Simulation of within-host dynamics in patients infected by HIV

Aslihan Celik, Anirudh Choudhary, Farshad Rafiei

Group 2.27

Modeling & Simulation (CSE6730) Final project

GitHub Repository: https://github.gatech.edu/frafiei3/CSE6730

Spring 2020

Tutorial: Simulation of within-host dynamics in patients infected by HIV

The goal of this tutorial is to model the “viral dynamics” of HIV infection and examine the effectiveness of antiretroviral drug when used to treat HIV patients. Generally, HIV disease progression consist of three main phases: acute, chronic and AIDS. Each of these phases are characterized by changes in CD4+ T-cell count and the plasma viral load. The first part of this project includes simulating the first phase of virus spread using stochastic agent-based modeling of HIV transmission. We used “cell-to-cell transmission” hypothesis for this reason to simulate the T-cell dynamics in acute phase. In the second part of the project, we extended the analysis by discrete time modeling of differential equations which is used to explain the HIV infection kinetics in AIDS phase as well as system’s behavior when undergoes a long-term treatment. Finally, we opt for reinforcement learning-based approach to determine optimal treatment strategy for patients with HIV and use a ODE simulation model to generate the patient clinical data. This ODE model takes into account drug combinations and we compare the performance of RL-based model with 'high drug' dosage and 'no drug dosage' strategies, tracking their physiological response to separate classes of treatments and determine the optimal drug level to be administered to the patient.

Through this tutorial, we develop three ways to model and analyse the HIV dynamics and response to treatment with inhibitors (drugs). Our models are based on empirically motivated HIV models developed in various studies.To validate our models, we run simulations with different model parameters and analyze the infection dynamics.

Part 1 - Cellullar Automata Model : In this section, our aim is to simulate the first phase of HIV infection (acute phase) and show how an arbitrary initial infection in a lattice like cell population can lead in progression of virus within a host body[1]. https://github.gatech.edu/frafiei3/CSE6730/blob/master/hiv%20cellular%20automata%20simulations.ipynb (https://github.gatech.edu/frafiei3/CSE6730/blob/master/hiv%20cellular%20automata%20simulations.ipynb)

Part 2 - ODE-based Mean-Field Model: In this section, we focus on modeling the spread of virus in third phase (AIDS) and try to simulate the HIV growth dynamics using the concept of ordinary differential equations (ODEs) [2,3]. We examine the effect of treatment on our simulations and see how this can slow down and even decrease the growth of infection within a host. https://github.gatech.edu/frafiei3/CSE6730/blob/master/hiv%20mean%20field%20simulations.ipynb (https://github.gatech.edu/frafiei3/CSE6730/blob/master/hiv%20mean%20field%20simulations.ipynb)

Part 3 - ODE and Reinforcement Learning based treatment strategy: Finally, we opt for reinforcement learning to find optimal treatment plan and use a ODE model proposed by Adams et. al.[5] to simulate patients with HIV. Such treatment plans are also referred to as Structured Treatment Intervention (STI). Various studies[5][6] have explored using mathematical models of HIV infection dynamics for addressing the problem of designing STI treatments. These models are usually represented by a set of Ordinary Differential Equations(ODEs) and control theory is applied to deduce STI strategies. Reinforcement Learning(RL) computes control strategy directly from the measured trajectories and does not need the apriori identification of model of system dynamics.
https://github.gatech.edu/frafiei3/CSE6730/blob/master/hiv%20treatment%20RL.ipynb (https://github.gatech.edu/frafiei3/CSE6730/blob/master/hiv%20treatment%20RL.ipynb)

hiv cellular automata simulations

April 28, 2020

1 Part 1: Cellular Automata

In the first part of this tutorial we'll apply the concept of Cellular Automata (CA) to model HIV disease progression in acute phase

1.0.1 The phenomenon to be modeled and simulated

The immune response to any virus is generated by a complex web of interactions among different types of white blood cells (monocytes, T and B cells). The time scale to develop a specific immune response may vary from days to weeks. In the case of HIV, the entire course of infection involves two different time scales. The primary infection exhibits the same characteristics as any other viral infection: a dramatic increase of the virus population during the first 2–6 weeks, followed by a sharp decline, due to the action of the immune system. However, instead of being completely eliminated after the primary infection, as many other viruses, a low HIV concentration is detected for a long asymptomatic time: the clinical latency period. This period may vary from one to ten (or more) years. Besides the low virus burden detected during this period, a gradual deterioration of the immune system is manifested by the reduction of CD4+T-cell populations in the peripheral blood. The third phase of the disease is achieved when the concentration of the T cells is lower than a critical value (~30%), leading to the development of AIDS. As a consequence, the patient normally dies from opportunistic diseases. In this section our aim is to simulate the first phase of HIV infection (acute phase) and show how an arbitrary initial infection in a lattice like cell population can lead in progression of virus within a host body. - **Reference**: https://journals.aps.org/prl/pdf/10.1103/PhysRevLett.87.168102

1.0.2 Conceptual Model

Let the world be a square nxn grid $G = G(t)$ of cells that evolve over time, which is discrete and measured in weeks. Every cell of G shows a T-cell which could potentially be in one of the following states: * **UNINFECTED:** Normal state of T-cell at the beginning of the simulation. This states mean that the cell is uninfected and hence it is in a healthy state. * **INFECTED:** The T-cell is infected. It takes τ weeks for the infected T-cell to die. * **DEAD:** The T-cell is dead at this state. It can be replaced with an uninfected T-cell by immune sytem (with probability p_repl) or remain dead.

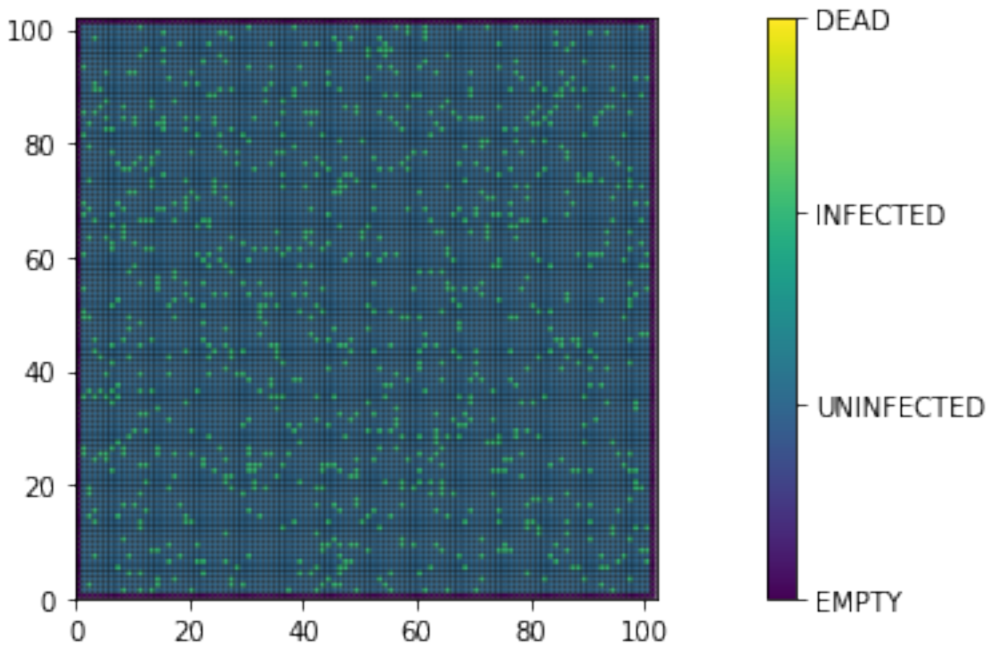
Let's associate these states with the following integers:

[1]:
```python
import numpy as np
import scipy as sc
import scipy.sparse
import random
import matplotlib.pyplot as plt
from ipywidgets import interact


# Possible states:
EMPTY = -1
UNINFECTED = 0
INFECTED = 1
DEAD = 2
```

The initial configuration is composed of healthy cells, with a small fraction, p_HIV, of infected cells, representing the initial contamination by the HIV. The follwoing function creates a $(n+2)x(n+2)$ T-cell population lattice with assigning empty to surrounding cells and an initial configuration to the interior cells by randomly assigning INFECTED to p_HIV of them and leave rest of them to be equal to UNINFECTED cells.

[2]:
```
def GridMap(n, I_fraction):
    """
    Returns an n by n NumPy array of integer values that are empty on the␣
 ↪boundary
    and an initial configuration in interior with some cells marked as infected
    and rest of them marked as uninfected

    """
    GM = EMPTY * np.ones(shape=(n+2,n+2), dtype=int)
    GM[1:-1, 1:-1] = UNINFECTED
    num_Infection = int(I_fraction * n * n)
    sequence = [i for i in range(n)]
    idx_row, idx_col = np.zeros(num_Infection), np.zeros(num_Infection)
    for i in range(num_Infection):
        idx_row[i] = random.choice(sequence) + 1
        idx_col[i] = random.choice(sequence) + 1

    for i in range(num_Infection):
        GM[int(idx_row[i]), int(idx_col[i])] = INFECTED

    return GM
```

[3]:
```
def show_peeps(GM, vmin=EMPTY, vmax=DEAD, values="states"):
    """
    A helper routine to visualize a 2-D world
    """
    assert values in ["states", "bool"]
    if values == "states":
        vticks = range(vmin, vmax+1)
        vlabels = ["EMPTY", "UNINFECTED", "INFECTED", "DEAD"]
    else:
        vticks = [0, 1]
        vlabels = ["False (0)", "True (1)"]

    m, n = GM.shape[0]-2, GM.shape[1]-2
    plt.pcolor(GM, vmin=vmin, vmax=vmax, edgecolor='black')
    cb = plt.colorbar()
    cb.set_ticks(vticks)
    cb.set_ticklabels(vlabels)
    plt.axis('square')
    plt.axis([0, m+2, 0, n+2])

# Create an initial world
N = 100
p_HIV = 0.1

peeps_0 = GridMap(N, p_HIV)
show_peeps(peeps_0)
```



Let's define some functions to help us identify uninfected, infected and dead T-cells in our world and calculate the ratio of them with respect to the total number of T-cells in the world

[4]:
```
def uninfected(GM):
    """
    Given a grid map, GM, it returns:
```

```
    - a boolean grid whose (i,j) entry equals 1 when GM[i,j] is uninfected and␣
↪0 otherwise,
    - ratio of the uninfected cells to total number of cells
    """
    GM_uninfected = (GM==UNINFECTED).astype(int)
    num_uninfected = len(np.where(GM_uninfected)[0])
    ratio = num_uninfected / ((GM.shape[0]-2) * (GM.shape[1]-2))
    return GM_uninfected, ratio

def infected(GM):
    """
    Given a grid map, GM, it returns:
    - a boolean grid whose (i,j) entry equals 1 when GM[i,j] is infected and 0␣
↪otherwise,
    - ratio of the infected cells to total number of cells
    """
    GM_infected = (GM==INFECTED).astype(int)
    num_infected = len(np.where(GM_infected)[0])
    ratio = num_infected / ((GM.shape[0]-2) * (GM.shape[1]-2))
    return GM_infected, ratio

def dead(GM):
    """
    Given a grid map, GM, it returns:
    - a boolean grid whose (i,j) entry equals 1 when GM[i,j] is dead and 0␣
↪otherwise,
    - ratio of the dead cells to total number of cells
    """
    GM_dead = (GM==DEAD).astype(int)
    num_dead = len(np.where(GM_dead)[0])
    ratio = num_dead / ((GM.shape[0]-2) * (GM.shape[1]-2))
    return GM_dead, ratio
```

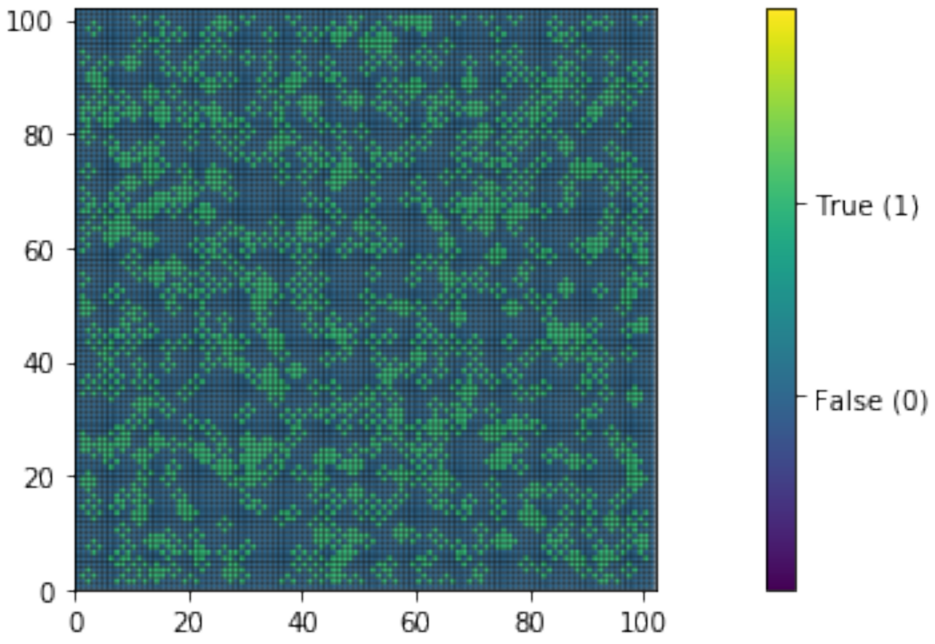
Time evolution Let's define rules which determined how the infection spreads within the host. Each of the time steps used in this simulation is equivalent to a week and includes following:

- **Update of a healthy cell** If it has at least R (R $\{1, 2, 3, 4\}$) infected neighbors, it becomes infected. Otherwise, it stays healthy.
- **Update of an infected cell** An infected cell becomes a dead cell after τ time steps.
- **Update of a dead cell** A dead cell can be replaced by a healthy cell with probability *p_repl* in the next time step. Each healthy cell can get infected again.

To help determine which cells are prone to infection in a given time step, let's write a function to determine who is exposed.

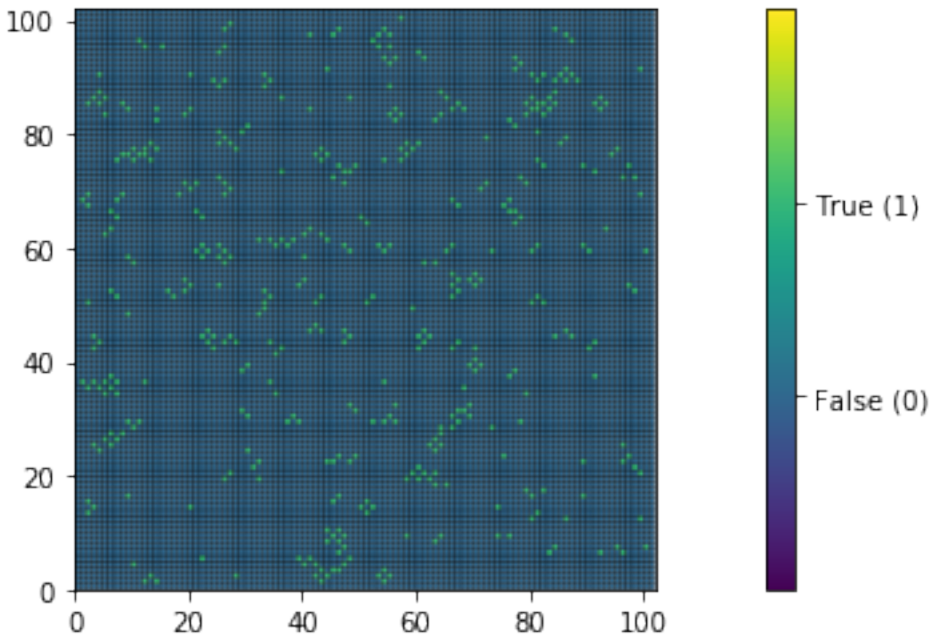
```
[5]: def exposed(GM):
         """
```

```
    Given a grid map, GM, returns a boolean grid whose (i,j) entry equals 1
    when GM[i,j] has at least one infected neighbor, and 0 otherwise.
    """
    E = np.zeros(shape=GM.shape, dtype=int)
    I, _ = infected(GM)
    E[1:-1, 1:-1] = I[0:-2, 1:-1] | I[1:-1, 2:] | I[2:, 1:-1] | I[1:-1, 0:-2]
    return E

show_peeps(exposed(peeps_0), values="bool")
```



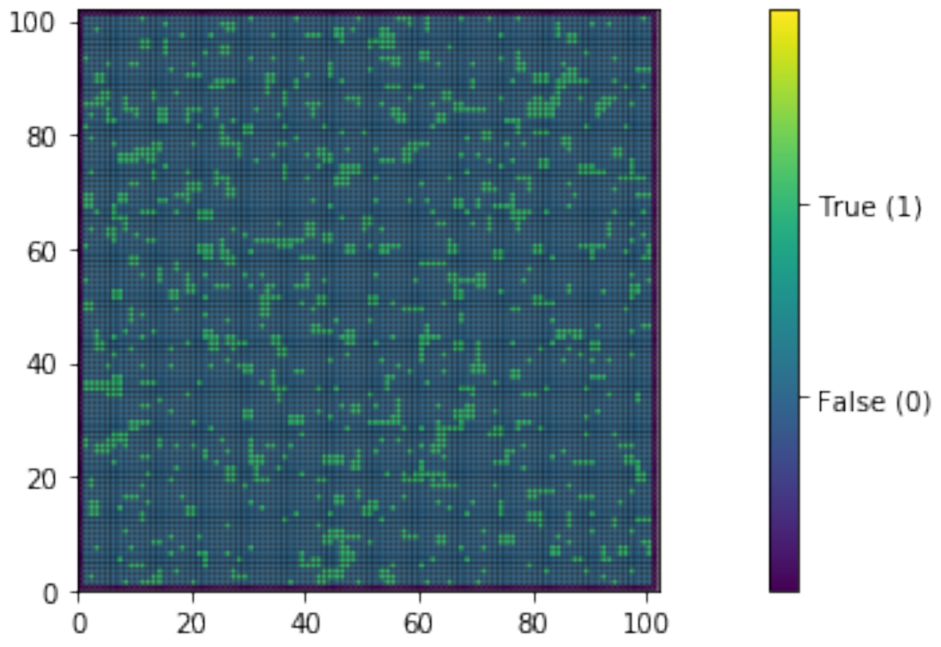
To determine the infected cells in subsequent time step, we need a function to count the number of infected cells surrounding a specific cell. Given this function, we can now determine how the infection spreads in next time step.

