.py:12: RuntimeWarning: divide by zero
encountered in log
  logY = np.log(delta_y)
/tmp/ipykernel_16137/1903625063.py:12: RuntimeWarning: divide by zero
encountered in log
  logY = np.log(delta_y)
/tmp/ipykernel_16137/1903625063.py:12: RuntimeWarning: divide by zero
encountered in log
  logY = np.log(delta_y)
```

```
/tmp/ipykernel_16137/1903625063.py:12: RuntimeWarning: divide by zero
encountered in log
  logY = np.log(delta_y)
/tmp/ipykernel_16137/1903625063.py:12: RuntimeWarning: divide by zero
encountered in log
  logY = np.log(delta_y)
/tmp/ipykernel_16137/1903625063.py:14: OptimizeWarning: Covariance of the
parameters could not be estimated
  params, cov = curve_fit(
/tmp/ipykernel_16137/1903625063.py:14: OptimizeWarning: Covariance of the
parameters could not be estimated
  params, cov = curve_fit(
/tmp/ipykernel_16137/1903625063.py:14: OptimizeWarning: Covariance of the
parameters could not be estimated
  params, cov = curve_fit(
```

[ ]:

```
r_crit = rs[liap == 0][-1]
plt.plot(rs, liap, marker="o", ls="none", label="Liapunov exponents")
plt.axvline(
    r_crit, c="tab:red", ls="--", label=f"highest non-chaotic $r_c$ = {r_crit:.
 ↪2f}"
)
plt.xlabel("$r$")
plt.ylabel("$\lambda$")
plt.legend()
plt.title("Liapunov exponents for different $r$ values")
None  # prevent ugly output
```



1.7 g)

Implement the dynamic step-size using a safety factor of $S = 0.95$. Compare the compute time to the fixed approach.

[ ]:
```
def RKF(f, y0, t0, tmax, h0, atol, rtol, hmax, S=0.95, alpha=0.2, **kwargs):
    dim = len(y0)
    coef = {  # Coefficients for the Dormand-Prince method, padded with zeros␣
 ↪for the 6th order
        "k2": np.array([1 / 5, 0, 0, 0, 0, 0]).reshape(-1, 1),
        "k3": np.array([3 / 40, 9 / 40, 0, 0, 0, 0]).reshape(-1, 1),
        "k4": np.array([44 / 45, -56 / 15, 32 / 9, 0, 0, 0]).reshape(-1, 1),
        "k5": np.array(
            [19372 / 6561, -25360 / 2187, 64448 / 6561, -212 / 729, 0, 0]
        ).reshape(-1, 1),
        "k6": np.array(
            [9017 / 3168, -355 / 33, 46732 / 5247, 49 / 176, -5103 / 18656, 0]
        ).reshape(
            -1, 1
        ),  # called a6 in the literature
        "y5": np.array(
            [35 / 384, 0, 500 / 1113, 125 / 192, -2187 / 6784, 11 / 84]
```

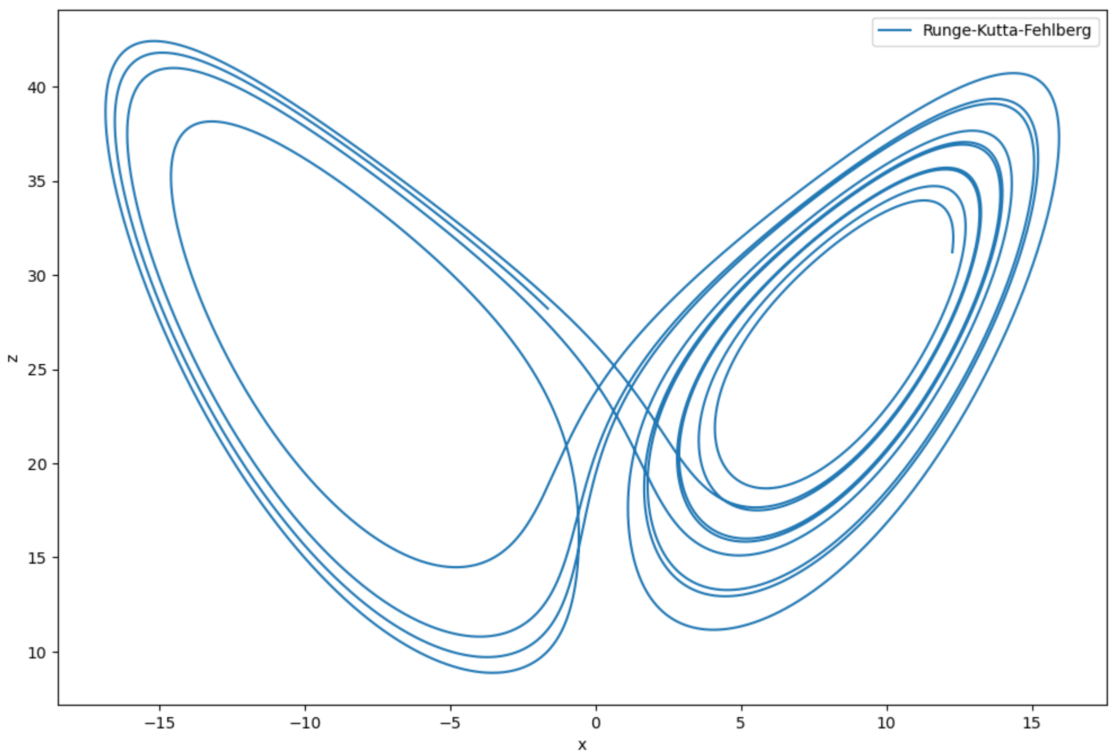
```
        ).reshape(
            -1, 1
        ),  # called b in the literature
        "y4": np.array(
            [5179 / 57600, 0, 7571 / 16695, 393 / 640, -92097 / 339200, 187 /
 ↪2100]
        ).reshape(
            -1, 1
        ),  # called b* in the literature
        "t": np.array(
            [0, 1 / 5, 3 / 10, 4 / 5, 8 / 9, 1]
        ),  # called c in the literature
    }

