DSAIRM Solutions: Basic Bacteria Model

2021-08-04

This is a walk-through for the *Basic Bacteria Model* app in DSAIRM.

If you haven't already loaded the `DSAIRM` package, it needs to be loaded.

```
library(DSAIRM)
```

Task 1

Start with 100 initial bacteria and an initial level of 1 for the immune response. Assume bacteria grow at a rate of 1 per day (i.e., g = 1), the carrying capacity is 10^5 and they live for about 2 days (remember that the inverse of the lifespan is the rate of death). Assume the immune response is activated and grows at a rate of 10^{-4}, kills bacteria at a rate of 10^{-4} and decays at a rate of 2 per day. Set simulation start time to 0 and final time of 100 (which we assume to be days). Set the time step to 0.01. As you'll see, if we run an ODE model, the time step is only relevant for plotting, not for the underlying model simulation (while this is not the case for the discrete time model). Only run the continuous time ODE model. Plot both x- and y-axes on a linear scale (i.e, no log scales for now). You can stick with ggplot as the plot engine. Run the simulation, see what you get. You should see some oscillations and then the system settles down, with bacteria and immune response at the end of the simulation at around 19996 and 3003, respectively. Change the time step to 0.05, re-run the simulation. Record the number of bacteria at the end of the simulation (Hint: not much changes).

Record

- Number of bacteria at end of simulation, *dt*=0.05

You likely performed the tasks by changing settings in the graphical interface. As you know from the package tutorial, it is also possible to call the underlying simulation functions directly. This is the approach we take for this and all other solutions. The **Further Resources** section of the app tells you that for this app, the relevant model functions are called `simulate_basicbacteria_ode` and `simulate_basicbacteria_discrete`.

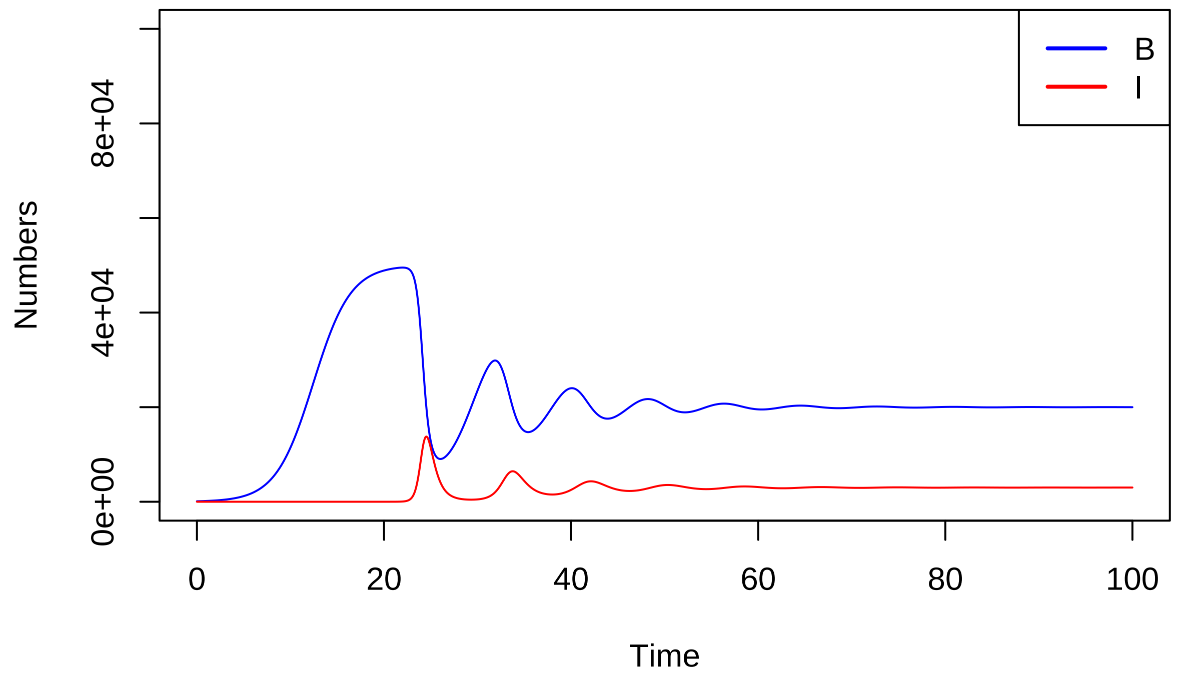
We start with the ODE model and call it with the specified settings, which can be done with this line of code.

```
#call the simulator function with the specified settings
sim_result <- simulate_basicbacteria_ode(B = 100, I = 1, g = 1,
                                        Bmax = 1e+05, dB = 0.5,
                                        k = 1e-4, r = 1e-4, dI = 2,
                                        tstart = 0, tfinal = 100, dt = 0.01)
```

The result that is returned is saved in the object *sim_result*. This is a list object, and it contains an object called *ts* which is the time series data from the simulation. That is, *ts* contains the values for each of the variables (*B* and *I* here) and the time. **All simulator functions return a list**, the elements contained in that list can differ between simulators. The help file for each simulator function describes what is returned.