[6]:
```
def count_surrounding(GM):
    """
    Given grid map, GM, returns a grid whose (i,j) entry equals
    to the number of infected neighbors for element GM[i,j].
    """
    C = np.zeros(shape=GM.shape, dtype=int)
    I, _ = infected(GM)
    C[1:-1, 1:-1] = I[0:-2, 1:-1] + I[1:-1, 2:] + I[2:, 1:-1] + I[1:-1, 0:-2]
    return C

def spreads(GM, threshold=1):
    """
    Given grid map, GM, returns a boolean grid whose (i,j) entry equals
    to 1 when the cell GM[i,j] has all conditions to get infection in
    subsequent time step, and 0 otherwise.
    """
    threshold = threshold # minimum number of neighbors needed to infect an␣
↪uninfected cell
    UI,_ = uninfected(GM)
    G_s = (UI * exposed(GM) * (count_surrounding(GM) > threshold))
    return G_s

show_peeps(spreads(peeps_0), values="bool")
```



Now we can write a routine to simulate one time step to determine the spread, given grid map GM.

[7]:
```
def step_spread(GM):
    """
    Simulates one time step of a SPREAD and
    returns a grid of the resulting states
    """
    return GM + spreads(GM)

show_peeps(step_spread(peeps_0), values="bool")
```



Next, we need to implement a function that accounts for death of infected cells in τ time steps. We also, need a function to replace dead cells with healthy cells after one time step, with some probability.

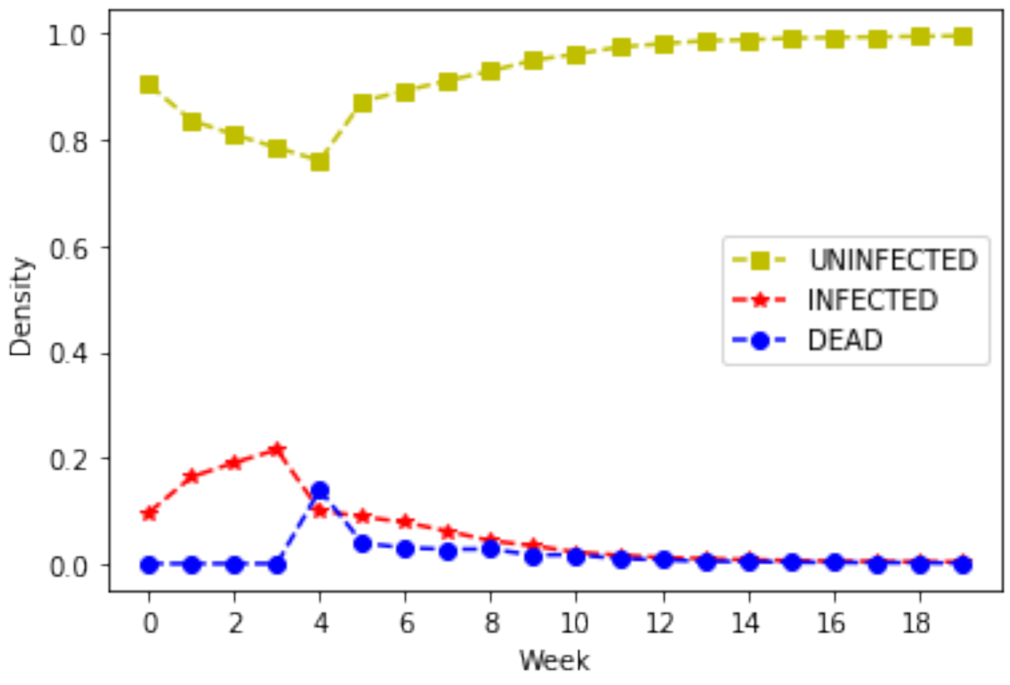
[9]:
```
def recover(GM, p_repl):
    """
    Given grid map, GM, and replacement probability, p_repl,
    returns a boolean grid whose (i,j) element equals to 0, when
    the GM[i,j] replaces with a healthy cell, and 1 otherwise.
    Note: 0 selcetd because it'll set the elemnt in GM to UNINFECTED state.
    As such, 1 will leave the GM[i,j] element unchanged.
    """
    random_draw = np.random.uniform(size=GM.shape)
    D, _ = dead(GM)
    G_r = (D * (random_draw < p_repl))
    return (1-G_r).astype(int)

def step_dead(GM, GM_tau, tau):
    """
    Given grid map, GM, and tau grid map history of GM, returns a grid map
    after replacing infected cells, which were infected in tua steps before,
    by DEDA state (2). It does not change the status of other cells.
    """
    I = np.ones(GM.shape)
    for i in range(tau):
        I1, _ = infected(GM_tau[i,:,:])
        I = I * I1
    GM_d = GM + I
    return GM_d
```

It's time to combine all we have together to see what happens when a grid like cell structure gets infected with HIV virus. In the following, we set the max steps to 100, which shows the maximum number of generations that our simulation will take. Also, we set $\tau = 4$, which indicates that it takes 4 time steps (i.e. weeks) for an infected cell to become a dead cell. Finally, we set the probability of cell recovery after its death to 90%.

[10]:
```
def sim(G_0, max_steps=100, tau=4, p_repl=0.90):
    """
    Given an initial grid map, G_0, returns max_steps generations of HIV spread

    tau: represents the time required for the immune system to
    develop a specific response to kill an infected cell
    p_repl: probability by which dead cells could be replaced with healthy cells
    """
    # In the first tau steps, there will only be spread (no dead cells):
    G_null = GridMap(G_0.shape[0]-2, 0)
    G_all = np.repeat(G_null[np.newaxis,:,:], max_steps, axis=0)
    G_all[0,:,:] = step_spread(G_0)
    for idx in range(1,tau):
        G_all[idx,:,:] = step_spread(G_all[idx-1,:,:])

    # Infected cells will die in tau step and immune system will recover them␣
↪with probability p_repl:
    _, infected_ratio = infected(G_all[tau-1,:,:])
    t = tau
    while(infected_ratio > 0) and (t < max_steps):
        G_t = step_spread(G_all[t-1,:,:]) # spread the virus
        G_t = G_t * recover(G_t, p_repl) # recover dead cells with p_prepl␣
↪probability
        G_t = step_dead(G_t, G_all[t-tau:t,:,:], tau) # tau-step before␣
↪infected cells become dead cells
        G_all[t,:,:] = G_t # store the result

        # Update the stop criterion
        _, infected_ratio = infected(G_t)
        t = t + 1

    return G_all

test = sim(peeps_0)
```

[13]:
```
def compute_ratio(G_0, GM_all):
    """
    Given initial configuraion and all generations of spread,
    returns three matrices, which shows uninfected cell ratio,
    infected cell ratio and dead cell ratio to the total number
    of cells, respectively.
    """
    uninfected_ratio = np.zeros(GM_all.shape[0]+1)
    infected_ratio = np.zeros(GM_all.shape[0]+1)
    dead_ratio = np.zeros(GM_all.shape[0]+1)

    _, uninfected_ratio[0] = uninfected(G_0)
    _, infected_ratio[0] = infected(G_0)
    _, dead_ratio[0] = dead(G_0)

    for i in range(1, GM_all.shape[0]):
        _, uninfected_ratio[i] = uninfected(GM_all[i,:,:])
        _, infected_ratio[i] = infected(GM_all[i,:,:])
        _, dead_ratio[i] = dead(GM_all[i,:,:])


    return uninfected_ratio, infected_ratio, dead_ratio

UR, IR, DR = compute_ratio(peeps_0, test)
plt.plot(UR[:20], 'ys--')
plt.plot(IR[:20], 'r*--')
plt.plot(DR[:20], 'bo--')
plt.legend(['UNINFECTED', 'INFECTED', 'DEAD'])
plt.xticks(np.arange(0, 20, step=2))
plt.xlabel('Week')
plt.ylabel('Density')
```

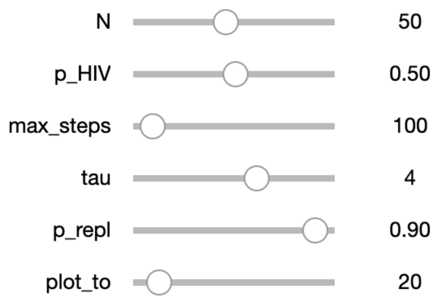
```
[13]: Text(0, 0.5, 'Density')
```



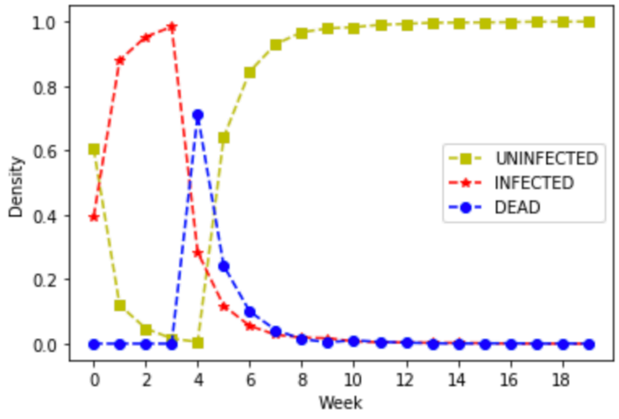
[12]:
```
def isim(N, p_HIV, max_steps=100, tau=4, p_repl=0.9, plot_to=20):
    G_0 = GridMap(N, p_HIV)
    G_t = sim(G_0, max_steps=max_steps, tau=tau, p_repl=p_repl)
    UR, IR, DR = compute_ratio(G_0, G_t)
    plt.plot(UR[:plot_to], 'ys--')
    plt.plot(IR[:plot_to], 'r*--')
    plt.plot(DR[:plot_to], 'bo--')
    plt.legend(['UNINFECTED', 'INFECTED', 'DEAD'])
    plt.xticks(np.arange(0, plot_to, step=2))
    plt.xlabel('Week')
    plt.ylabel('Density')

interact(isim,
        N = (10,100,10),
        p_HIV = (0.1,0.9,0.1),
        max_steps = (20,1000,2),
        tau = (1,6,1),
        p_repl = (0,1,0.1),
        plot_to = (10,100,10));
```

```
interactive(children=(IntSlider(value=50, description='N', min=10, step=10), FloatSlider(value
```

[ ]:



```
N          50
p_HIV      0.50
max_steps  100
tau        4
p_repl     0.90
plot_to    20
```



The following shows the simulation from the reference paper which is very similar to our simulations. Note that, in this section, we aimed just to simulate the behavior after weeks of infection and hence our simulations should reflect the behavior of left figure in the following.

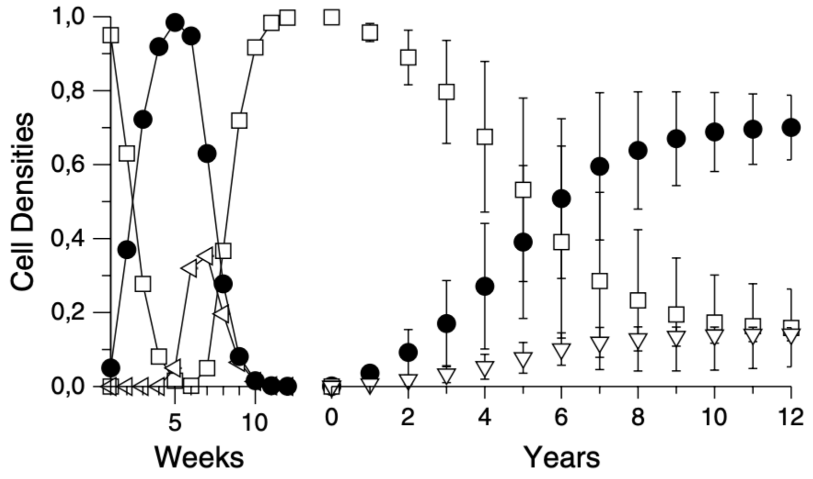


FIG. 2. The results obtained from our simulations for a two-dimensional lattice with $L = 700$, $p_{\text{HIV}} = 0.05$, $R = 4$, $\tau = 4$, $p_{\text{infec}} = 10^{-5}$, $p_{\text{repl}} = 0.99$. The evolution of the population densities exhibits the same three-phase dynamics observed for infected patients. We have adopted *open squares* for healthy cells, *full circles* for infected cells, and *open triangles* for dead cells.

Part 2: HIV Simulations Using Mean-Field Model

Introduction

Modeling in HIV has proven to be helpful in many ways. Specifically the authors of [1] note that because of insights gained from modeling they conclude that more emphasis needs to be put on finding a vaccine instead of treatment. This conduction was made because models show that viral loads persist even in long term simulations.

Modelling HIV dynamics are useful to compare the efficency levels of different treatments. Paper [2] explores the HIV models of patients that are treated with Highly Active Antiretroviral Therapy (HAART). According to paper, patients mostly achieve undetectable viral loads when they are treated with HAART for long periods of time.

Both of the papers referenced, review developments in HIV modeling. They display the quantitative discoveries about HIV, the rate of generation of HIV variants, treatments and response to drug therapy. In this tutorial we are simulating the HIV models from papers [1] and [2] by using the Mean-field Model to observe characteristics of HIV infection and to provide insight into the treatments.

Background

In implementing the discrete and continuous simulations of three HIV models including the models both before and after treatment, the following papers are used:

[1]Perelson, Alan S., and Ruy M. Ribeiro. "Modeling the within-host dynamics of HIV infection." BMC biology 11.1 (2013): 96. https://link.springer.com/content/pdf/10.1186/1741-7007-11-96.pdf (https://link.springer.com/content/pdf/10.1186/1741-7007-11-96.pdf)

[2]Di Mascio, Michele, et al. "Modeling the long-term control of viremia in HIV-1 infected patients treated with antiretroviral therapy." Mathematical biosciences 188.1-2 (2004): 47-62. https://www.sciencedirect.com/science/article/pii/S0025556403001305 (https://www.sciencedirect.com/science/article/pii/S0025556403001305)

In [25]:
```
import numpy as np
import matplotlib.pyplot as plt
```

First HIV model -- Modeling virus growth and infected cells.

Following the notation from we can begin to implement our first model using the notation and equations from [1]

In first HIV model initially each cell has two types of possible states which are "T" and "I". "T" represents the uninfected target cells while "I" represents the infected cells."T" cells are mostly CD4+ T cells which are susceptible for infection. "V" represents the free virus. In a mean-field model, you would first define a time-dependent variable for each cell states which you interpret as the fraction of the population in that state and a dependent variable for the virus. That is, let

- T_t be the fraction of the cells that is susceptible at (discrete) time t;
- I_t be the fraction that is infected at t; and
- V_t be the fraction of virus at t,

where $T_t + I_t = 1$ since infected cells I and uninfected target cells T are complementary we always only need to compute one in order to compute the other. We will implicitly assume that the number of individuals is large enough that we can treat these fractions as being continuous.

- λ is a parameter that represents the constant rate per cell that T cells are produced
- d_T is a parameter that represents the die rate of T cells per cell
- $\beta V T$ represents the rate that T cells get infected by free virus
- δ is the rate that I cells are lost
- p is the rate per cell that V (free viruses) are produced by I cells
- c is the rate per virus that V are cleared from circulation

Then, a corresponding discrete-time dynamical system might be

$$\begin{aligned} T_{t+1} &\equiv T_t + \lambda - d_T T_t - \beta V_t T_t \\ I_{t+1} &\equiv I_t + \beta V_t T_t \ - \delta I_t \\ V_{t+1} &\equiv V_t + p I_t - c V_t \end{aligned}$$

The first step will be to define a discrete logical mapping F_hiv_1 by using the differential equations from [1]

In [48]:
```
def F_hiv_1(x,t, lambda_l, d_t, beta, p, c):
    """
    Description:Logical map to find discrete values for time t+1 by usin
g time t values for T cells,
                Viral load and infected cells. I variable is implicitly
 calculated by using T values
                since they are complementary to each other.
    Input: Numpy array x with T and V variables and parameters of model
s.
    Output:The future values of T, and V stored in the Numpy array calle
d x_next.

    """
    # x = (T, V)
    x_next = x.copy ()

    ### BEGIN SOLUTION
    T, V = 0, 1
    I    = 1 - x[T] #we can always easily recover I
    x_next[T] = max(0, x[T] + (lambda_l - d_t*x[T]- beta*x[V]*x[T]))
    x_next[V] = max(0, x[V] + p*I - c*x[V])
    ### END SOLUTION

    return x_next
```

Now that we have a logical map we need to write a simulation function which steps the logical map forward in time. Below function returns the T and V values in discrete time steps for the selected HIV model.