    t = t0
    y = y0
    h = h0
    fp = partial(f, **kwargs)
    while t < tmax:
        ks = np.zeros((6, dim))
        ks[0] = h * fp(t, y)
        ks[1] = h * fp(t + h * coef["t"][1], y + np.sum(ks * coef["k2"]))
        ks[2] = h * fp(t + h * coef["t"][2], y + np.sum(ks * coef["k3"]))
        ks[3] = h * fp(t + h * coef["t"][3], y + np.sum(ks * coef["k4"]))
        ks[4] = h * fp(t + h * coef["t"][4], y + np.sum(ks * coef["k5"]))
        ks[5] = h * fp(t + h * coef["t"][5], y + np.sum(ks * coef["k6"]))
        y5 = np.sum(coef["y5"] * ks, axis=0)
        y4 = np.sum(coef["y4"] * ks, axis=0) + y5 / 40
        scale = atol + rtol * np.max([norm(y), norm(y5)], axis=0)
        err = np.sqrt(np.mean(((y5 - y4) / scale) ** 2))

        if err < 1:
            t += h
            h = np.min(
                [S * h * (1 / err) ** alpha, hmax]
            )  # here we check for the maximum step size
            y += y5
            yield t, *y
        else:
            h = S * h * (1 / err) ** alpha
```

[ ]:

