Aufgabe 2.1

(a) a Der Massenzufluss an Salz in das Gefäß pro Zeiteinheit sei definiert als $m_{in} = rc_0$. Weiterhin sei der Massenabfluß an Salz definiert als $m_{out} = rc$, mit $c = c(t)$. Die Menge an Salz im Gefäß definiert als $m = Vc$, mit $m = m(t)$, woraus sich eine zeitliche Änderung der Salzmasse im Gefäß von $\dot{m} = V\dot{c}$ ergibt. Diese lässt sich außerdem darstellen als $m \dot{=} rc_0 - rc$.
Kombiniert man diese Gleichungen, so erhält man:

$$\dot{m} = \dot{m} \quad (1)$$
$$\leftrightarrow \quad V\dot{c} = rc_0 - rc \quad (2)$$
$$\leftrightarrow \quad \dot{c} = \frac{rc_0}{V} - \frac{rc}{V} \quad (3)$$

(b) b

Aufgabe 2.2

Lösungscode ist in der Angefügten .ipynb Datei und im Appendix des PDFs angegeben. Ein direkter Link befindet sich im jeweiligen Aufgabenteil. Die berechneten Löesungspunkte wurden mittels Jupyter Notebook bestimmt (Ergebnisse zu finden üeber dem Plot).

(a) Lösungscode expliziter Euler
Berechnete Lösungspunkte für $t = 0, 1, 2$: 1.0, 1.33333333, 1.63665777

(b) Lösungscode impliziter Euler
Berechnete Lösungspunkte für $t = 0, 1, 2$: 1.0, 1.28831999, 2.35160323

(c) Lösungscode Heun
Berechnete Lösungspunkte für $t = 0, 1, 2$: 1.0, 1.31832888, 1.87664008

(d) Lösungscode Runge-Kutta
Berechnete Lösungspunkte für $t = 0, 1, 2$: 1.0, 0.58699063, 1.60222733

Figure 1 zeigt das Ergebnis der Berechnungen als Plot dargestellt.

Aufgabe 2.3

(a) a und b sind Üebergangsvariablen, die die Änderungrate der Susceptibles zu Infected (a), bzw. die Änderungrate von Infected zu Recovered (b) pro Zeiteinheit modellieren. Hierbei stehen a und b allerdings nicht nur für eine 'simple' Infektions-, bzw. Genesungsrate, sondern schließen alle möglichen Faktoren ein, die zum Infektions-/Genesungsgeschehen beitragen.

(b) Der komplette Code zur Lösung ist im beigefügten Jupyter Notebook einsehbar. Das Modell selbst war definiert durch den unten stehenden Code. Die Zeitabhängigkeit der einzelnen Variablen wird implizit über ihre Position in der Lösungsliste miteinbezogen. $V(t)$ und c wurden aus Einfachheit bereits nonfunktional miteinbezogen. Als Einschrittverfahren wurde das explizite Euler-Verfahren gewählt.