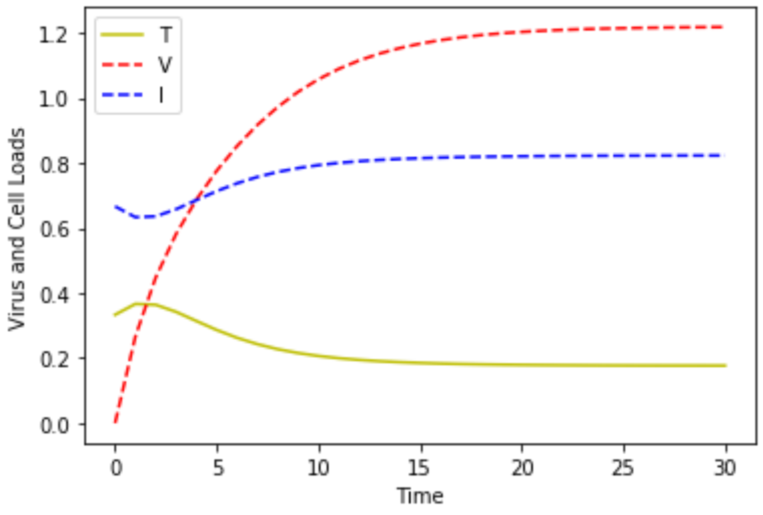
In [49]:
```
def sim(fun, t_max, x0, **fun_args):

    """
    Description:Simulating a discrete model.
    Input: fun representing the function for logical map,
           t_max for number of iterations of time,
           x0 as the initial values for state variables
           **fun_args for set of parameters that the logical map takes
    Output:2D Numpy array X representing simulation values. Rows are tim
e and columns are state variables.

    """
    X = np.zeros ((len(x0), t_max+1))


    X[:, 0] = np.array (x0) #initial conditions
    for t in range (t_max):
        X[:, t+1] = fun(X[:, t],t, **fun_args)

    return X
```

Now we will create a plotting function to visualize the results of our discrete simulation of the First HIV model described above.

In [50]:
```
def plot_sim_1 (X,alpha, t, d_t, beta,p, c):

    """
    Description:Plotting the discrete model.2D Plot of the simulation va
lues including
                T, V and I values on y axis versus time on x axis.
    Input:X as 2D numpy array to plot simulation values that contains th
e following:
               in X[0, :] Susceptible T-cells   T
               in X[1, :] Viral load            V
               in X[2, :] Infected cells        I
    Output:none

    """

    t_max = X.shape[1] - 1

    T = np.arange (t_max+1)
    use_points = len (T) <= 30
    plt.plot (T, X[0, :], 'ys--' if use_points else 'y-')
    plt.plot (T, X[1, :], 'r*--' if use_points else 'r--')
    plt.plot (T, 1. - X[0, :], 'bo--' if use_points else 'b--')
    plt.legend (['T', 'V', 'I'])
    plt.xlabel('Time')
    plt.ylabel('Virus and Cell Loads')
    #plt.axis ([0, t_max+1, 0, 1])
    #plt.title ("alpha = {}, tau = {}, kappa = {}".format (alpha, tau, k
appa))
```

Now we can set up some constants for our discrete First HIV model provided above and run our simulation with these values and observe the plot including the change of state variables which are T as T-cells which are susceptible cells for infection, V as viral load and I as infected cells .

In [51]:
```
T_MAX = 30

ALPHA = 1. / 3
LAMBDA_L= 0.1
D_T = 0.2
BETA = 0.3
P = 0.4
C = 0.27

# X[:, t] = [T_t, V_t]
x0 = np.array ([ALPHA, 0])

#create simulation data
X = sim (F_hiv_1,T_MAX, x0, lambda_l=LAMBDA_L, d_t=D_T, beta=BETA, p=P,
c=C)

#plot simulation data
plot_sim_1 (X, ALPHA, LAMBDA_L, D_T,BETA, P, C)
```



From the discrete simulation plot above, we observe that number of infected cells grow as the viral load grows over time. In accordance with this the number of susceptible cells which are T cells decreases over time. Before we continue our observations we will also implement a continous version of this simulation that is based on the ordinary differential equations solver from numpy.

Implementation for continuous time

Next, suppose we wish to treat time as a continuous, rather than discrete, variable. Doing so gives rise to a system of ordinary differential equations (ODEs):

$$\frac{d\vec{y}}{dt} = \frac{d}{dt}\begin{pmatrix} T(t) \\ I(t) \\ V(t) \end{pmatrix} = \begin{pmatrix} \lambda - d_T T(t) - \beta V(t) T(t) \\ \beta V(t) T(t) \; - \delta I(t) \\ p I(t) - c V(t) \end{pmatrix} \equiv \vec{F}(\vec{y}),$$

where $\vec{y}(t)$ is the state vector.

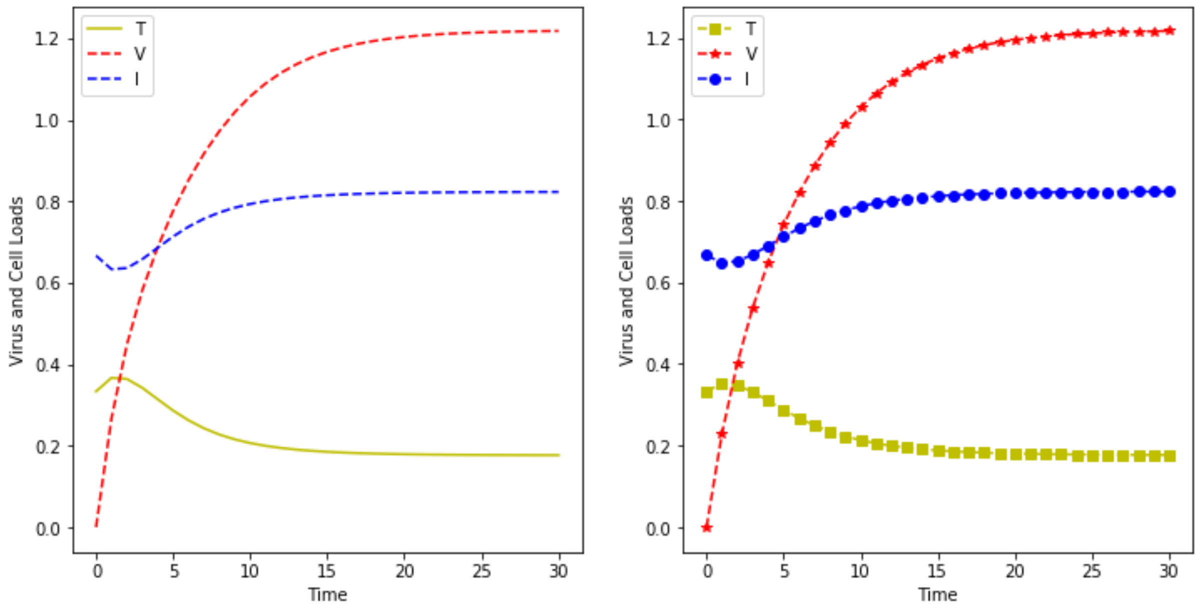
- Use the initial population parameters $T(0) = \alpha$, $I(0) = 1 - \alpha$, and $V(0) = 0$. These values are set in the `y0[:2]` array, below.
- α is the proportion of target cells that are alive.
- Store the results for $T(t)$, $I(t)$, and $V(t)$ for time points (i.e., including $t = 0$) in three NumPy arrays named `T_ode[:31]`, `V_ode[:31]`, and `I_ode[:31]`, respectively. The plotting code below assume these names.

Below code implements the ODE simulation for the First HIV model in continuous case.

In [52]:
```
# Initial populations, i.e., [T(0), V(0)]
y0 = np.array ([ALPHA, 0])

from scipy.integrate import odeint

def F_hiv_ode (y, t,lambda_l, d_t,beta, p, c):
    return F_hiv_1 (y,t,lambda_l, d_t,beta, p, c) - y

# Time points at which to compute the solutions:
time = np.arange (31).astype (float)

Y = np.zeros ((2, len (time)))
Y[:, 0] = y0[:2]

Y = odeint (F_hiv_ode,
            Y[:, 0],
            time,
            args=( LAMBDA_L, D_T, BETA, P, C)).T

T_ode = Y[0, :]
V_ode = Y[1, :]
I_ode = 1.0 - T_ode
```

In [53]:
```
def plot_sim_ode_1 (T, V, I, time):
    """
    Description: Plotting the continuous First HIV model state variable
s.
    Input:1D numpy arrays for state variables T, V, and I for First HIV
 Model and 1D array "time" for time steps
    Output: none

    """
    t_max = time[-1]
    use_points = len (time) <= 35
    plt.plot (time, T, 'ys--' if use_points else 'y-')
    plt.plot (time, V, 'r*--' if use_points else 'r--')
    plt.plot (time, 1. - T, 'bo--' if use_points else 'b--')
    plt.legend (['T', 'V', 'I'])
    plt.xlabel('Time')
    plt.ylabel('Virus and Cell Loads')
    #plt.axis ([0, t_max+1, 0, 1])
    #plt.title ("lambda_l = {}, d_t = {},beta={}, p={}, c={}".format(lam
bda_l, d_t,beta, p, c))
```

In [54]:
```
# Figure to compare discrete-time and continuous-time models
plt.figure (figsize=(12, 6))
plt.subplot (1, 2, 1)
plot_sim_1 (X, ALPHA, LAMBDA_L, D_T, BETA, P, C)
plt.subplot (1, 2, 2)
plot_sim_ode_1(T_ode, V_ode, I_ode, time)
```



Plot for discrete time model can be seen on the left hand-side and the continuous time model can be seen on right. We can observe from our simulation that T and I are complementary since a target cell will become and infected cell.

The other main observation is that once a steady state is reached, The virus will stay in the system forever. Our next model will look at what would happen if we start to give medicine to the patient in order to reduce the viral load V.

Second HIV model -- Modeling treatment

In paper[2], we can find a model with differential equations that lets use observe the effect of treatment on the viral load on a patient. This treatment helps in a way to cause the I cells to produce immature virus particles which are non-infectious and it can prevent the succesful infection of a cell as well as decreasing the virus level.To model this we will introduce new parameters, states and equations as follows:

- V_{It} be the fraction of the Virus that is infectious (discrete) time t;
- V_{NIt} be the fraction of the Virus that is non-infectious at t; and

where $V_{It} + V_{NIt} = 1$ since infectious Virus V_{It} and non-infectious Virus V_{NIt} are complementary we always only need to compute one in order to compute the other. Here, we will also implicitly assume that the number of individuals is large enough that we can treat these fractions as being continuous.

- ϵ_{RT} is a parameter between 0 and 1 that represents the effectiviness of the inhibitor that prevents the establishment of productive infection of a cell. $\epsilon_{RT} = 1$ implies 100% effective inhibitor.
- ϵ_{PI} is a parameter that represents the effectiveness of protease inhibitor which prevents the maturation of HIV virions into infectious particles.

Then, a corresponding discrete-time dynamical system might be

$$\begin{aligned} T_{t+1} &\equiv T_t + \lambda - d_T T_t - (1-\epsilon_{RT})\beta V_{It} T_t \\ I_{t+1} &\equiv I_t + (1-\epsilon_{RT})\beta V_{It} T_t \ - \delta I_t \\ V_{It+1} &\equiv V_{It} + (1-\epsilon_{PI}) p I_t - c V_{It} \\ V_{NIt+1} &\equiv V_{NIt} + \epsilon_{PI} p I_t - c V_{NIt} \end{aligned}$$

In [55]:
```
def F_hiv_2 (x,t,lambda_l, d_t,beta, p, c, eps_RT, eps_PI):

    """

    Description:Logical map for Second HIV Model to find discrete values
in time t+1 by using time t
                values for T cells, Viral load of infectious virus, vira
l load of non-infectious virus
                and infected cells. I variable is implicitly calculated
 by using T values since they are
                complementary to each other.
    Input: x as 1D numpy array that will contain
              in x[0] Susceptible T-cells         T
              in x[1] Infectious Viral load       V_i
    Output: 1D numpy array x_next that returns state variables T and V_i
values at time t+1


    """

    # x = (T, V_i)
    x_next = x.copy ()

    T, V_i = 0, 1
    I = 1 - x[T]
    x_next[T]   = max(0, x[T] + (lambda_l - d_t*x[T]-(1-eps_RT)*beta*x[V
_i]*x[T]))
    x_next[V_i] = max(0, x[V_i] + (1-eps_PI)*p*I - c*x[V_i])

    return x_next
```

In [56]:
```
def plot_sim_2 (X):

    """
    Description:Plotting the discrete model.2D Plot of the simulation va
lues including T, V and I values
                on y axis versus time on x axis.
    Input: X as 2D numpy array that will contain
                in X[0, :] Susceptible T-cells    T
                in X[1, :] Infectious Viral load  V_i
                in X[2, :] Infected cells         I
    Output:none

    """
    t_max = X.shape[1] - 1

    T = np.arange (t_max+1)
    use_points = len (T) <= 30
    plt.plot (T, X[0, :], 'ys--' if use_points else 'y-')
    plt.plot (T, X[1, :], 'r*--' if use_points else 'r--')
    plt.plot (T, 1. - X[0, :], 'bo--' if use_points else 'b--')
    plt.legend (['T', 'V_i', 'I'])
    plt.xlabel('Time')
    plt.ylabel('Virus and Cell Loads')
    #plt.axis ([0, t_max+1, 0, 1])
```

Implementation for continuous time

Next, suppose we wish to treat time as a continuous, rather than discrete, variable. Doing so gives rise to a system of ordinary differential equations (ODEs):

$$\frac{d\vec{y}}{dt} = \frac{d}{dt}\begin{pmatrix} T(t) \\ I(t) \\ V_I(t) \\ V_{NI}(t) \end{pmatrix} = \begin{pmatrix} \lambda - d_T T(t) - (1-\epsilon_{RT})\beta V_I(t) T(t) \\ (1-\epsilon_{RT})\beta V_I(t) T(t) \ - \delta I(t) \\ (1-\epsilon_{PI}) p I(t) - c V_I(t) \\ \epsilon_{PI}\, p I(t) - c V_{NI}(t) \end{pmatrix} \equiv \vec{F}(\vec{y}),$$

where $\vec{y}(t)$ is the state vector.

- Use the initial population parameters $T(0) = \alpha$ and $V_i(0) = 10$. These values are set in the `y0[:2]` array, below.
- α is the proportion of target cells that are alive.
- Store the results for $T(t)$ and $V_i(t)$ for time points (i.e., including $t = 0$) in two NumPy arrays named `T_ode[:31]` and `V_i_ode[:31]` respectively. The plotting code below assume these names.

Make plotting function for continous simulation of second HIV model where we introduce the treatment modeled in paper[2]:

In [57]:
```
def plot_sim_2_ode (T, V_i, I, T_):

    """
    Description: Plotting the continuous Second HIV model state variable
s.
    Input: T, V_i, and I as 1D numpy arrays for state variables in Secon
d HIV Model and 1D array T_ for time steps
    Output: none

    """

    t_max = T_[-1]
    use_points = len (T_) <= 35
    plt.plot (T_, T, 'ys--' if use_points else 'y-')
    plt.plot (T_, V_i, 'r*--' if use_points else 'r--')
    plt.plot (T_, 1. - T, 'bo--' if use_points else 'b--')
    plt.legend (['T', 'V_i', 'I'])
    plt.xlabel('Time')
    plt.ylabel('Virus and Cell Loads')
    #plt.axis ([0, t_max+1, 0, 1])
    #plt.title ("lambda_l = {}, d_t = {},beta={}, p={}, c={}, eps_RT={},
eps_PI={}".format(lambda_l, d_t,beta, p, c, eps_RT, eps_PI))
```

Prepare ODE function for continuous simulation of the Second HIV model:

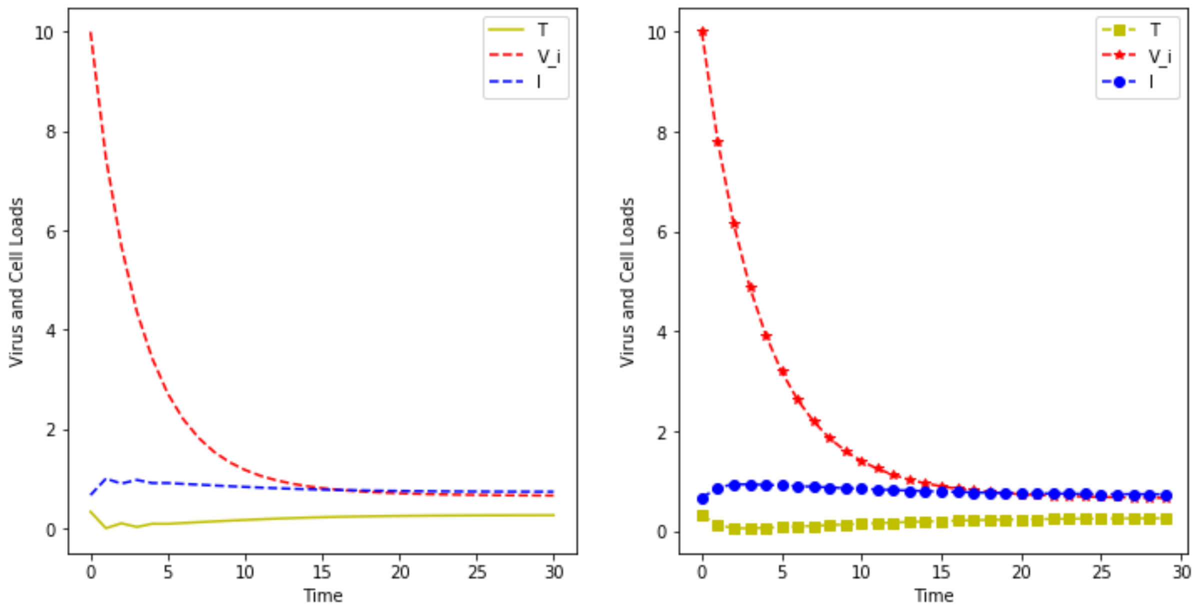
In [58]:
```
### BEGIN SOLUTION
from scipy.integrate import odeint

def F_hiv_2_ode (y, t, lambda_l, d_t,beta, p, c, eps_RT, eps_PI):
    return F_hiv_2 (y, t,lambda_l, d_t,beta, p, c, eps_RT, eps_PI) - y
```

Now we can set up some constants and run our simulations for the Second HIV model that includes the treatment mentioned in paper[2]:

In [59]:
```
ALPHA = 1. / 3
LAMBDA_L= 0.1
D_T = 0.2
BETA = 0.3
P = 0.4
C = 0.27
EPS_RT = 0.1
EPS_PI = 0.4

# X[:, t] = [T_t, V_i_t]
x0 = np.array ([ALPHA, 10])
```

Run both discrete and continous simulations and plot for the Second HIV model. Once we run them we can use the plotting functions to display the state variables over time.