```
rkf = RKF(
    lorentz,
    y0,
    t0=0,
    tmax=10,
    h0=0.01,
```

```
    hmax=2,
    atol=1e-8,  # 1e-12 takes way too long
    rtol=1e-8,  # took around 10 minutes
    sigma=SIGMA,
    r=R,
    b=B,
)
system_rkf = np.array(list(rkf))
time_rkf = system_rkf[:, 0]
trajectory_rkf = system_rkf[:, 1:]
```

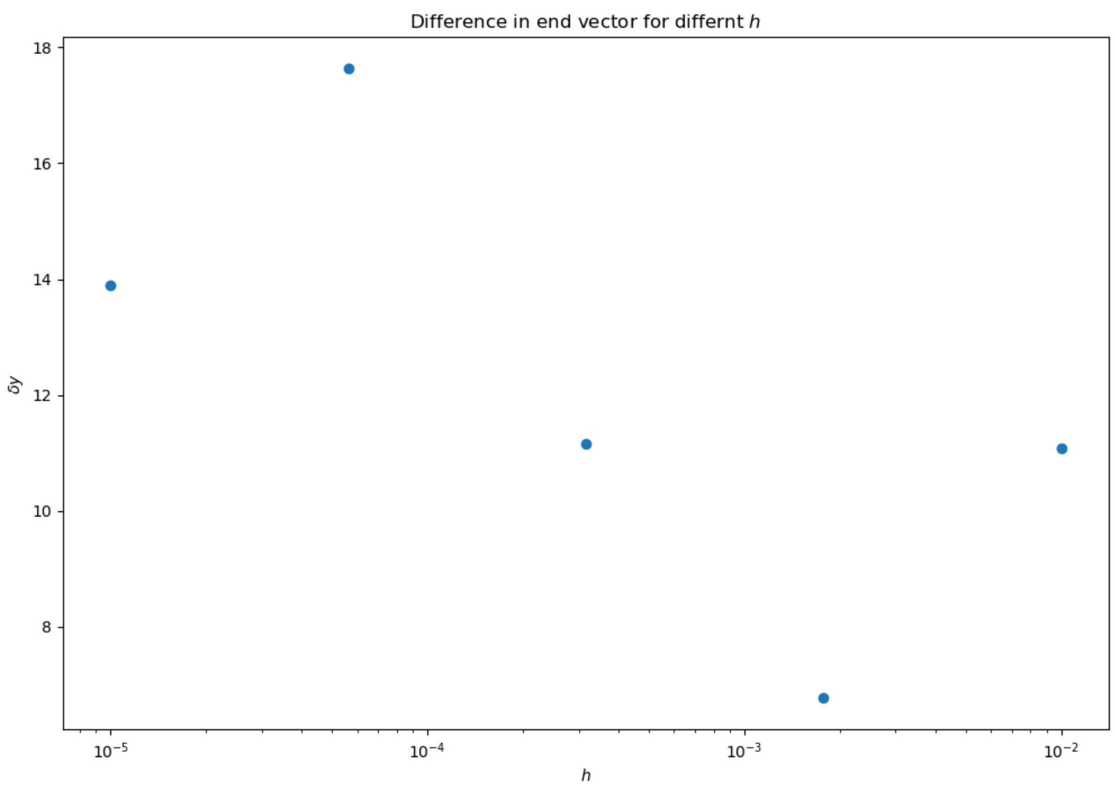
[ ]:
```
plt.plot(trajectory_rkf[:, 0], trajectory_rkf[:, 2],␣
 ↪label="Runge-Kutta-Fehlberg")
plt.legend()
plt.xlabel("x")
plt.ylabel("z")
None  # prevent ugly output
```



[ ]:
```
hs = np.logspace(-5, -2, 5)


def hToDeltaY(h):
```

```
    rkdp_ = RKDP(lorentz, y0, t0=0, tmax=10, h=h, sigma=SIGMA, r=R, b=B)
    system_ = np.array(list(rkdp_))
    time_temp = system_[:, 0]
    trajectory_ = system_[:, 1:]
    return (norm(trajectory_[-1] - trajectory_rkf[-1]), h)


with Pool(cpu_count() - 2) as p:  # use all but 2 cores
    pool_result = np.array(p.map(hToDeltaY, hs))
```

[ ]:
```
plt.scatter(pool_result[:, 1], pool_result[:, 0])
plt.xscale("log")
plt.xlabel("$h$")
plt.ylabel("$\delta y$")
plt.title("Difference in end vector for differnt $h$")
None  # prevent ugly output
```



We are not even close to $\delta y(t_e) < 10^{-6}$. But reducing h any more leeds to unreasonable computation time. We might have an error in this implementation.

2 A1

2.1 a)

ODE integration methods with a dynamik stepsize have the advantage, that generally they run as fast as possible. Due to the overhead it is possible that a given equation computes slower than a fixed stepsize, but this should rarely be the case.

2.2 b)

The stepsize estimator is based on the different error estimator of the runge kutta methods. In the rk4 method we have $err = O(h^{-5}) - O(h^{-4})$. So for a given error we can calculate the optimal h