We can plot the result obtained from the simulator using basic R plotting commands. The following lines of code produce a figure that looks similar to the one you get through the graphical interface.

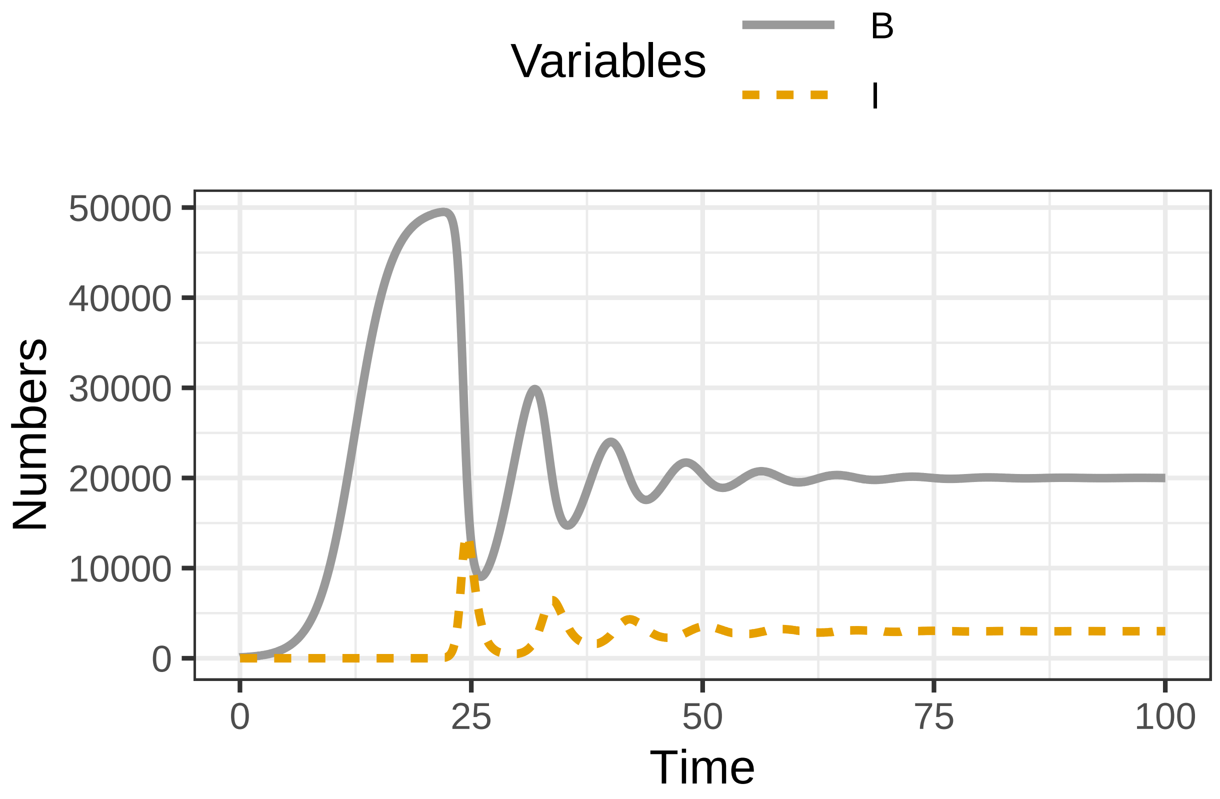
```
#plot using the base R plotting functions; all further plotting will be done using the functions includ
plot(sim_result$ts[,"time"], sim_result$ts[,"B"], type = 'l', col='blue',
     xlab='Time', ylab='Numbers', ylim=c(0,100000))
lines(sim_result$ts[,"time"], sim_result$ts[,"I"], type = 'l', col='red')
legend("topright", legend=c('B','I'), lwd=2, col=c('blue','red'))
```



To match the figure produced by the GUI, we will instead use the built-in plotting function `generate_ggplot` that comes with DSIARM. A similar function, `generate_text`, can be used to produce the information printed below each plot in the user interface. These functions require input to be in a specific form: a vector of lists. These lists are the time series *ts* from our simulation results (or sometimes lists with other structures). Here, there is only one figure, therefore the vector is of length 1. To create multiple plots, we would add another list to the vector for each plot we wanted to make.