In [60]:
```
#create discrete simulation data
X = sim (F_hiv_2, T_MAX, x0, lambda_l=LAMBDA_L, d_t=D_T, beta=BETA, p=P,
c=C, eps_RT = EPS_RT, eps_PI = EPS_PI )


time = np.arange (T_MAX).astype (float)
Y = np.zeros ((2, len (time)))
Y[:, 0] = x0[:2]
#create continous simulation data
Y = odeint (F_hiv_2_ode,
            Y[:, 0],
            time,
            args=(LAMBDA_L, D_T, BETA, P, C, EPS_RT, EPS_PI)).T


T_ode = Y[0, :]
V_i_ode = Y[1, :]
I_ode = 1.0 - T_ode
V_ni_ode = 1.0 - V_i_ode

#plot both simulations
plt.figure (figsize=(12, 6))
plt.subplot (1, 2, 1)
plot_sim_2(X)
plt.subplot (1, 2, 2)
plot_sim_2_ode(T_ode, V_i_ode, I_ode, time)
```



Plot for discrete time model can be seen on the left hand-side and the continuous time model can be seen on right. As we can observe, if we start from some viral load, the treatment will bring down the amount of virus down.

According to [2], in real case scenarios we would observe a tapering off in the speed at which the virus is cleared out. The paper refers to this as phase 1 and phase 2. There are a few ways to model this but we will do it by introducing the latent cell M in the third HIV model below.

Third HIV model -- Modeling treatment phases

In paper[2], we can find a model with differential equations that lets us observe the effect of treatment on the viral load on a patient also during phase 2 of viral load. Third HIV Model introduces productively infected cells I ,long-lived infected cells M^* and latently infected cells L which means that these cells don't produce virions until they get activated. To model this we will introduce new parameters, states and equations as follows:

- M_t^* be the fraction of the cells that are long-lived infected cells M^* at t;
- L_t be the fraction of the cells that are latent cells t;

The model we present below holds these:

- $V_{It} + V_{NIt} = 1$ since infectious Virus V_{It} and non-infectious Virus V_{NIt} are complementary we always only need to compute one in order to compute the other.
- Also $M_t^* + I_t + L_t = 1$ Here, we will also implicitly assume that the number of individuals is large enough that we can treat these fractions as being continuous.

Introduction of new parameters:

- τ_{RT} and τ_{PI} represents the pharmacological delay which takes into account that antiretroviral drugs are not instantly active and the delay values may be different for reverse transcriptase inhibitors and protease inhibitors
- T (non-infected susceptible cells) and M (long-lived cells) cells remain constant during the observation
- f_k is the parameter for the rate that L cells are produced
- δ_l is the constant rate that L cells die
- k is the constant rate that I cells are generated
- k_m is the constant rate that M^* cells are generated
- $N\delta$ is the average rate per cell that I cells produce virus
- p_m is the average rate per cell that M^* cells produce virus
- δ is the constant rate that I cells are lost
- μ is the constant rate that M^* cells are lost
- a is the constant rate that L cells are activated into productively infected cells
- c is the constant rate that both the infectious and non-infectious virions are cleared

Then, a corresponding discrete-time dynamical system is:

$$\begin{aligned}
I_{t+1} &\equiv I_t + (1-\epsilon_{RT} h(t-\tau_{RT}))\beta T V_{It} + aL_t - \delta I_t \\
M_{t+1}^* &\equiv M_t^* + (1-\epsilon_{RT} h(t-\tau_{RT}))k_M M V_{It} - \mu M_t^* \\
L_{t+1} &\equiv L_t + (1-\epsilon_{RT} h(t-\tau_{RT}))f_k T V_{It} - aL_t - \delta_L L_t \\
V_{It+1} &\equiv V_{It} + (1-\epsilon_{PI} h(t-\tau_{PI}))N\delta I_t + (1-\epsilon_{PI} h(t-\tau_{PI}))p_M M_t^* - cV_{It} \\
V_{NIt+1} &\equiv V_{NIt} + \epsilon_{PI} h(t-\tau_{RT})N\delta I_t + \epsilon_{PI} h(t-\tau_{PI})p_M M_t^* - cV_{NIt}
\end{aligned}$$

where h(t- τ) is a Heavyside function that takes 0 value for t< τ and 1 value for $t \geq \tau$.

In [61]:
```
def h(t,tau):
    """
    Description: Heavyside function used in the Third HIV model.
    Input: Integer variable t as time and integer constant tau.
    Output: 0 if t is less than tau and 1 if otherwise.
    """
    if(t<tau):
        a=0
    else:
        a=1
    return a
```

In [62]:
```
def F_hiv_3 (x, t, d_t, beta, c, eps_RT, eps_PI, a, beta_M, mu, N, p_M,
tau_RT, tau_PI, T, M, f_k,delta, delta_L):
    """
    Description: Third HIV model in discrete time.
    Input: x as 1D numpy array that will contain
                in x[0] Infected cells    I
                in x[1] Viral load        V_i
                in x[2] Long lived cells M*
                in x[3] Latent cells      L
    Output: 1D numpy array x_next that returns state variables I, M_sta
r, V_i and L values at time t+1
    """


    # x = (I, V_i, M_star, L)
    x_next = x.copy ()

    I, V_i, M_star, L = 0, 1, 2, 3
    x_next[I]       = max(0, x[I] + (1-eps_RT*h(t,tau_RT))*beta*T*x[V_i]+
a*x[L]- delta*x[I] )
    x_next[M_star] = max(0, x[M_star] + (1-eps_RT*h(t,tau_RT))*beta_M* M
*x[V_i]-mu*x[M_star])
    x_next[V_i]     = max(0, x[V_i] +(1-eps_PI*h(t,tau_PI))*N*delta*x[I]
+ (1-eps_PI*h(t,tau_PI))*p_M* x[M_star] - c*x[V_i])
    x_next[L]       = max(0, x[L] + (1-eps_RT*h(t,tau_RT))*f_k*T*x[V_i]-a
*L-delta_L*x[L])
    return x_next
```

Implementation for continuous time

Next, suppose we wish to treat time as a continuous, rather than discrete, variable. Doing so gives rise to a system of ordinary differential equations (ODEs):

$$\frac{d\vec{y}}{dt} = \frac{d}{dt}\begin{pmatrix} I(t) \\ M^*(t) \\ L(t) \\ V_I(t) \\ V_{NI}(t) \end{pmatrix} = \begin{pmatrix} (1-\epsilon_{RT}h(t-\tau_{RT}))\beta T V_I(t) + aL(t) - \delta I(t) \\ (1-\epsilon_{RT}h(t-\tau_{RT}))k_M M V_I(t) - \mu M^*(t) \\ (1-\epsilon_{RT}h(t-\tau_{RT}))f_k T V_I(t) - aL(t) - \delta_L L(t) \\ (1-\epsilon_{PI}h(t-\tau_{PI}))N\delta I(t) + (1-\epsilon_{PI}h(t-\tau_{PI}))p_M M^*(t) - cV_I(t) \\ \epsilon_{PI}h(t-\tau_{RT})N\delta I(t) + \epsilon_{PI}h(t-\tau_{PI})p_M M^*(t) - cV_{NI}(t) \end{pmatrix} \equiv \vec{F}(\vec{y}),$$

where $\vec{y}(t)$ is the state vector.

- Use the initial population parameters $I(0) = 1 - \alpha$, $V_i(0) = 10$, $M_{star}(0) = \alpha$, and $L(0) = 0.1$.
- α is the proportion of target cells that are alive.
- Store the results for $I_{ode}(t)$, $Vi_{ode}(t)$, $Mstar_{ode}(t)$ and L_{ode} for time points (i.e., including $t = 0$) in four NumPy arrays named `I_ode[:T_MAX]` , `V_i_ode[:T_MAX]` , `M_star_ode[:T_MAX] and L_ode[:T_MAX]` , respectively. The plotting code below assume these names.

In [63]:
```
def plot_sim_3 (X):

    """
    Description: Plotting the discrete model state variables.
    Input: X as 2D numpy array that will contain
                in X[0, :] Infected cells    I
                in X[1, :] Viral load        V_i
                in X[2, :] Long lived cells M*
                in X[3, :] Latent cells      L
    Output: none
    """

    t_max = X.shape[1] - 1

    T = np.arange (t_max+1)
    use_points = len (T) <= 30
    plt.plot (T, X[0, :], 'ys--' if use_points else 'y-')
    plt.plot (T, X[1, :], 'r*--' if use_points else 'r--')
    plt.plot (T, X[2, :], 'bo--' if use_points else 'b--')
    plt.plot (T, X[3, :], 'go--' if use_points else 'g--')
    plt.legend (['I', 'V_i', 'M*','L'])
    plt.xlabel('Time')
    plt.ylabel('Virus and Cell Loads')
    #X[:, t] = [I_t, V_i_t, M_t, L_t]
```

Prepare ODE for continous simulation of Third HIV Model:

In [64]:
```
#F_phase_2 (x, t, d_t, beta, c, eps_RT, eps_PI, a, beta_M, mu, N, p_M, t
au_RT, tau_PI)

#Initial populations, i.e., [I(0), V_i(0), M_star(0), L(0)]
y0 = np.array ([1.0 - ALPHA, 10 , ALPHA, .1 ])

# Time points at which to compute the solutions:
time = np.arange (100).astype (float)

### BEGIN SOLUTION
from scipy.integrate import odeint

def F_phase_2_ode (y, t, d_t, beta, c, eps_RT, eps_PI, a, beta_M, mu, N,
p_M, tau_RT, tau_PI, T, M, f_k, delta,delta_L): #bu t burdaydi
    return F_hiv_3  (y, t, d_t, beta, c, eps_RT, eps_PI, a, beta_M, mu,
N, p_M, tau_RT, tau_PI, T, M, f_k,delta, delta_L) - y
```

Setup plotting for continous case:

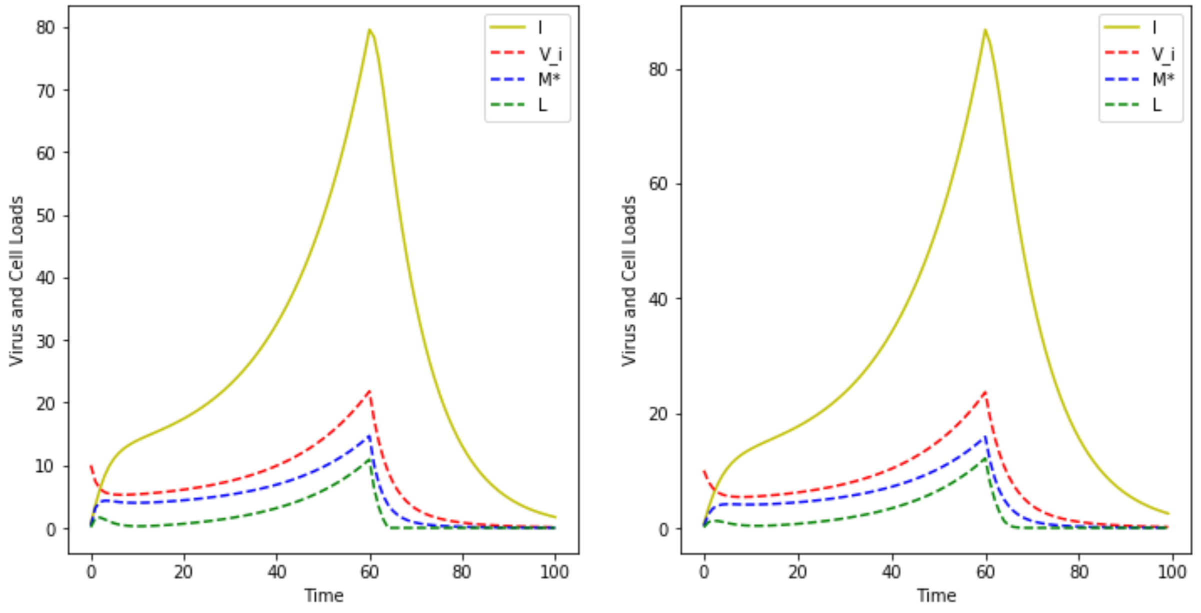
In [65]:
```
def plot_sim_ode (I, V_i, M_star, L, time):

    """
    Description:Plotting the continous model state variables.
    Input: I -> Numpy 1d array containing infected cell simulation value
s
         V_i -> Numpy 1d array containing viral load simulation values
      M_star -> Numpy 1d array containing long lived infected cells.
           L -> Numpy 1d array containing latent cell simulation values
        time -> Numpy 1d array containing time steps that need to be plo
tted
    Output: none
    """

    t_max = time[-1]
    use_points = len (time) <= 35
    plt.plot (time, I, 'ys--' if use_points else 'y-')
    plt.plot (time, V_i, 'r*--' if use_points else 'r--')
    plt.plot (time, M_star, 'bo--' if use_points else 'b--')
    plt.plot (time, L, 'go--' if use_points else 'g--')
    plt.legend (['I', 'V_i', 'M*', 'L'])
    plt.xlabel('Time')
    plt.ylabel('Virus and Cell Loads')
    #plt.axis ([0, t_max+1, 0, 1])
```

Now we can set up some simulation parameters such as running time and also we need to set the various model parameters. At the very bottom, we also intialize our state variables.

In [66]:
```
T_MAX = 100

ALPHA      = 1. / 3
LAMBDA_L   = 0.1
D_T        = 0.2
BETA       = 0.3
P          = 0.4
C          = 0.27
EPS_RT   = EPS_PI = 0.8
A        = 0.5
BETA_M   = 0.3
MU       = 0.4
N_C      = 0.5
P_M      = 0.2
TAU_RT   = TAU_PI = 60
T_C      = 1
M_C      = 1
F_K      = 0.3
DELTA    = 0.1
DELTA_L  = 0.4

T        = 1
M        = 1
f_k      = 0.3
delta    = 0.1
delta_L  = 0.4

#X[:, t] = [I_t, V_i_t, M_t, L_t]
x0 = np.array ([1.0 - ALPHA, 10, ALPHA, 0.1])
```

Now we are ready to run both discrete and continous simulations. Once we run them we can use the plotting functions to display the state variables over time.

```
In [67]: #create discrete simulation data
X = sim (F_hiv_3, T_MAX, x0, d_t=D_T, beta=BETA, c=C, eps_RT = EPS_RT, e
ps_PI = EPS_PI,
                               a=A, beta_M=BETA_M, mu=MU, N=N_C, p_M=P_M
, tau_RT=TAU_RT,
                              tau_PI=TAU_PI, T=T_C, M=M_C, f_k=F_K,delta
=DELTA, delta_L=DELTA_L)

Y = np.zeros ((4, len (time)))
Y[:, 0] = x0[:4]
#create continous simulation data
Y = odeint (F_phase_2_ode,
            Y[:, 0],
            time,
            args=(D_T, BETA, C, EPS_RT, EPS_PI, A, BETA_M, MU, N_C, P_M,
TAU_RT,
                              TAU_PI, T_C, M_C, F_K, DELTA, DELTA_L)).T


I_ode = Y[0, :]
V_i_ode = Y[1, :]
M_star_ode = Y[2, :]
L_ode = Y[3, :]

#plot both simulations
plt.figure (figsize=(12, 6))
plt.subplot (1, 2, 1)
plot_sim_3(X)
plt.subplot (1, 2, 2)
plot_sim_ode(I_ode, V_i_ode, M_star_ode, L_ode, time)
```



Plot for discrete time model can be seen on the left hand-side and the continuous time model can be seen on right. As can be observed from the plots once the drug therapy kicks in the level of plasma virus is predicted to decay. After the treatment productively infected cells I decay faster compared to the long lived infected cells M^*. Both from the plot and the paper [2] if these two populations of cells are assumed to be the only sources of virus, the second phase of decay extrapolates to zero residual infected cells in 2–3 years of completely suppressive antiretroviral therapy.

Part 3: HIV treatment using ODE simulation and reinforcement learning

Introduction

Discovering effective treatment strategies for HIV remains a significant challenge in medical research. To date, the clinically effective way to treat HIV is using a combination of anti-HIV drugs named as anitretrovirals to inhibit the development of drug resistant HIV strains. Anti-HIV drugs are currently grouped into two main categories: Reverse Transcriptase inhibitors(RTI) and Protease Inhibitors(PI). RTIs prevent HIV RNA from being converted into DNA which blocks the virus replication process initiated in the infected cell. PIs work at the final stage of viral replication and attempt to prevent HIV from making new copies of itself by interfering with the HIV protease enzyme. This prevents new copies of HIV from infecting new cells.

Although the combination of these drugs reduce and maintain the viral loads below the detection limit, their long term use can lead to complications and patients often experience side-effects thus leading to poor compliance. Effective drug scheduling strategies have been proposed to address this concern. The goal of drug-scheduling strategy is to bring the immune system into a state that allows it to independently maintain immune control over the virus. Also, transfer to a drug-independent viral control situation needs to be done with as low systemic effects as possible.

Structured treatment interruption (STI) is one such strategy which has received a lot of attention. In STI, the patient is cycled on and off drug therapy. Since STI involves periods of relief from treatment, it is well received by the patients. When the treatment is interrupted, viral load increases to a high level which leads to activating adaptive immune response. Repeated STI simulations has been observed to maintain immune control over the virus in the absence of treatment.