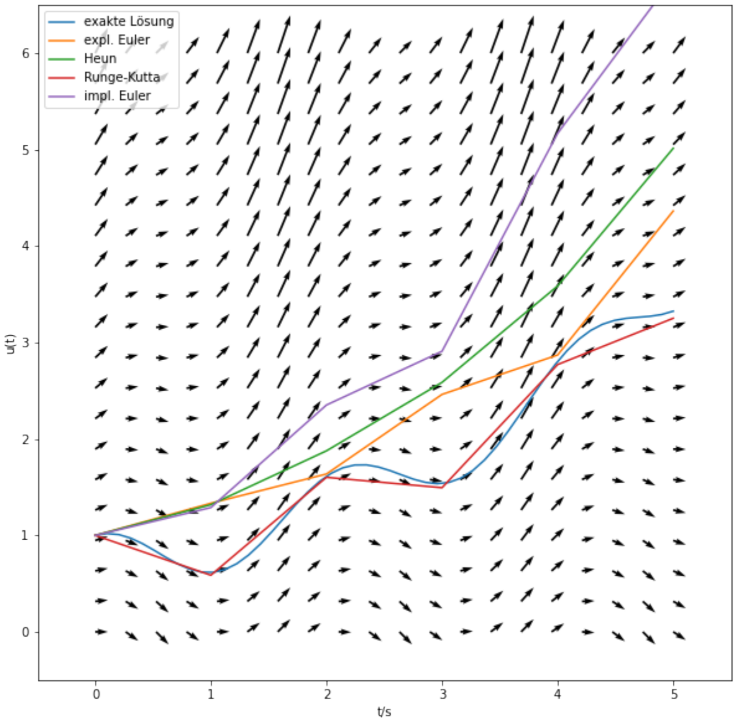

Figure 1: Ergebnisse aller Einschrittverfahren geplottet gegen die exakte Lösung

```
def SUS(SIRV, a=0.5, b=0.1):
    return -a*SIRV[0]*SIRV[1]

def INF(SIRV, a=0.5, b=0.1):
    return a*SIRV[0]*SIRV[1] - b*SIRV[1]

def REC(SIRV, a=0.5, b=0.1):
    return b*SIRV[1]

def VAC(SIRV, a=0.5, b=0.1):
    return 0
```

Folgend das Ergebnis der Berechnung in Figure 2:

(c)

$$\frac{dS}{dt} = -aS(t)I(t) - cS(t) \tag{4}$$
$$\frac{dI}{dt} = aS(t)I(t) - bI(t) - cI(t) \tag{5}$$
$$\frac{dR}{dt} = bI(t) - cR(t) \tag{6}$$
$$\frac{dV}{dt} = cS(t) + cI(t) + cR(t) \tag{7}$$

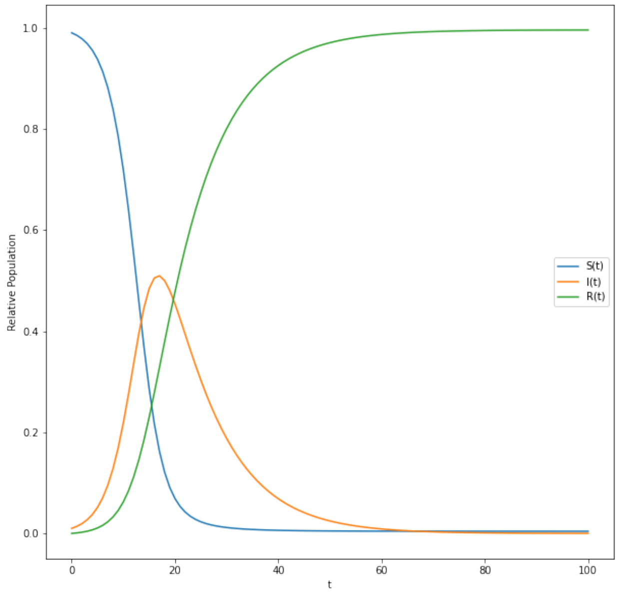

Figure 2: Lösung des SIR-Modells mittels explizitem Euler-Verfahren

(d) Das erweitere Modell wurde mit dem analogen Code simuliert der auch in (b) verwendet wurde. Allerdings wurden die Funktionen für die einzelnen Gruppen entsprechend (c) angepasst. Der Code ist im Folgenden zu sehen.

```
def SUS(SIRV, a=0.5, b=0.1, c=0.01):
    return -a*SIRV[0]*SIRV[1] - c*SIRV[0]

def INF(SIRV, a=0.5, b=0.1, c=0.01):
    return a*SIRV[0]*SIRV[1] - b*SIRV[1] - c*SIRV[1]

def REC(SIRV, a=0.5, b=0.1, c=0.01):
    return b*SIRV[1] - c*SIRV[2]

def VAC(SIRV, a=0.5, b=0.1, c=0.01):
    return c*SIRV[0] + c*SIRV[1] + c*SIRV[2]
```

Folgend das Ergebnis der Berechnung in Figure 3:

