Python Coding Practical 2: Solutions

Task 1.

The state-space and function vectors simply combine the three equations and variables, for easier notation and application of the numerical methods. The state-space vector for the system is

$$\boldsymbol{u}(t) = \begin{bmatrix} H(t) \\ Z(t) \\ R(t) \end{bmatrix}$$

with initial conditions at time t=0

$$\boldsymbol{u}(0) = \begin{bmatrix} H(0) \\ Z(0) \\ R(0) \end{bmatrix} = \begin{bmatrix} H_0 \\ Z_0 \\ R_0 \end{bmatrix}$$

The function **f** defining the time derivative of the state-space (vector of unknowns) $\dot{\boldsymbol{u}}$ then becomes

$$\dot{\boldsymbol{u}}(t) = \boldsymbol{f}(u(t), t) = \boldsymbol{f}\left(\begin{bmatrix} H(t) \\ Z(t) \\ R(t) \end{bmatrix}\right) = \begin{bmatrix} \dot{H}(t) \\ \dot{Z}(t) \\ \dot{R}(t) \end{bmatrix} = \begin{bmatrix} -\beta\, H(t)Z(t) \\ \beta\, H(t)Z(t) + \zeta R(t) - \alpha H(t)Z(t) \\ \alpha H(t)Z(t) - \zeta R(t) \end{bmatrix}$$

Task 2.

The numerical techniques discussed in the lectures can now be applied more conveniently to the system of differential equations. For instance, applying the forward Euler method to the 'vectorised' equation $\dot{\boldsymbol{u}}(t) = \boldsymbol{f}(\boldsymbol{u}(t), t)$, yields

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \Delta t\, f(\boldsymbol{u}(t_n), t_n)$$

where t_n is the time position, n, given by $t_n = n\Delta t$ and $t_0=0$.

Writing the vectors out fully makes it evident how simple the method is in application

$$\boldsymbol{u}^{n+1} = \begin{bmatrix} H(t_{n+1}) \\ Z(t_{n+1}) \\ R(t_{n+1}) \end{bmatrix} = \begin{bmatrix} H(t_n) \\ Z(t_n) \\ R(t_n) \end{bmatrix} + \Delta t \begin{bmatrix} -\beta\, H(t_n)Z(t_n) \\ \beta\, H(t_n)Z(t_n) + \zeta R(t) - \alpha H(t_n)Z(t_n) \\ \alpha H(t_n)Z(t_n) - \zeta R(t_n) \end{bmatrix}$$

using the given initial values for the state-space vector $\mathbf{u}(t_0)=[H_0,Z_0,R_0]$ to start off the time loop.

Implementing the above model, we can consider a range of zombie outbreak scenarios.