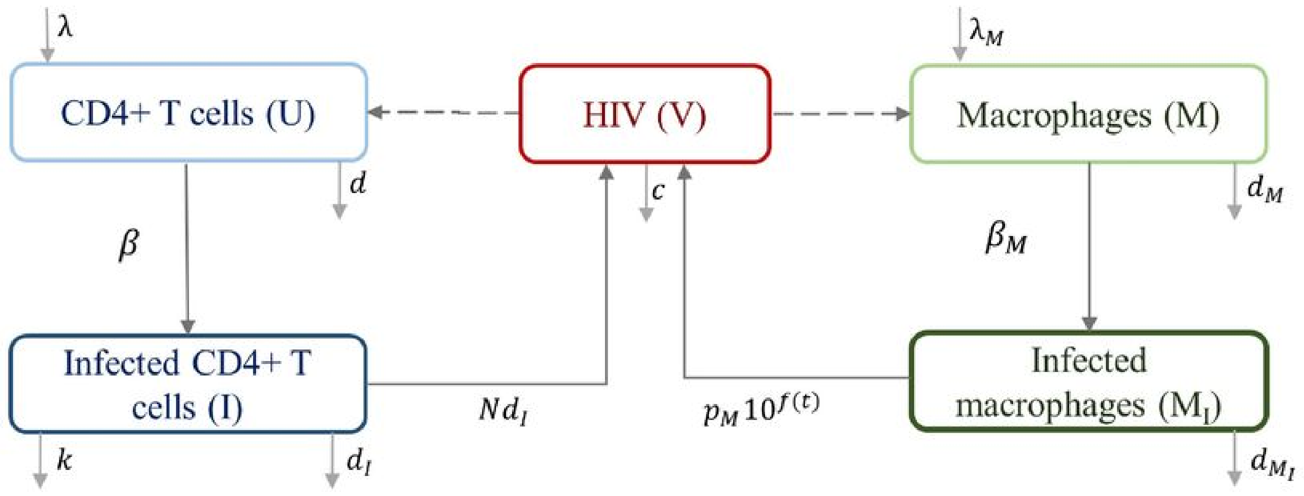
Background

Previous studies have explored uzing mathematical models of HIV infection dynamics for addressing the problem of designing STI treatments. These models are usually represented by a set of Ordinary Differential Equations(ODEs) and control theory is applied to deduce STI strategies. Modeling the HIV infection dynamics is a complex task and along with selecting the right parametric system of ODEs, one must fit their parameters to reflect quantitatively biological observations. Two main approaches have been proposed:

1. Control theory based studies first state an optimality criterion and then search for control strategies optimizing this criterion.
2. Reinforcement Learning(RL) computes control strategy directly from the measured trajectories and does not need the apriori identification of model of system dynamics.

In this project, we investigate the feasibility of using RL to determine optimal dosing strategy for clinical data. We use simulation to artificially generate the HIV clinical data. This is because of limited availability of publicly available HIV datasets.

Conceptual Model Diagram



3.1 HIV Simulation Model

Exercise 1 : (20 points) Design and implement a continous time model system for simulating the dynamics of viral load and infected cells under STI strategy. The model should take into consideration that the patient characteristics can change and allow adjustment of drug combinations (RTI and PI) to study the impact on patient's condition and viral dynamics.

Exercise 1.1: Write an ordinary differential equation to model the system.

We use the mathematical model proposed by Adams et. al.[1]. The model is a continuous ODE formulation. Although modeling HIV infection requires taking into consideration multiple factors, we can choose a small subset of these factors to keep our model simple. The proposed model includes the following patient wellness indicators, which adequately describe patient's condition (state) at a particular time:

1. $T1$: Infected CD4+ cells
2. $T1^*$: Non-infected CD4+ cells
3. $T2$: Infected macrophages
4. $T2^*$: Non-infected macrophages
5. V : HIV Viral Load (RNA copies per ml of blood)
6. E : Immune effector CD8+ cells which measure the body's immune response to the presence of infected T-cells

The model should also include the action of commonly used antiretrovirals and allow using a combination of RTI and PI drugs which are major classes of drugs used for HIV treatment. We define drug efficacy parameters ϵ_1 and ϵ_2 for this reason. ϵ_1 models a reverse transcriptase(RT) inhibitor and is more effective in maintaining population of CD4+ cells $(T1)$ while ϵ_2 models the PT inhibitor. The efficacy of the drug is controlled using $f \in [0, 1]$ and $f * \epsilon$ defines the overall impact of the drug.

The populations of uninfected $T1$ and $T2$ cells have different birth rates (λ_i) and death rates (d_i). A complete description of the model along with the parameters is included below:

Model Equations

These equations describe the complete dynamics of the state **s** = $[T1, T2, T1^*, T2^*, V, E]$ of the model:

$$\frac{d\vec{s}}{dt} = \frac{d}{dt}\begin{pmatrix} T_1(t) \\ T_2(t) \\ T_1^*(t) \\ T_2^*(t) \\ V(t) \\ E(t) \end{pmatrix} = \begin{pmatrix} \lambda_1 - d_1 T_1(t) - (1-\epsilon_1)k_1 V(t) T_1(t) \\ \lambda_2 - d_2 T_2(t) - (1 - f * \epsilon_1)k_2 V(t) T_2(t) \\ (1-\epsilon_1)k_1 V(t) T_1(t) - \delta T_1^*(t) - m_1 E(t) T_1^*(t) \\ (1 - f\epsilon_1)k_2 V(t) T_2(t) - \delta T_2^*(t) - m_2 E(t) T_2^*(t) \\ (1-\epsilon_2) N_T \delta [T_1^*(t) + T_2^*(t)] - cV(t) - [(1-\epsilon_1)\rho_1 k_1 T_1(t) + (1 - f\epsilon_1)\rho_2 k_2 T_2(t))] V(t) \\ \lambda_E + \frac{b_E(T_1^*(t)+T_2^*(t))}{T_1^*(t)+T_2^*(t)+K_b} E - \frac{d_E(T_1^*(t)+T_2^*(t))}{T_1^*(t)+T_2^*(t)+K_d} E - \delta_E E \end{pmatrix} \equiv \vec{F}(\vec{s}),$$

Model Parameters

Parameters	Value of Parameters	Description
λ_1	10000	production rate of CD4+ cells
d_1	0.01	death rate of CD4+ cells
ϵ_1	$[0, 1)$	efficacy of RTI
ϵ_2	$[0, 1)$	efficacy of PI
k_1	$8.0 * 10^{-7}$	infection rate of CD4+ cells
λ_2	31.98	production rate of macrophages
d_2	0.01	death rate of macrophages
f	0.34	reduction of treatment efficacy for macrophages
k_2	$1.0 * 10^{-4}$	infection rate of macrophages
δ	0.7	death rate of infected cell
m_1	$1.0 * 10^{-5}$	immune-induced clearance rate for CD4+ cells
m_2	$1.0 * 10^{-5}$	immune-induced clearance rate for macrophages
N_T	100	virions produced per infected cell
c	13	natural death rate of virus

Parameters	Value of Parameters	Description
ρ_1	1	average number of virions infecting a CD4+ cell
ρ_2	1	average number of virions infecting a macrophage
Immune effector parameters		
λ_E	1	production rate of immune effector/cytotix T-cell
b_E	0.3	maximum birth rate for cytotoxic T-cell
K_b	100	saturation constant for cytotoxic T-cell birth
d_E	0.25	maximum death rate for cytotoxic T-cell
K_d	500	saturation constant for cytotoxic T-cell death
$delta_E$	0.1	natural death rate of cytotoxic T-cells

In [1772]:
```
%matplotlib inline
%load_ext autoreload
%autoreload 2

import numpy as np
import jdc
from scipy.integrate import odeint, ode
from IPython.display import clear_output
from matplotlib import pyplot as plt
import collections
import seaborn as sns
import pandas as pd
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Step 1

We define our ODE model equations here to compute $\frac{d\vec{s}}{dt}$. The function returns the derivative ds. We also pass the drug efficacy parameter (f) along with the efficacy parameters ϵ_1 and ϵ_2. Since the model parameters defined above can vary between individuals, we pass an additional list variable named *params* which allow us to simulate the impact of better immune system on the overall dynamics of HIV virus.

In [1773]:
```
def derivs_dt(s,t=0,eps1=0,eps2=0,f=0.34,params=None):
    t1,t2,t11,t21,v,e = s
    if params is None:
        lambda1 = 1e4
        lambda2 = 31.98
        d1 = 0.01
        d2 = 0.01
        f = f
        k1 = 8e-7
        k2 = 1e-4
        delta = .7

        NT = 100.
        c = 13.
        rho1 = 1.
        rho2 = 1.
        deltaE = 0.1
        lambdaE = 1
        m1 = 1e-5
        m2 = 1e-5
        bE = 0.3
        Kb = 100
        d_E = 0.25
        Kd = 500
    else:
        lambda1,lambda2,d1,d2,k1,k2,delta,NT,c,rho1,rho2,deltaE,lambdaE,m1,m2,bE,Kb,d_E,Kd = params

    ds = s.copy()

    tmp1 = (1-eps1) * k1 * v * t1
    tmp2 = (1-f*eps1) * k2 * v * t2
    ds[0] = lambda1 - d1 * t1 - tmp1
    ds[1] = lambda2 - d2 * t2 - tmp2
    ds[2] = tmp1 - delta * t11 - m1 * e * t11
    ds[3] = tmp2 - delta * t21 - m2 * e * t21
    ds[4] = (1-eps2) * NT * delta * (t11 + t21) - c * v - ((1. - eps1) * rho1 * k1 * t1 + (1. - f * eps1)
    ds[5] = lambdaE + bE * (t11 + t21) / (t11 + t21 + Kb) * e - d_E * (t11 + t21) / (t11 + t21 + Kd) * e
    return ds
```

Exercise 1.2(5 points) Find fixed points of the system (without treatment) and perform their stability analysis.

At steady state, $\frac{d\vec{s}}{dt} = 0$. However, since the equations are slightly complicated here, we utilize the *fsolve* function to solve for steady state. We assume the standard model parameters and drug efficacy (ϵ_1, ϵ_2) is set to zero. We also consider states with positive state variables.

In [1779]:
```python
from scipy.optimize import fsolve
x02 = []
for i in range(5000):
    x = np.random.uniform(0,100000,6)
    x_temp = fsolve(derivs_dt, x)
    x_temp_sum = np.sum(np.abs(derivs_dt(x_temp)))

    if (x_temp>=0).all() and x_temp_sum < 1e-7:
        x02.append(np.round(x_temp))
x_final = np.unique(x02,axis=0)
print("Fixed points of the system")
for x in x_final:
    print(x)
```

```
/home/achoudhary/anaconda3/lib/python3.7/site-packages/scipy/optimize/minpack.py:162: RuntimeWarning: Th
e iteration is not making good progress, as measured by the
  improvement from the last five Jacobian evaluations.
  warnings.warn(msg, RuntimeWarning)
/home/achoudhary/anaconda3/lib/python3.7/site-packages/scipy/optimize/minpack.py:162: RuntimeWarning: Th
e iteration is not making good progress, as measured by the
  improvement from the last ten iterations.
  warnings.warn(msg, RuntimeWarning)

Fixed points of the system
[163573.      5.  11945.     46.  63919.     24.]
[664938.     50.   1207.     11.   6299. 207658.]
[967839.    621.     76.      6.    415. 353108.]
[1000000.    3198.       0.       0.       0.      10.]
```

Adams et. al. highlight that when both ϵ_1 and ϵ_2 are zero, the dynamic model achieves four physical equilibrium points with all state variables being non-negative.

1. Uninfected individual - $(T_1, T_2, T_1^*, T_2^*, V, E)$ = $(10000000, 3, 198, 0, 0, 0, 10)$
2. Infected individual - $(T_1, T_2, T_1^*, T_2^*, V, E)$ = $(664938, 50, 1207, 11, 6299, 207658)$
3. Infected individual - $(T_1, T_2, T_1^*, T_2^*, V, E)$ = $(967839, 621, 76, 6, 415, 353108)$
4. Infected individual - $(T_1, T_2, T_1^*, T_2^*, V, E)$ = $(163573, 5, 111945, 46, 63919, 24)$

State 3 corresponds to an individual with good immune control over the virus while state 4 represents an individual in unhealthy state whose viral load is considerably elevated and T-cells are in short-supply in absence of treatment.

Analyze stability of fixed points

To analyze the stability of fixed points, we compute the eigenvalues of Jacobian matrix given by

$$\frac{d\vec{s}}{dt} = \mathbf{J}\vec{s}$$

$$\mathbf{J} = \begin{bmatrix} -d_1 - k_1 V & 0 & 0 & 0 & -k_1 T_1 & 0 \\ 0 & -d_2 - k_2 V & 0 & 0 & -k_2 T_2 & 0 \\ k_1 V & 0 & -\delta - m_1 E & 0 & k_1 T_1 & -m_1 T_1^* \\ 0 & k_2 * V & 0 & -\delta - m_2 E & k_2 T_2 & -m_2 T_2^* \\ -\rho_1 k_1 V & -\rho_2 k_2 V & \delta N_T & \delta N_T & -c - \rho_1 k_1 T_1 - \rho_2 k_2 T_2 & 0 \\ 0 & 0 & J_{6,3} & J_{6,4} & 0 & J_{6,6} \end{bmatrix}$$

$$J_{6,3} = J_{6,4} = \frac{b_E K b E}{(T_1^* + T_2^* + K_b)^2} - \frac{d_E K_d E}{(T_1^* + T_2^* + K_d)^2}$$

$$J_{6,6} = (\frac{b_E}{T_1^* + T_2^* + K_b} - \frac{d_E}{T_1^* + T_2^* + K_d})(T_1^* + T_2^*) - \delta_E$$

```
In [1775]: np.set_printoptions(suppress=True)
def jacob(x):
    [t1,t2,t11,t21,v,e] = x
    a63 = bE * Kb*e/ (t11 + t21 + Kb)**2 - d_E * Kd*e / (t11 + t21 + Kd)**2
    a64 = bE * Kb*e/ (t11 + t21 + Kb)**2 - d_E * Kd*e / (t11 + t21 + Kd)**2
    a66 = (bE/(t11+t21+Kb) - d_E/(t11+t21+Kd))*(t11+t21) - deltaE
    J = [[-d1-k1*v, 0, 0, 0, -k1*t1, 0],
     [0,-d2-k2*v, 0, 0, -k2*t2, 0],
     [k1*v,0, -delta-m1*e, 0, k1*t1, -m1*t11],
     [0,k2*v, 0, -delta-m2*e, k2*t2, -m2*t21],
     [-rho1*k1*v,-rho2*k2*v, delta*NT,delta*NT , -c-rho1*k1*t1-rho2*k2*t2, 0],
     [0,0,a63,a63,0,a66]
     ]
    return J
for i in range(x_final.shape[0]):
    max_eigenval = np.linalg.eigvals(jacob(x_final[i,:]))
    max2 = np.argsort(max_eigenval)
    eigen1,eigen2 = max_eigenval[max2[-2:]][::-1]

    if eigen1.real < 0:
        if eigen1.imag > 0:
            print("Point {} is stable with spiral focus".format(x_final[i,:]))
        else:
            print("Point {} is stable".format(x_final[i,:]))
    else:
        if eigen1.imag > 0:
                print("Point {} is unstable with spiral focus".format(x_final[i,:]))
        else:
            if eigen2.real<0:
                print("Point {} is unstable with saddle point".format(x_final[i,:]))
            else:
                print("Point {} is unstable".format(x_final[i,:]))
```

```
Point [163573.      5.  11945.     46.  63919.     24.] is stable with spiral focus
Point [664938.     50.   1207.     11.   6299. 207658.] is unstable with saddle point
Point [967839.    621.     76.      6.    415. 353108.] is stable with spiral focus
Point [1000000.    3198.       0.       0.       0.      10.] is unstable with saddle point
```

Exercise 1.3(5 points) Write the necessary functions to simulate a continuous time model for HIV infection considering patients with different immunities.

Step 1

We define a class for our simulator (HIVSimulator) which initializes the parameters for individual's immunity. We also initialize the action parameters (ϵ_1, ϵ_2) and the function for reseting the initial state of the simulator. Also, we include an option to randomize the initial state using slight perturbations

In [1776]:
```python
class HIVSimulator():
    def __init__(self, immunity_type):
        # immunity of the individual
        if immunity_type == 'strong':
            self.params = (1e4,31.98,0.01,0.01,8e-7,1e-4,0.7,100.,13.,1.0,1.0,0.1,100.0,1e-5,1e-5,0.5,500
        else:
            self.params = (1e4,31.98,0.01,0.01,8e-7,1e-4,0.7,100.,13.,1.0,1.0,0.1,1.0,1e-5,1e-5,0.3,100,0

        self.state = []

    def simulate(self,eps1,eps2,t,derivs):
        deriv_args = (eps1,eps2,0.34,self.params)
        #solving the ode using odeint
        sol = odeint(derivs, self.state, t, args=deriv_args)
        return sol

    def reset(self, state_type, randomize):
        """Reset the environment."""
        self.t = 0
        if state_type == 'low':
            self.state = [1000000., 3198., 0., 0., 1., 10.]
        elif state_type == 'high':
            self.state = [163573., 5., 11945., 46., 63919., 24.]
        elif state_type == 'early':
            self.state = [1000000., 3198., 1e-4, 1e-4, 1., 10.]
        if randomize:
            self.state = self.state + (self.state * np.random.uniform(-0.1,0.1,6))
        return self.state
```

Step 2

Define the visualizer for plotting the simulation output (cell counts and viral counts)

In [1777]:
```python
def visualize_plot(t,data_dict,plot_phase=True):
    if plot_phase:
        f, ((ax1, ax2), (ax3, ax4), (ax5,ax6), (ax7,ax8)) = plt.subplots(4, 2, figsize=(15,15))
    else:
        f, ((ax1, ax2), (ax3, ax4), (ax5,ax6)) = plt.subplots(3, 2, figsize=(15,10))
    axs = [ax1,ax2,ax3,ax4,ax5,ax6]
    ylabels = ['T1','T2','T1*','T2*','V','E']
    for k,data in data_dict.items():
        for i in range(6):
            axs[i].plot(t,data[:,i],label = k)
            axs[i].set_yscale('log')
            axs[i].set_xlabel('t')
            axs[i].set_ylabel(ylabels[i])
            axs[i].legend(loc="upper right")