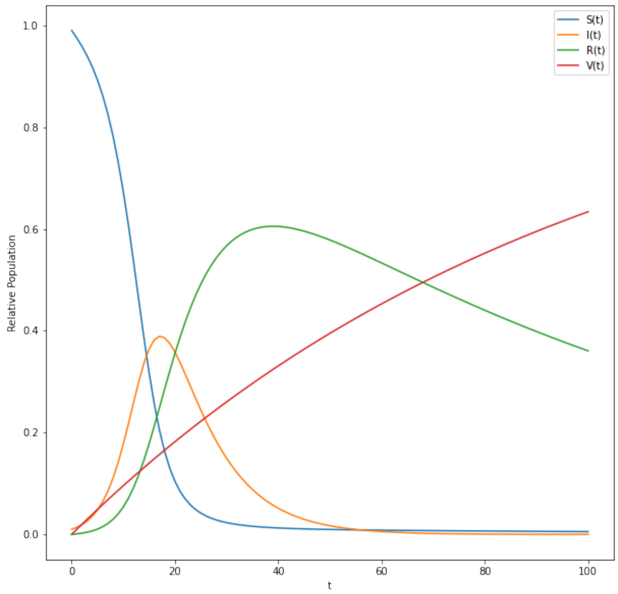

Figure 3: Lösung des erweiterten SIR-Modells mittels explizitem Euler-Verfahren

Appendix

Code für explizites Euler-Verfahren:

```
def explicit_euler(f, u0, t_disc):

    # neuer array mit u0 als erstem Element. Wenn u0 ein Vektor ist,
    # wird u so ein zweidimensionaler Array
    u = np.zeros((len(t_disc), len(u0))) #Loesungsmatrix
    u[0] = u0                            #Anfangswert

    for i in range(1, len(t_disc)):
        t_last = t_disc[i-1]
        t = t_disc[i]
        tau = t-t_last

        # der letzte berechnete Wert von u
        u_last = u[i-1]

        # Ausfuehrung des Zeitschritts
        u_new = u_last + tau*f(t_last, u_last)

        # Speichern des neuen Werts
        u[i] = u_new

    return u
```

Code für implizites Euler-Verfahren:

```
def implizit_euler(f, u0, t_disc):
    u = np.zeros((len(t_disc), len(u0)))
    u[0] = u0

    for i in range(1, len(t_disc)):

        t_last=t_disc[i-1]
        t = t_disc[i]
        tau = t-t_last
        u_last = u[i-1]
        # bis hier analog zu euler

        # berechne ui anhand der im Notebook gegebenen Gleichung
        u[i] = (u_last + tau*-1*np.sin(3*t))/(1-(tau/3))
    return u
```

Code für Heun-Verfahren:

```
def heun(f, u0, t_disc):

    u = np.zeros((len(t_disc), len(u0)))
    u[0] = u0

    for i in range(1, len(t_disc)):
        t_last=t_disc[i-1]
        t = t_disc[i]
        tau = t-t_last
        u_last = u[i-1]
        # bis hier analog zu euler

        #berechne k1 und k2
        k1 = f(t_last, u_last)
        k2 = f(t, u_last + tau*k1)

        #Kombiniere zu neuer Loesung
        u[i] = u_last + tau*(1/2)*(k1 + k2)
    return u
```

Code für Runge-Kutta-Verfahren:

```
def runge_kutta(f, u0, t_disc):

    u = np.zeros((len(t_disc), len(u0)))
    u[0] = u0

    for i in range(1, len(t_disc)):

        t_last=t_disc[i-1]
        t = t_disc[i]
        tau = t-t_last
        u_last = u[i-1]
        # bis hier analog zu euler

        #berechne k1,k2,k3 und k4
        tau_h = tau * (1/2)
        k1 = f(t_last, u_last)
        k2 = f(t_last + tau_h, u_last+tau_h*k1)
        k3 = f(t_last + tau_h, u_last+tau_h*k2)
        k4 = f(t, u_last+tau*k3)

        #Kombiniere zu neuer Loesung
        u[i] = u_last + tau*(1/6)*(k1+(2*k2)+(2*k3)+k4)
    return u
```