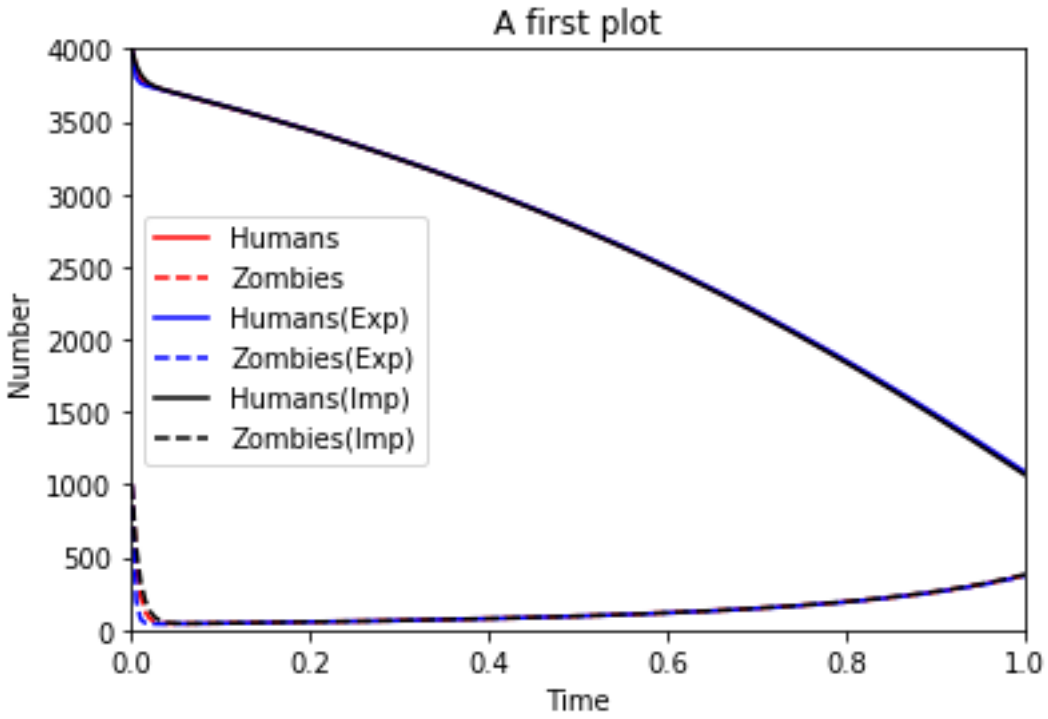
A Python code for implementing this is given in the file **practical2_task2.py:**

```
# practical2_task2.py

import numpy as np

from scipy.integrate import odeint

from scipy.optimize import fsolve


#############################################################################

# Functions
```

```
###############################################################################
# time derivative for zombie function: u=(H,Z,R)^T for use with odeint
# note has argument t so can be used with odeint
def f_odeint(u,t,alpha,beta,zeta):
    u0dot = -beta*u[0]*u[1]
    u1dot = beta*u[0]*u[1]+zeta*u[2]-alpha*u[0]*u[1]
    u2dot = alpha*u[0]*u[1]-zeta*u[2]
    return [u0dot,u1dot,u2dot]

# time derivative for zombie function: u=(H,Z,R)^T for use with exp/imp euler
def f(u,alpha,beta,zeta):
    u0dot = -beta*u[0]*u[1]
    u1dot = beta*u[0]*u[1]+zeta*u[2]-alpha*u[0]*u[1]
    u2dot = alpha*u[0]*u[1]-zeta*u[2]
    return [u0dot,u1dot,u2dot]

# Forward Euler
def exp_euler(u0,tend,nsteps,f,alpha,beta,zeta):
    dt = tend/nsteps
    u = np.zeros([nsteps+1,3])
    u[0,:] = u0
    for i in range(nsteps):
        u[i+1,:] = u[i,:] + dt*np.array(f(u[i,:],alpha,beta,zeta))
    return u

# define the function which needs to be solved at each implicit time step
def F_imp(u,dt,u_init,alpha,beta,zeta):
    return u - dt*np.array(f(u,alpha,beta,zeta)) - u_init

# Backward Euler
def imp_euler(u0,tend,nsteps,f,alpha,beta,zeta):
    dt = tend/nsteps
    u = np.zeros([nsteps+1,3])
    u[0,:] = u0
    for i in range(nsteps):
        u_init = u[i,:]
        sol = fsolve(F_imp,u_init,args=(dt,u_init,alpha,beta,zeta))
        u[i+1,:] = sol
    return u
###############################################################################
```

```
alpha=0.05
beta=0.01
zeta=5
N=200
t_end=1.0
t_axis = np.linspace(0, t_end, N+1)
dt=t_end/N

H_0=4000
Z_0=1000
R_0=0
u0 = [H_0,Z_0,R_0]

# odeint solution
# Solve the problem using odeint
taxis = np.linspace(0,t_end,N+1)
u_odeint = odeint(f_odeint,u0,taxis,args=(alpha,beta,zeta,))

# Explicit Euler solution
u_exp = exp_euler(u0,t_end,N,f,alpha,beta,zeta)

# Implicit Euler solution
u_imp = imp_euler(u0,t_end,N,f,alpha,beta,zeta)

# plotting
import matplotlib.pyplot as plt
plot1 = plt.figure(1)
plt.plot(taxis,u_odeint[:,0],'r')
plt.plot(taxis,u_odeint[:,1],'r--')
plt.plot(taxis,u_exp[:,0],'b')
plt.plot(taxis,u_exp[:,1],'b--')
plt.plot(taxis,u_imp[:,0],'k')
plt.plot(taxis,u_imp[:,1],'k--')
plt.xlim([0,1])
plt.ylim([0,4000])
plt.xlabel('Time')
plt.ylabel('Number')
plt.legend(['Humans','Zombies','Humans(Exp)','Zombies(Exp)','Humans(Imp)','Zombies(Imp)'])
plt.title('A first plot')
```