        if plot_phase:
            ax7.plot(data[:,4],data[:,5])
            ax7.set_yscale('log')
            ax7.set_xscale('log')
            ax7.set_title('Phase Plot (E vs V)')
            ax7.set_xlabel('V')
            ax7.set_ylabel('E')
            ax7.set_label(k)

            ax8.plot(data[:,0],data[:,1])
            ax8.set_yscale('log')
            ax8.set_xscale('log')
            ax8.set_title('Phase Plot (T2 vs T1)')
            ax8.set_xlabel('T1')
            ax8.set_ylabel('T2')
            ax8.set_label(k)
```

Step 3

We simulate our model and verify whether it reaches the physical equilibria highlighted above. To initiate the simulation, we consider an individual in healthy state and introduce 1 viral copy per ml (V = 1c/ml), this is defined by state type low in *reset_()* function.

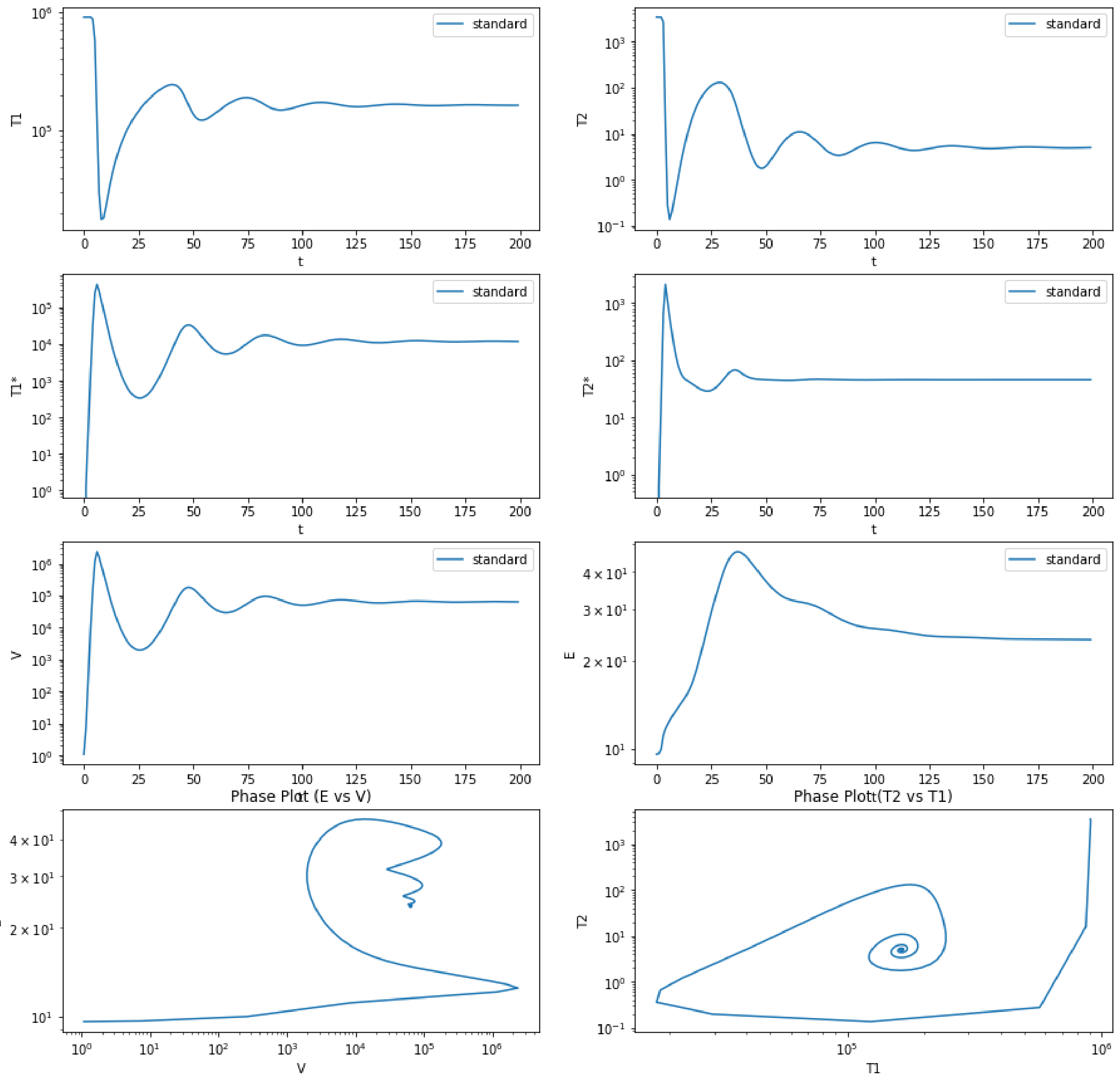
Individual with standard immune system

Infect a healthy individual with low viral count and observe the dynamics over t=200 days. Here we assume that no drug is being administered to the patient (ϵ_1, ϵ_2 = 0) and the patient has a immune system characterized by standard parameters

$$(\lambda_E, m_1, m_2, b_E, K_b, d_E, K_d) = (1.0, 1e{-}5, 1e{-}5, 0.3, 100, 0.25, 500)$$

```
In [1778]: h = HIVSimulator('standard')
           h.reset('low',True)
           dt = 1
           max_time = 200
           eps1, eps2 = 0,0
           t = list(range(0,max_time,dt))
           sol = h.simulate(eps1,eps2,t,derivs_dt)
           sol_dict ={'standard':sol}

           #visualize the state variables dynamics
           visualize_plot(t,sol_dict)
           #print the final state
           print("Equilibria State (T1,T2,T1*,T2*,V,E) = ", np.round(sol[-1,:]))
```

```
Equilibria State (T1,T2,T1*,T2*,V,E) =  [163086.      5.  11889.     46.  63631.     24.]
```



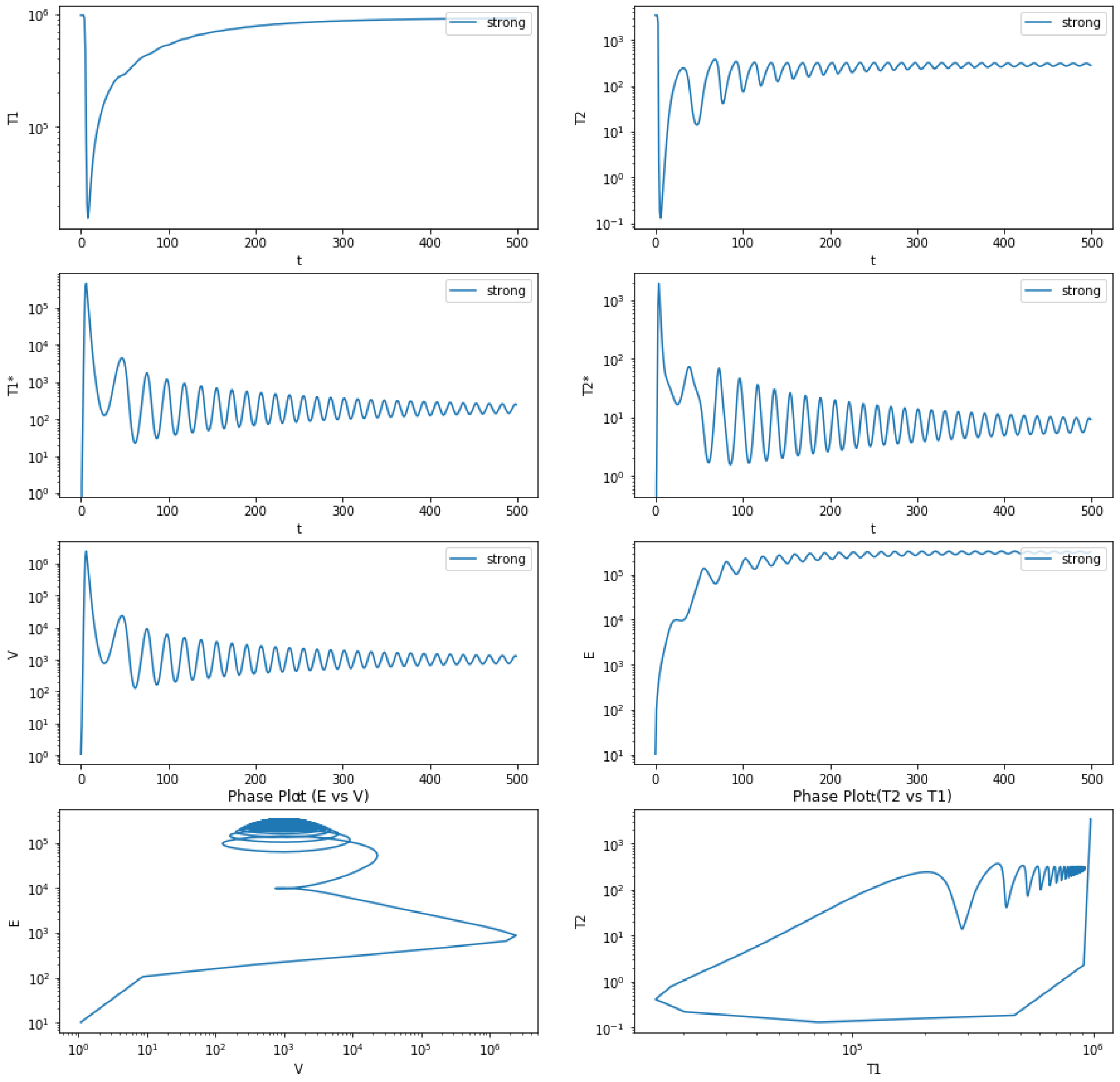
Individual with stronger immune system

Now, we consider an individual with stronger immune system (higher T-cell birth rates and saturation constant) Infect the individual with low viral count and observe the dynamics over t = 500 days. Here we assume that no drug is being administered to the patient (ϵ_1 , ϵ_2 = 0)

Immune effector parameters $(\lambda_E, m_1, m_2, b_E, K_b, d_E, K_d) = (100.0, 1e^{-5}, 1e^{-5}, 0.6, 500, 0.25, 500)$

```
In [1780]: h = HIVSimulator('strong')
           h.reset('low',True)
           dt = 1
           max_time = 500
           eps1, eps2 = 0,0
           t = list(range(0,max_time,dt))
           sol = h.simulate(eps1,eps2,t,derivs_dt)

           #visualize the state variables dynamics
           visualize_plot(t,{'strong':sol})
           #print the final state
           print("Equilibria State (T1,T2,T1*,T2*,V,E) = ", np.round(sol[-1,:]))
```

```
Equilibria State (T1,T2,T1*,T2*,V,E) =  [917828.    277.    242.      9.   1280. 323822.]
```

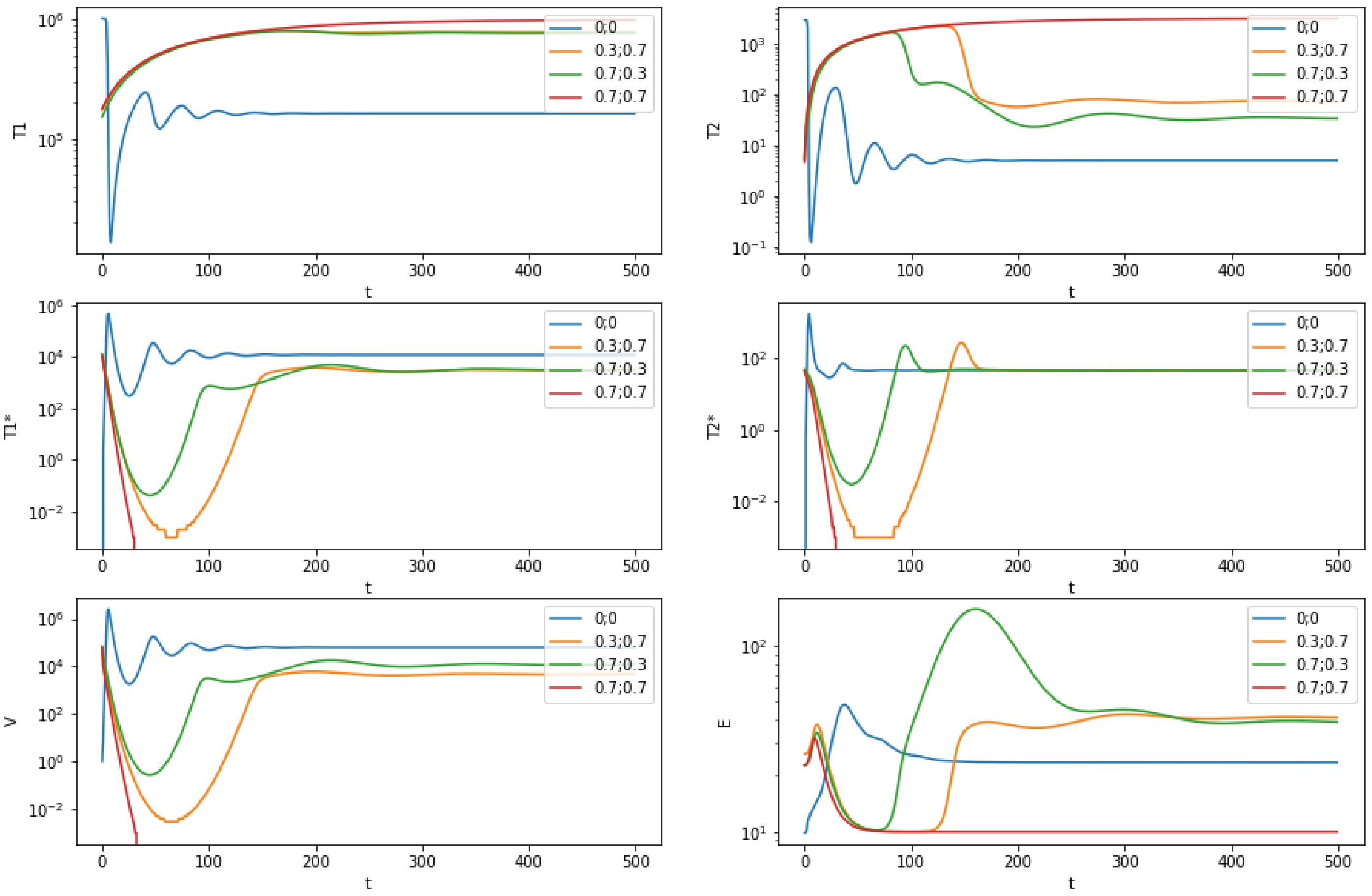


As observed, in both cases, the unstable steady state (healthy individual) transitions to stable steady state. The steady state for an individual with stronger immune system is much closer to fixed state 4 (described earlier) and his body is able to maintain a lower viral count without any treatment. Individual with standard immmune system transitions from healthy fixed state(1) to unhealthy fixed state(3) with high viral load and depleted immunity cells. Also, as evident from the phase plots, the stable fixed points exhibit a spiral behaviour.

Exercise 1.4(5 points) Simulate the effect of different drug combinations using different values for RT inhibitor and PT inhibitor efficacies (ϵ_1, ϵ_2)

As suggested by Adams *et* al., we consider 4 drug combinations: (ϵ_1, ϵ_2) = (0,0);(0.3,0.7);(0.7,0.3);(0.7,0.7)

In [1783]:
```
h = HIVSimulator('standard')
h.reset('high',True)
dt = 1
max_time = 500
eps = [[0,0],[0.3,0.7],[0.7,0.3],[0.7,0.7]]
sols = {}
t = list(range(0,max_time,dt))
for eps1,eps2 in eps:
    label = '{};{}'.format(eps1,eps2)
    if eps1 > 0:
        h.reset('high',True)
    else:
        h.reset('low',True)
    sols[label] = np.round(h.simulate(eps1,eps2,t,derivs_dt),3)

t = list(range(0,max_time,dt))
visualize_plot(t,sols,False)
```



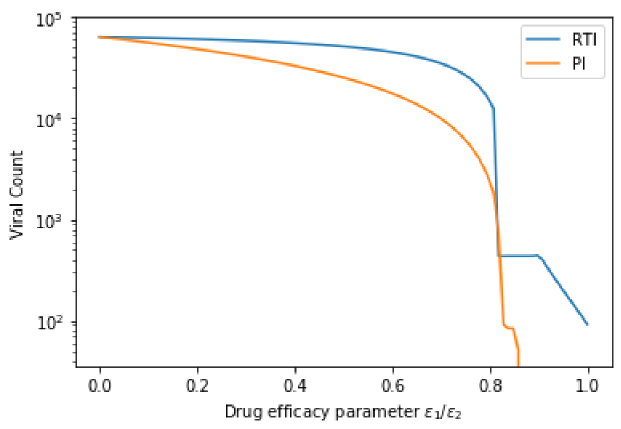
The simulation shows that treatment using high dosage of both drugs leads to much lower steady state viral load and healthy CD4 and macrophages count. Now we analyze the effect of varying RTI and PI inhibitors individually.

Evaluate effect of varying the RT inhibitor treatment efficacy (ϵ_1)

We vary the RT/P inhibitor parameter value from 0 to 1 individually and observe the final viral load at equilibria. Our initial state is now an individual with high initial viral load state (steady state 4 equilibria highligted above). We do not include the effect of other drug while varying a particular drug.

In [1784]:
```
h = HIVSimulator('standard')
efficacy = np.linspace(0,1,100)
viral_load_rt = []
viral_load_pt = []
max_time = 1000
dt = 1
t = list(range(0,max_time,dt))
for i in efficacy:
    h.reset('high',True)
    sol = h.simulate(i,0,t,derivs_dt)
    viral_load_rt.append(np.round(sol[-1,4]))
    h.reset('high',True)
    sol = h.simulate(0,i,t,derivs_dt)
    viral_load_pt.append(np.round(sol[-1,4]))

fig, ax = plt.subplots(1,1)
plt.plot(efficacy,np.array(viral_load_rt), label='RTI')
plt.plot(efficacy,np.array(viral_load_pt), label='PI')
plt.yscale('log',basey=10)
plt.yticks([100,1000,10000,100000])
plt.ylabel("Viral Count")
plt.xlabel("Drug efficacy parameter $\epsilon_1/\epsilon_2$")
plt.legend(loc="upper right")
plt.show()
```

```
/home/achoudhary/anaconda3/lib/python3.7/site-packages/scipy/integrate/odepack.py:248: ODEintWarning: Ex
cess work done on this call (perhaps wrong Dfun type). Run with full_output = 1 to get quantitative info
rmation.
  warnings.warn(warning_msg, ODEintWarning)
```



Increasing the drug efficacy beyond 0.8 leads to sudden drop in viral count. In case of PI, the viral count falls below the clinically detectable level of 43. Hence, now we focus on optimal drug dosage strategy with maximal range decided basis the computed curve above.

3.2 Determining ideal drug dosage for patient infected with high viral load

We saw that drug combination helps reduce and maintain viral load. However, their long term use can lead to complications and patients often experience side-effects which leads to poor compliance. Hence, we consider two drug scheduling strategies which essentially tries to vary drug efficacies (ϵ) over time and maximize a reward function(objective). We assume that we control dosage by controlling the efficacy parameter. The most common cost function used in various studies is:

$$J(\epsilon_1, \epsilon_2) = E_t[QV(t) + R_1\epsilon_1^2 + R_2\epsilon_1^2 - SE(t)]$$

where Q, R1, R2 and S are weight constants for the virus, controls inputs, and immune effectors, respectively. V and E are viral load and immune effector cell population. The objective is to mimimize the cost function, i.e. minimize the systemic costs of drug treatment and viral load while encouraging higher immunity cells population.

Optimal Control

Exercise 1.5(10 points) Develop a drug dosage control algorithm using Optimal Control method.

Using the control model proposed by Adams et. al.[1] we determine the optimal dosage strategy. We attempt to control HIV populations in finite time intervals using a control function $\epsilon(t)$ which represents the drug efficacy satisfying $0 <= a <= \epsilon(t) <= b < 1$. Here $\epsilon(t) = b$ represents maximum efficacy. We use the forward and backward integration along with optimal control parameter equation. We consider a patient with low viral load as considered by Adams et. al.[1] and only consider the scenario wherein RTI is administered i.e. we keep PI dosage to be zero (ϵ_2 = 0) & consider the simpler cost function used by Adams et. al. [1]:

$$J(\epsilon_1, \epsilon_2) = E_t[QV(t) + R_1\epsilon_1^2]$$

Step 1

We simulate the process on forward directions using our earlier simulation model and assuming a random value for ϵ_1 to begin with. Using the final state at t=200, we introduce adjoint variables and perform backward integration to reach t = 0 and determine the ideal control parameter for this iteration. We keep iterating until the control parameter value stabilizes.

Writing the equations for backward integration

```
In [1785]: def derivs_dt_inv(s,t,state_1,Q,eps1=0,eps2=0,f=0.34,params=None):
               e1,e2,e3,e4,e5,e6 = s
               t1,t2,t11,t21,v,e = state_1
               if params is None:
                   lambda1 = 1e4
                   lambda2 = 31.98
                   d1 = 0.01
                   d2 = 0.01
                   f = f
                   k1 = 8e-7
                   k2 = 1e-4
                   delta = .7

                   NT = 100.
                   c = 13.
                   rho1 = 1.
                   rho2 = 1.
                   deltaE = 0.1
                   lambdaE = 1
                   m1 = 1e-5
                   m2 = 1e-5
                   bE = 0.3
                   Kb = 100
                   d_E = 0.25
                   Kd = 500
               else:
                   lambda1,lambda2,d1,d2,k1,k2,delta,NT,c,rho1,rho2,deltaE,lambdaE,m1,m2,bE,Kb,d_E,Kd = params

               ds_inv = s.copy()
               tmp1 = bE*e*Kb/(t11+t21+Kb)**2 - d_E*e*Kd/(t11+t21+Kb)**2
               tmp2 = e5*NT*delta + e6*tmp1
               ds_inv[0] = -(e1*(-d1 - (1-eps1)*k1*v) + e3*(1-eps1)*k1*v -e5*(1-eps1)*rho1*k1*v)
               ds_inv[1] = -(e2*(-d2 - (1-f*eps1)*k2*v) + e4*(1-f*eps1)*k2*v -e5*(1-f*eps1)*rho2*k2*v)
               ds_inv[2] = -(e3*(-delta - m1*e) + tmp2)
               ds_inv[3] = -(e4*(-delta - m2*e) + tmp2)
               ds_inv[4] = -(Q - e1*(1-eps1)*k1*t1 + e2*(1 - f*eps1)*k2*t2 + e3*(1 - eps1)*k1*t1 + \
                             e4*(1 - f*eps1)*k2*t2 + e5*(-c - (1 - eps1)*rho1*k1*t1 - (1 - f*eps1)*rho2*k2*t2))
               ds_inv[5] = -(-e3*m1*t11 -e4*m2*t21 + e6*(bE*(t11+t21)/(t11+t21+Kb) - d_E*(t11+t21)/(t11+t21+Kd) - de
               return ds_inv
```

Solving the HIV model to determine optimal control parameter

In this case, we update our simulator class to include the forward and backward state variable and customize the simulation function to account for both forward and backward cases. The ϵ_1 parameter is constrained to be between 0 and 0.8.

In [1797]:
```python
class HIVSimulator():
    def __init__(self, immunity_type):
        # immunity of the individual
        if immunity_type == 'strong':
            self.params = (1e4,31.98,0.01,0.01,8e-7,1e-4,0.7,100.,13.,1.0,1.0,0.1,100.0,1e-5,1e-5,0.5,500
        else:
            self.params = (1e4,31.98,0.01,0.01,8e-7,1e-4,0.7,100.,13.,1.0,1.0,0.1,1.0,1e-5,1e-5,0.3,100,0

        self.f_state = []
        self.b_state = []

    def simulate(self,eps1,eps2,t,derivs,Q=None,state=None, backward=False):
        if backward:
            deriv_args = (state,Q,eps1,eps2,0.34,self.params)
            s = self.b_state
        else:
            deriv_args = (eps1,eps2,0.34,self.params)
            s = self.f_state
        #solving the ode using odeint
        sol = odeint(derivs, s, t, args=deriv_args)
        return sol

    def reset(self, state_type, randomize):
        """Reset the environment."""
        self.t = 0
        if state_type == 'low':
            self.f_state = [1000000., 3198., 0., 0., 1., 10.]
        elif state_type == 'high':
            self.f_state = [163573., 5., 11945., 46., 63919., 24.]
        elif state_type == 'early':
            self.f_state = [1000000., 3198., 1e-4, 1e-4, 1., 10.]
        if randomize:
            self.f_state = self.f_state + (self.f_state * np.random.uniform(-0.1,0.1,6))
        return self.f_state
```

In [1846]:

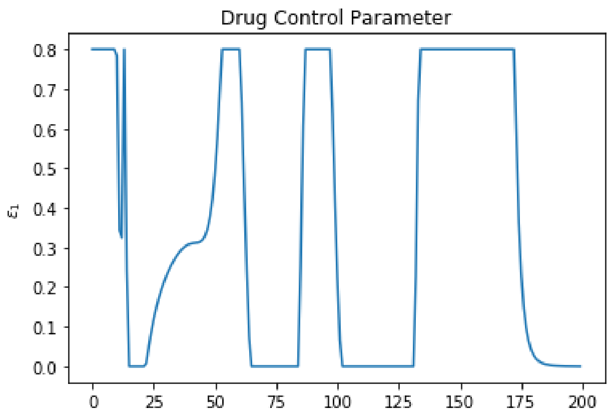
```
#solving the HIV equations
a = 0.0
b = 0.8
R = 10000

h = HIVSimulator('standard')

t = list(range(0,200))
t_inv = t[::-1]
error = 1000
eps_init = np.random.uniform(a,b) * np.ones(len(t))

obj_init = 1e20
obj_best = float("Inf")
counter = 0
while counter < 400:
    f_states = np.zeros((len(t)+1,6))
    f_states[0] = init_state
    h.reset('low',False)
    for t_it in t:
        f_states[t_it+1] = h.simulate(eps_init[t_it],0,[t_it,t_it+1],derivs_dt)[-1,:]
        h.f_state = f_states[t_it+1]
    v = f_states[1:,4]
    t1 = f_states[1:,0]
    t2 = f_states[1:,1]
    obj_new = np.sum(Q*v+R*eps_init**2)

    if obj_new < obj_best and counter>5:
        eps_best = eps_init.copy()
        v_best = v.copy()

    error = abs(obj_new - obj_init)/obj_init
    r_states = np.zeros((len(t)+1,6))
    for t_it in t_inv:
        h.b_state = r_states[t_it+1]
        r_states[t_it] = h.simulate(eps_init[t_it],0,[t_it+1,t_it],derivs_dt_inv,Q,list(f_states[t_it+1])
        h.r_state = r_states[t_it]

    e1 = r_states[:-1,0]
    e2 = r_states[:-1,1]
    e3 = r_states[:-1,2]
    e4 = r_states[:-1,3]
    e5 = r_states[:-1,4]
    e6 = r_states[:-1,5]
    eps_new = np.maximum(a, np.minimum(b, (-(e1-e3+rho1*e5)*k1*v*t1 - (e2-e4+rho2*e5)*f*k2*v*t2)/(2*R)))
    error = np.linalg.norm(eps_new - eps_init)/np.linalg.norm(eps_init)
    eps_init = eps_new.copy()
    obj_init = obj_new
    counter+= 1
```

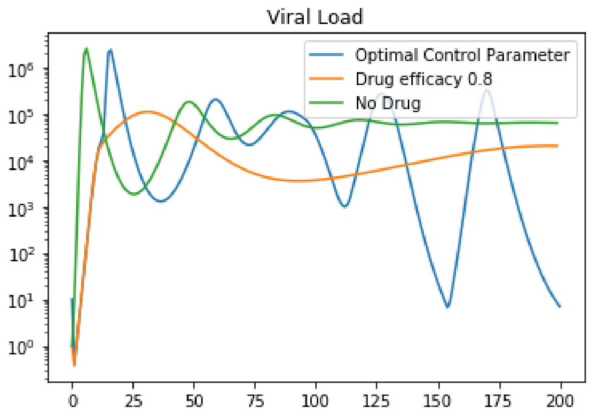
In [1847]:

```
# patient with epsilon1  = 0.8 (strong drug efficacy)
max_time = 200
dt = 1
h.reset('early',False)
t = list(range(0,max_time,dt))
full_eps = h.simulate(0.8,0,t,derivs_dt)

# patient with no drug being given
h.reset('early',False)
no_eps = h.simulate(0,0,t,derivs_dt)
```

Compare viral load for optimal control parameter with $\epsilon_1 = 0$ and $\epsilon_1 = 0.8$

```
In [1848]: plt.title("Drug Control Parameter")
           plt.plot(eps_init)
           plt.ylabel('$\epsilon_1$')
           plt.show()
           plt.plot(f_states[:,4], label = 'Optimal Control Parameter')
           plt.plot(full_eps[:,4], label = 'Drug efficacy 0.8')
           plt.plot(no_eps[:,4], label = 'No Drug')
           plt.legend(loc="upper right")
           plt.title("Viral Load")
           plt.yscale('log',basey=10)
```





As observed, the optimal control model tries to control the drug dosage for RTI (plot 1) and is able to minimize the viral load below 'no drug' scenario to certain extent. However, the viral load still jumps to quite high levels intermittently and thus, we focus on reinforcement learning-based optimization.

Reinforcement Learning

Exercise 1.6(10 point) Develop a drug dosage control algorithm using Reinforcement Learning(RL).

To simulate this scenario, we use our simulation model to generate trajectories for RL model. Then, we train a Policy Gradient based Reinforcement Learning model using batch data without access to the underlying simulation model to determine the optimal drug dosage for

RTI and PI, both. In line with Adams et. al.[2], we consider 4 possible actions:

1. Action 0: no drug, costs 0 (ϵ_1 = 0, ϵ_2 = 0)
2. Action 1: protease inhibitor only (ϵ_1 = 0, ϵ_2 = 0.3)
3. Action 2: RT inhibitor only, (ϵ_1 = 0.7, ϵ_2 = 0.0)
4. Action 3: both RT inhibitor and protease inhibitor, c(ϵ_1 = 0.7, ϵ_2 = 0.3)

The reward at each step is defined based on the current state and the action. In this case, we use the full cost function:
$$J(\epsilon_1, \epsilon_2) = E_t[QV(t) + R_1\epsilon_1^2 + R_2\epsilon_1^2 - SE(t)]$$

Here we use the following parameters in our objective:R1 =20000 , R2 = 2000, Q=0.1 , S=1000

Define Simulator for RL

We define our class to perform simulation. This is an updated version of our HIVSimulator class defined earlier

```
In [1823]: class HIVRL(object):
               state_names = ("T1", "T2", "T1*", "T2*", "V", "E")
               eps_values_for_actions = np.array([[0., 0.], [.7, 0.], [0., .3], [.7, .3]])

               def __init__(self,dt=1, derivs=None):
                   self.statespace_limits = np.array([[0., 1e8]] * 6)
                   self.model_derivatives = dsdt
                   self.dt = dt
                   self.state = []
                   self.reward_bound = 1e300
                   self.num_actions = 4
                   self.reset('high',False)

               def reset(self, state_type, randomize=False):
                   """Reset the environment."""
                   self.t = 0
                   if state_type == 'low':
                       self.state = [1000000., 3198., 0., 0., 1., 10.]
                   elif state_type == 'high':
                       self.state = [163573., 5., 11945., 46., 63919., 24.]
                   elif state_type == 'early':
                       self.state = [1000000., 3198., 1e-4, 1e-4, 1., 10.]
                   if randomize:
                       self.state = self.f_state + (self.f_state * np.random.uniform(-0.1,0.1,6))
                   self.state = np.array(self.state)
                   return self.state

               def observe(self):
                   return self.state

               def is_done(self, episode_length=200):
                   ##Check if the episode is complete
                   return True if self.t >= episode_length else False

               def calc_reward(self, action=0, state=None, **kw ):
                   #define the reward function
                   eps1, eps2 = self.eps_values_for_actions[action]
                   if state is None:
                       state = self.observe()
                   T1, T2, T1s, T2s, V, E = state

                   reward = -0.1*V - 2e4*eps1**2 - 2e3*eps2**2 + 1e3*E

                   # Constrain reward to be within specified range
                   if np.isnan(reward):
                       reward = -self.reward_bound
                   elif reward > self.reward_bound:
                       reward = self.reward_bound
                   elif reward < -self.reward_bound:
                       reward = -self.reward_bound
                   return reward


               def step(self, action):
                   self.t += 1
                   self.action = action
                   eps1, eps2 = self.eps_values_for_actions[action]
                   r = ode(self.model_derivatives).set_integrator('vode',nsteps=10000,method='bdf')
                   t0 = 0
                   deriv_args = (eps1, eps2)
                   r.set_initial_value(self.state, t0).set_f_params(deriv_args)
                   self.state = r.integrate(self.dt)
                   reward = self.calc_reward(action=action)
                   done = self.is_done()
                   return self.state, reward, done

           def dsdt(t, s, params):
               derivs = np.empty_like(s)
               eps1,eps2 = params
               T1, T2, T1s, T2s, V, E = s

               # baseline model parameter constants
               lambda1 = 1e4
               lambda2 = 31.98
```

```
    d1 = 0.01
    d2 = 0.01
    f = .34
    k1 = 8e-7
    k2 = 1e-4
    delta = .7
    m1 = 1e-5
    m2 = 1e-5
    NT = 100.
    c = 13.
    rho1 = 1.
    rho2 = 1.
    lambdaE = 1.
    bE = 0.3
    Kb = 100.
    d_E = 0.25
    Kd = 500.
    deltaE = 0.1
    out = s.copy()