Running the program practical2_task2.py generates the following result with $\alpha=0.05$, $\beta=0.01$ and $\zeta=5$, with $H_0=4000$, $Z_0=1000$ and $R_0=0$.



Task 3.

The idea here is to rewrite the matrix system so that

$$M\boldsymbol{u}^{n+1} = \boldsymbol{u}^{n}$$

Now the linearised system is:

$$A = \pi r^2$$

$$\dot{\tilde{H}} = -\beta\bar{H}\tilde{Z}(t)$$

$$\dot{\tilde{Z}} = \beta\bar{H}\tilde{Z}(t) + \zeta\tilde{R}(t) - \alpha\bar{H}\tilde{Z}(t)$$

$$\dot{\tilde{R}} = \alpha\bar{H}\tilde{Z}(t) - \zeta\tilde{R}(t)$$

so that now

$$\boldsymbol{u}(t) = \begin{bmatrix} \tilde{H}(t) \\ \tilde{Z}(t) \\ \tilde{R}(t) \end{bmatrix}$$

and

$$\dot{\boldsymbol{u}}(t) = \boldsymbol{f}(u(t), t) = \begin{bmatrix} 0 & -\beta\tilde{H} & 0 \\ 0 & \beta\bar{H} - \alpha\bar{H} & \zeta \\ 0 & \alpha\bar{H} & -\zeta \end{bmatrix} \begin{bmatrix} \tilde{H}(t) \\ \tilde{Z}(t) \\ \tilde{R}(t) \end{bmatrix}$$

This can be written as:

$$\dot{\boldsymbol{u}}(t) = \boldsymbol{f}(u(t), t) = \begin{bmatrix} 0 & -\beta\tilde{H} & 0 \\ 0 & \beta\bar{H} - \alpha\bar{H} & \zeta \\ 0 & \alpha\bar{H} & -\zeta \end{bmatrix} \begin{bmatrix} \tilde{H}(t) \\ \tilde{Z}(t) \\ \tilde{R}(t) \end{bmatrix}$$

$$\dot{\boldsymbol{u}}(t) = F\begin{bmatrix}\tilde{H}(t)\\ \tilde{Z}(t)\\ \tilde{R}(t)\end{bmatrix} \text{ where } F = \begin{bmatrix}0 & -\beta\tilde{H} & 0\\ 0 & \beta\bar{H} - \alpha\bar{H} & \zeta\\ 0 & \alpha\bar{H} & -\zeta\end{bmatrix}$$

The backward Euler solution of this system of equations with a first order time discretisation results in:

$$\frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^{n}}{\Delta t} = F\,\boldsymbol{u}^{n+1}$$

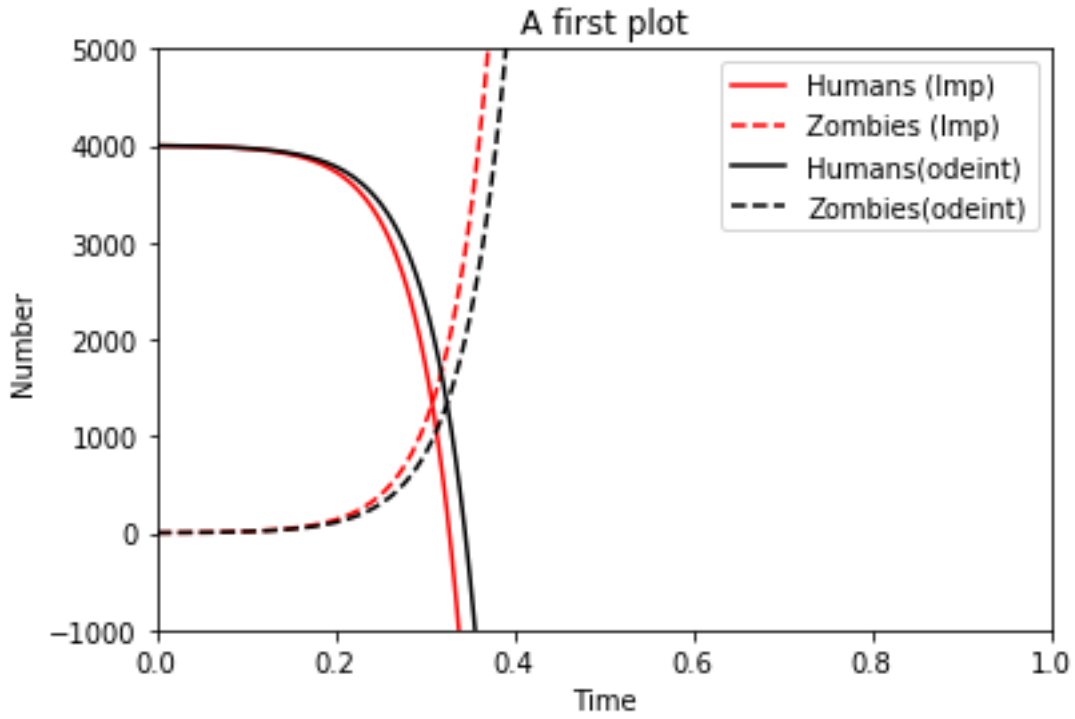
which can be written as $(I_3 - \Delta t\, F)\boldsymbol{u}^{n+1} = \boldsymbol{u}^{n}$ where I_3 is the 3x3 unit matrix. Hence if $M = (I_3 - \Delta t\, F)$, the backward Euler equations at every time step can be solved by $\boldsymbol{u}^{n+1} = M^{-1}\boldsymbol{u}^{n}$.

A matlab code implementation of the linear system is given in **practical2_task4.py**:

```
# practical2_task4.py

import numpy as np

from scipy.integrate import odeint


# time derivative for linearised analysis - note need argument to use odeint

def f(u,t,A):

    return np.matmul(A,u)


# initialise parameters

alpha=0.005

beta=0.01

zeta=0.02

N=200

t_end=1.0

t_axis = np.linspace(0, t_end, N+1)

dt=t_end/N

H_0=4000

H=H_0

Z_0=2

R_0=0

u0 = [H_0,Z_0,R_0]


# define the linearised matrix A

A = np.array([[0,-beta*H,0], [0,beta*H-alpha*H,zeta], [0,alpha*H,-zeta]])


# Implicit euler solution

M = np.identity(3) - dt*A

u_imp = np.zeros([N+1,3])

u_imp[0,:] = u0
```

```
for i in range(N):
    u_imp[i+1,:] = np.linalg.solve(M,u_imp[i,:])


# solve with odeint
u_odeint = odeint(f,u0,t_axis,args=(A,))


# plotting
import matplotlib.pyplot as plt
plot1 = plt.figure(1)
plt.plot(t_axis,u_imp[:,0],'r')
plt.plot(t_axis,u_imp[:,1],'r--')
plt.plot(t_axis,u_odeint[:,0],'k')
plt.plot(t_axis,u_odeint[:,1],'k--')
plt.xlim([0,1])
plt.ylim([-1000,5000])
plt.xlabel('Time')
plt.ylabel('Number')
plt.legend(['Humans (Imp)','Zombies (Imp)','Humans(odeint)','Zombies(odeint)'])
plt.title('A first plot')
```

For the case with H_0=4000, Z_0=2 and R_0=0 and α=0.005, β=0.01 and ζ=0.02 this generates the result:



Note that the rate of death of humans, production of zombies and death of zombies all depend on the initial 'background' value of humans $\bar{H} = H_0$. Hence the dynamics of the

system are ‘blind’ to the actual population of humans. This means that rather than the death-rate levelling out, it continues to reach a negative number of humans!