    # compute derivatives
    tmp1 = (1. - eps1) * k1 * V * T1
    tmp2 = (1. - f * eps1) * k2 * V * T2
    out[0] = lambda1 - d1 * T1 - tmp1
    out[1] = lambda2 - d2 * T2 - tmp2
    out[2] = tmp1 - delta * T1s - m1 * E * T1s
    out[3] = tmp2 - delta * T2s - m2 * E * T2s
    out[4] = (1. - eps2) * NT * delta * (T1s + T2s) - c * V - ((1. - eps1) * rho1 * k1 * T1 + (1. - f * e
    out[5] = lambdaE + bE * (T1s + T2s) / (T1s + T2s + Kb) * E - d_E * (T1s + T2s) / (T1s + T2s + Kd) * E
    return out
```

Policy Gradient-based RL

For a given state s, a policy can be written as a probability distribution $\pi_\theta(s, a)$ over actions a, where θ is the parameter of the policy.

The reinforcement learning objective is to learn a θ^* that maximizes the objective function

$$J(\theta) = E_{\tau \sim \pi_\theta}[r(\tau)],$$

where τ is the trajectory sampled according to policy π_θ and $r(\tau)$ is the sum of discounted rewards on trajectory τ.

The policy gradient approach is to take the gradient of this objective

$$\nabla_\theta J(\theta) = \nabla_\theta \int \pi_\theta(\tau) r(\tau) d\tau = \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau = E_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)]$$

We sample trajectories $\tau^{(i)} = \{s_0^{(i)}, a_0^{(i)}, s_1^{(i)}, a_1^{(i)}, \cdots\} \sim \pi_\theta(\tau)$ and compute the gradient (w.r.t. θ) of loss function

$$Loss = -\frac{1}{N} \sum_i [\sum_{t=0}^T \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)})\ Q_t^{(i)}].$$

Define policy and episode generation functions

We refer one simulation trajectory as an episode. For reinforcement learning, in each iteration, we generate 10 trajectories using the action proposed by our policy agent. We use a one-layer perceptron network as our policy agent and use epsilon-greedy framework for taking actions using our policy i.e. select action recommended by trained policy agent with certain probability, otherwise select action randomly.

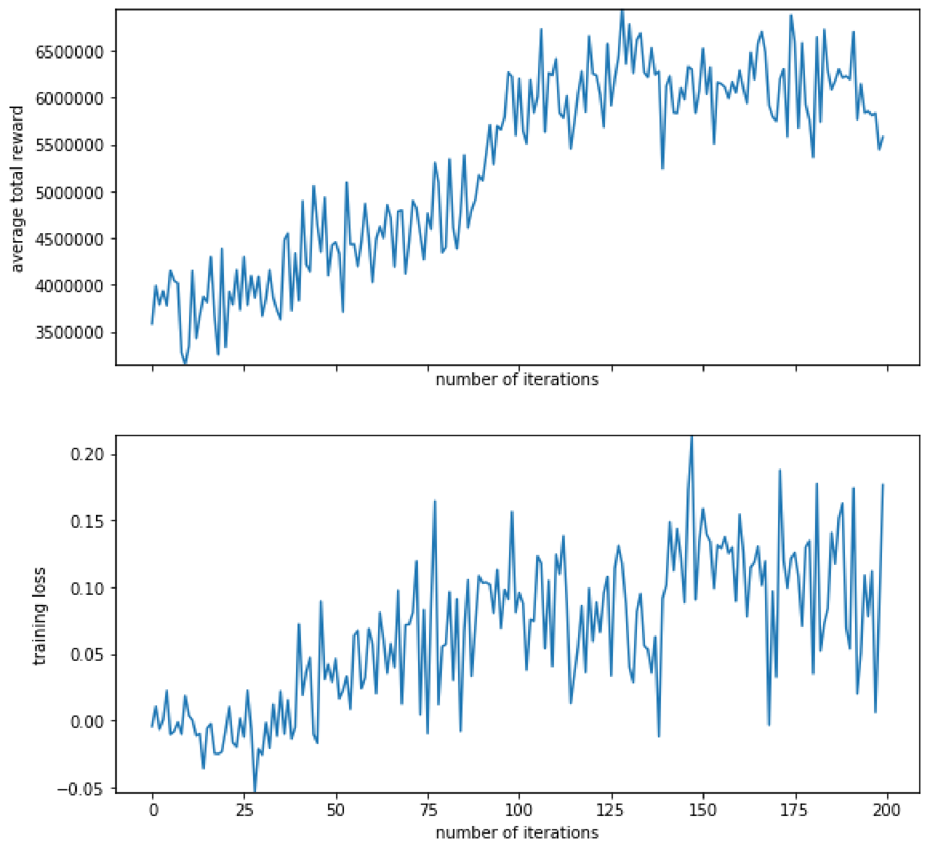
In [1831]:
```python
import matplotlib.pylab as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

class PolicyGradient(nn.Module):
    def __init__(self, outputs):
        super(PolicyGradient, self).__init__()
        self.network = nn.Sequential(
            nn.BatchNorm1d(num_features=6, affine=False),
            nn.Linear(6,10),
            nn.ReLU(),
            nn.Linear(10, outputs))

    def forward(self, x):
        x=x.double()
        x = self.network(x)
        return x

def sample_episode(env, policy, max_episode_length,epsilon):
    ob = env.reset('high')
    obs, acs, log_p, rewards, next_obs, terminals = [], [], [], [], [], []
    steps = 0
    while True:
        # use the most recent observation
        obs.append(ob)
        ac = sample_action(policy, ob, epsilon)
        acs.append(ac)
        # take that action and record results
        ob, rew, done = env.step(ac)
        # record result of taking that action
        steps += 1
        next_obs.append(ob)
        rewards.append(rew)
        if done or steps > max_episode_length:
            rollout_done = 1
        else:
            rollout_done = 0
        terminals.append(rollout_done)
        if rollout_done:
            break
    return obs, acs, rewards, next_obs, terminals


def sample_action(policy_net, obs, epsilon):
        if np.random.random() < epsilon:
            return np.random.randint(4)
        obs = torch.tensor(obs.reshape(1, -1), dtype=torch.float64)
        return (
            torch.distributions.Categorical(logits=policy_net.eval().double().forward(obs))
            .sample()
            .item()
        )

def sample_batch_episodes(env, policy, episodes_per_batch, max_episode_length,epsilon):
    episode_count = 0
    episodes = []
    for i in range(episodes_per_batch):
        episode = sample_episode(env, policy, max_episode_length,epsilon)
        episodes.append(episode)
    return episodes

def log_prob(policy_net,obs, action):
        log_probs = nn.functional.log_softmax(policy_net.forward(obs), dim=1)[:,]
        action_one_hot = nn.functional.one_hot(action, num_classes=4)
        return torch.sum(log_probs * action_one_hot, dim=1)

def reward_discounted(gamma,rewards):
    all_discounted_cumsums = []
    # for loop over steps (t) of the given rollout
    for start_time_index in range(len(rewards)):
        indices = np.arange(start_time_index, len(rewards))
        discounts = gamma ** (indices - start_time_index)
        all_discounted_cumsums.append(sum(discounts * rewards[start_time_index:]))
    return np.array(all_discounted_cumsums)
```

Learn RL-based treatment policy by simulating the model and applying policy gradient-based updates for 200 iterations

```
In [ ]: n_iter = 200
        batch_size = 10
        max_episode_length = 100
        epsilon = 1.0
        GAMMA = 0.999
        learning_rate = 1e-3

        policy_net = PolicyGradient(4).double()
        avg_rewards = np.zeros(n_iter)
        avg_episode_lengths = np.zeros(n_iter)
        env = HIVRL()
        log_loss = np.zeros(n_iter)
        optimizer = optim.Adam(policy_net.parameters(), lr=learning_rate)
        policy_net.train()

        for itr in range(n_iter):
            if itr % 10 == 0:
                print(f"*****Iteration {itr}*****")
            episodes = sample_batch_episodes(env, policy_net, batch_size, max_episode_length,epsilon)
            total_reward = 0
            obs = np.concatenate([tau[0] for tau in episodes], axis=0).astype(np.float64)
            acs = np.concatenate([tau[1] for tau in episodes], axis=0).astype(np.int64)
            obs = torch.from_numpy(obs)
            acs = torch.from_numpy(acs)

            disc_rewards = []
            for e in episodes:
                total_reward += np.sum(e[2])

            disc_rewards = np.concatenate([reward_discounted(GAMMA,tau[2]) for tau in episodes], axis=0).astype(r
            log_ps = log_prob(policy_net,obs,acs)

            avg_reward = total_reward/batch_size

            advantage = (disc_rewards - disc_rewards.mean())/disc_rewards.std() + 1e-8 #np.standardize(disc_rewar
            loss = -torch.mean(log_ps * torch.tensor(advantage, dtype=torch.float64))
            print(loss.item(),avg_reward)
            avg_rewards[itr] = avg_reward
            # Update network weights
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            log_loss[itr] = loss.item()
            # Update rule for epsilon s.t. after 100 iterations it's around 0.05.
            epsilon = np.maximum(0.05,epsilon*0.97)
```

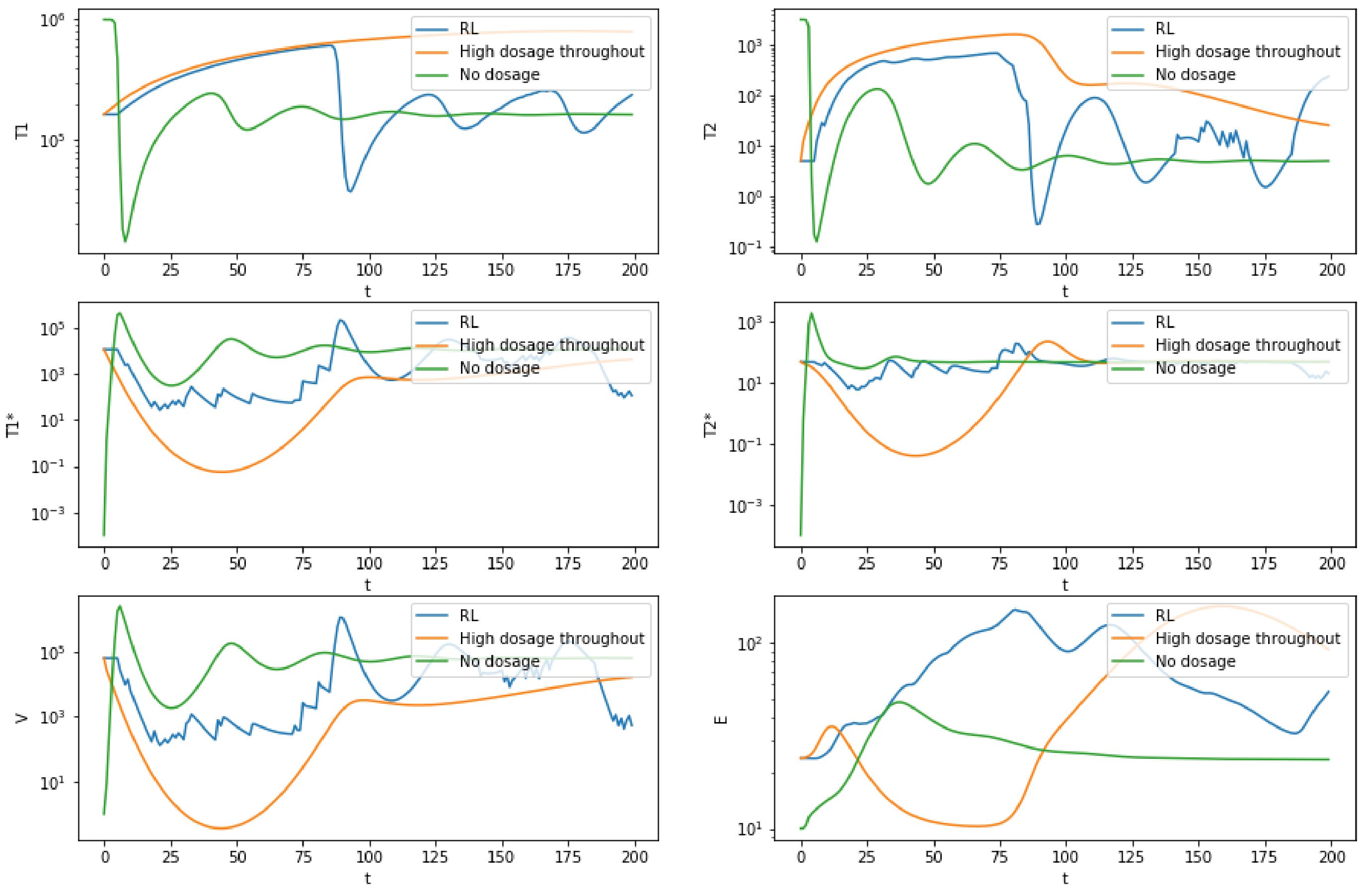
Plotting the reward and loss curves

In [1834]:
```
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=[9, 9])
ax1.plot(avg_rewards)
ax1.set_xlabel("number of iterations")
ax1.set_ylabel("average total reward")
ax1.set_ylim(avg_rewards.min(), avg_rewards.max())
ax2.plot(log_loss)
ax2.set_xlabel("number of iterations")
ax2.set_ylabel("training loss")
ax2.set_ylim(log_loss.min(), log_loss.max())
plt.show()
```

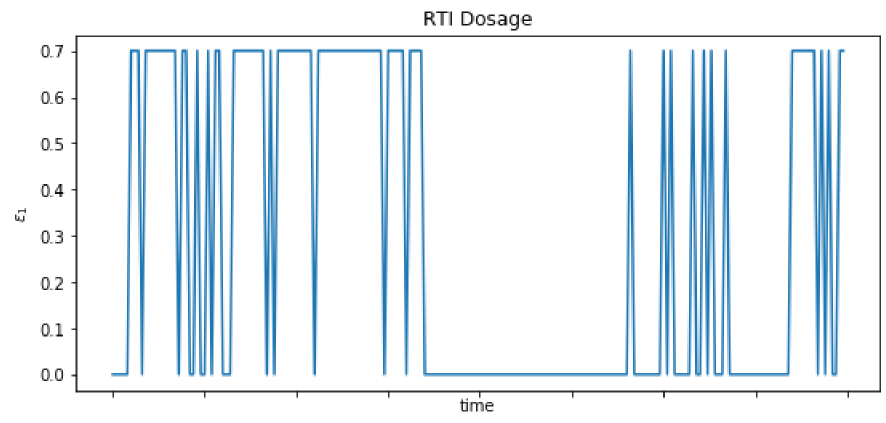


Plot the state space dynamics and drug dosages across time

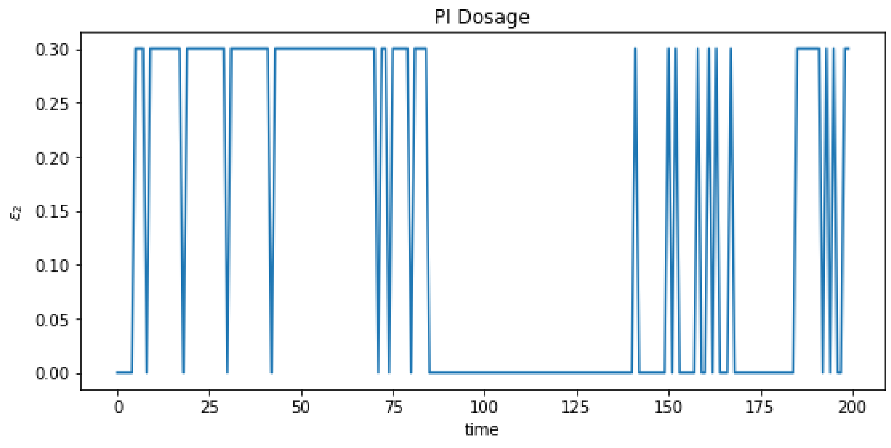
```
In [1842]: max_time = 200
dt = 1
episode = sample_episode(env, policy_net, max_time, 0)
obs = np.array(episode[0])[:max_time]

# patient with high drug dosage

h.reset('high',False)
t = list(range(0,max_time,dt))
full_eps = h.simulate(0.7,0.3,t,derivs_dt)

# patient with no drug being given
h.reset('early',False)
no_eps = h.simulate(0,0,t,derivs_dt)

sols = {'RL':obs, 'High dosage throughout':full_eps, 'No dosage':no_eps}
visualize_plot(t,sols,False)

#plotting the dosage strategy
eps1 = []
eps2 = []
for i in episode[1][:max_time]:
    if i in [1,3]:
        eps1.append(0.7)
    else:
        eps1.append(0.0)
    if i in [2,3]:
        eps2.append(0.3)
    else:
        eps2.append(0.0)

fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=[9, 9])
ax1.plot(eps1)
ax1.set_xlabel("time")
ax1.set_ylabel("$\epsilon_1$")
ax1.set_title("RTI Dosage")
ax2.plot(eps2)
ax2.set_xlabel("time")
ax2.set_ylabel("$\epsilon_2$")
ax2.set_title("PI Dosage")
plt.show()
```



T1
T2
T1*
T2*
V
E
t
RL
High dosage throughout
No dosage



RTI Dosage
ϵ_1
time



PI Dosage
ϵ_2
time

As observed, the reinforcement learning-based model is able to optimize the dosage for both RTI and PI and is able to consistently keep the viral load below the 'no drug scenario' and close to the 'full drug' scenario. CD4 cells count is also mainted quite well using RL-based treatment policy. The model tries to keep RTI and PI dosages to minimal and tries to minimize viral load taking help from the immune system cells(E).

References

1. Adams, Brian Michael, et al. "HIV dynamics: modeling, data analysis, and optimal treatment protocols." Journal of Computational and Applied Mathematics 184.1 (2005): 10-49.
2. Adams, Brian Michael, et al. Dynamic multidrug therapies for HIV: Optimal and STI control approaches. North Carolina State University. Center for Research in Scientific Computation, 2004.

Division of Labor

1. Farshad Rafiei: Cellular Automata (Part 1)
2. Aslihan Celik: ODE implementation (Part 2)
3. Anirudh Choudhary: ODE and Reinforcement Learning (Part 3)

Literature review was divided equally between the members of the group and each person contributed a part of literature review which is relevant to their coding section.

References

[1] Graw F, Perelson AS (2013) Spatial Aspects of HIV Infection. Mathematical Methods and Models in Biomedicine: Springer. pp. 3-31.

[2] Perelson, A. S., & Ribeiro, R. M. (2013). Modeling the within-host dynamics of HIV infection. BMC biology, 11(1), 96.

[3] Di Mascio, M., Ribeiro, R. M., Markowitz, M., Ho, D. D., & Perelson, A. S. (2004). Modeling the longterm control of viremia in HIV-1 infected patients treated with antiretroviral therapy. Mathematical biosciences, 188(1-2), 47-62.

[4] Ernst, D., Stan, G. B., Goncalves, J., & Wehenkel, L. (2006, December). Clinical data based optimal STI strategies for HIV: a reinforcement learning approach. In Proceedings of the 45th IEEE Conference on Decision and Control (pp. 667-672). IEEE.

[5] Adams, Brian Michael, et al. "HIV dynamics: modeling, data analysis, and optimal treatment protocols." Journal of Computational and Applied Mathematics 184.1 (2005): 10-49.

[6] Bitmead, R., Gevers, M., & Werts, V. “Adaptive Optimal Control: The Thinking Man’s GPC.” Prentice Hall International (1990)

[7] S. Parbhoo, J. Bogojeska, M. Zazzi, V. Roth, and F. Doshi-Velez,“Combining kernel and model based learning for hiv therapy selection,”AMIA Summits on Translational Science Proceedings, vol. 2017, p.239, 2017.

[8] S. Parbhoo, “A reinforcement learning design for hiv clinical trials,”Ph.D. dissertation, 2014.

[9] D .Wodarz, M .A .Nowak, “Specific therapy regimes could lead to long-term immunological control of HIV”, Proc .Natl .Acad .Sci .96 (1999) 14464–14469.

[10] S .Bonhoeffer, M .Rembiszewski, G .M .Ortiz, D .F .Nixon, “Risks and benefits of structured antiretroviral drug therapy interruptions in HIV-1 infection”, AIDS 14 (2000) 2313–2322.