For this outcome, we can simply apply the `list` command to the result returned from the simulator, and then send it to the `generate_` functions. The following lines of code then generate the figure and text you see in the GUI.

```
generate_ggplot(list(sim_result))
```



```
generate_text(list(sim_result))
```

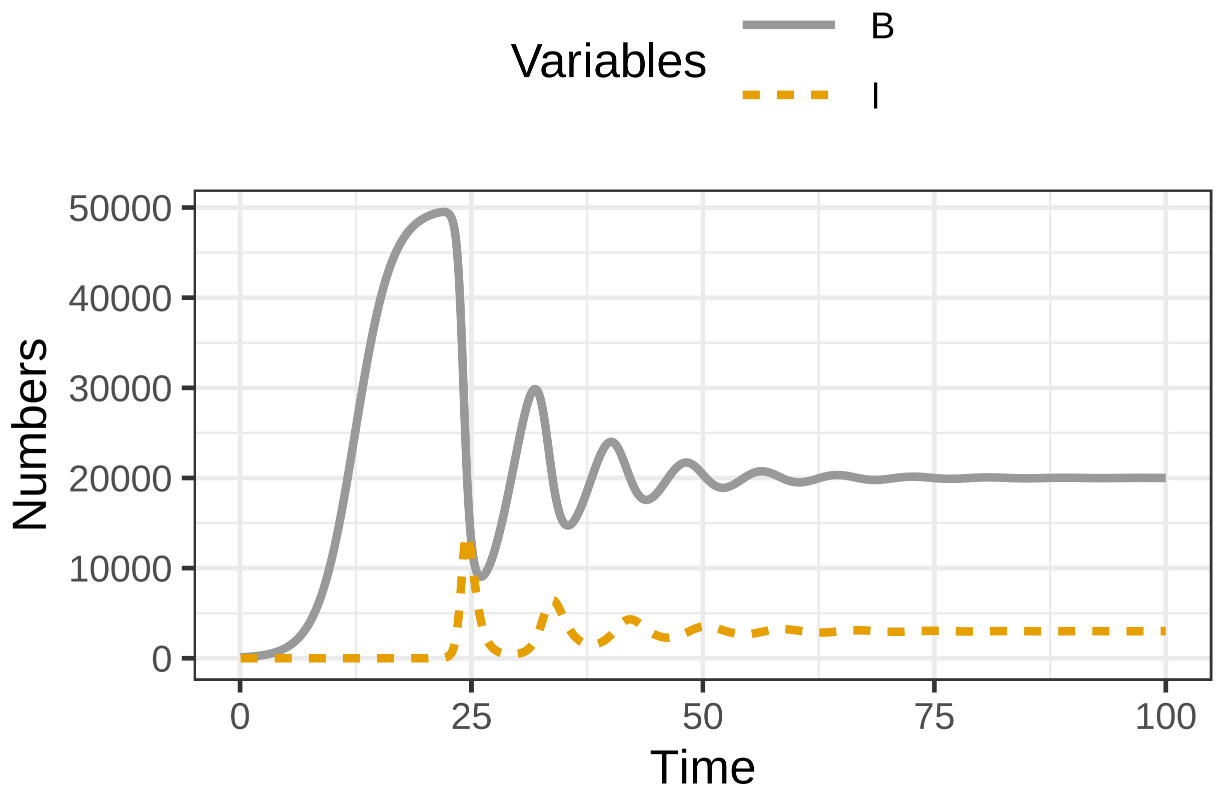
For the remainder of the solutions, we will use the built-in plot and text generation functions.

We can repeat above simulation with a changed time-step.

```
#call the simulator function with the specified settings
sim_result <- simulate_basicbacteria_ode(B = 100, I = 1, g = 1,
                                      Bmax = 1e+05, dB = 0.5,
                                      k = 1e-4, r = 1e-4, dI = 2,
                                      tstart = 0, tfinal = 100, dt = 0.05)
```

The plot and final numbers are basically the same.

```
generate_ggplot(list(sim_result))
```



```
generate_text(list(sim_result))
```

The text output provides the answer to the task question. We can also pull it out directly from the `sim_result` object, without using the `generate_text` function. We want the number of bacteria at the end of the simulation which is the number in the `B` column reported in the last row of `ts`. To get the last row we use the the `tail()` with the setting of 1 to specify we only want the last entry. We then round to the nearest integer and print the result. The code below shows how the value is obtained.

```
Bfinal=tail(sim_result$ts$B,1)
Bfinal_rounded=round(Bfinal,0)
print(Bfinal_rounded)
```

```
## [1] 19996
```

Answer for Task 1

Number of bacteria at end of simulation, *dt*=0.05: **19996**

Task 2

Play around with the plot settings. Switch the plotting to have x-axis, y-axis or both plotted on a log scale. Also try plotly as the plot engine. Leave all other settings as before. You should see that while the look of the plot changes, the underlying numbers do not. This is something to be aware of when you see plots in papers or produce your own. The best plot to use is the one that shows results of interest in the clearest form. Usually, the x-axis is linear and the y-axis is either linear or logarithmic. One nice feature about plotly is that the plot is interactive and you can read off numbers. Use this feature to determine the day at which the bacteria and immune response have their second peak. (*Hint: day 32 for bacteria, a bit later for the immune response.*)

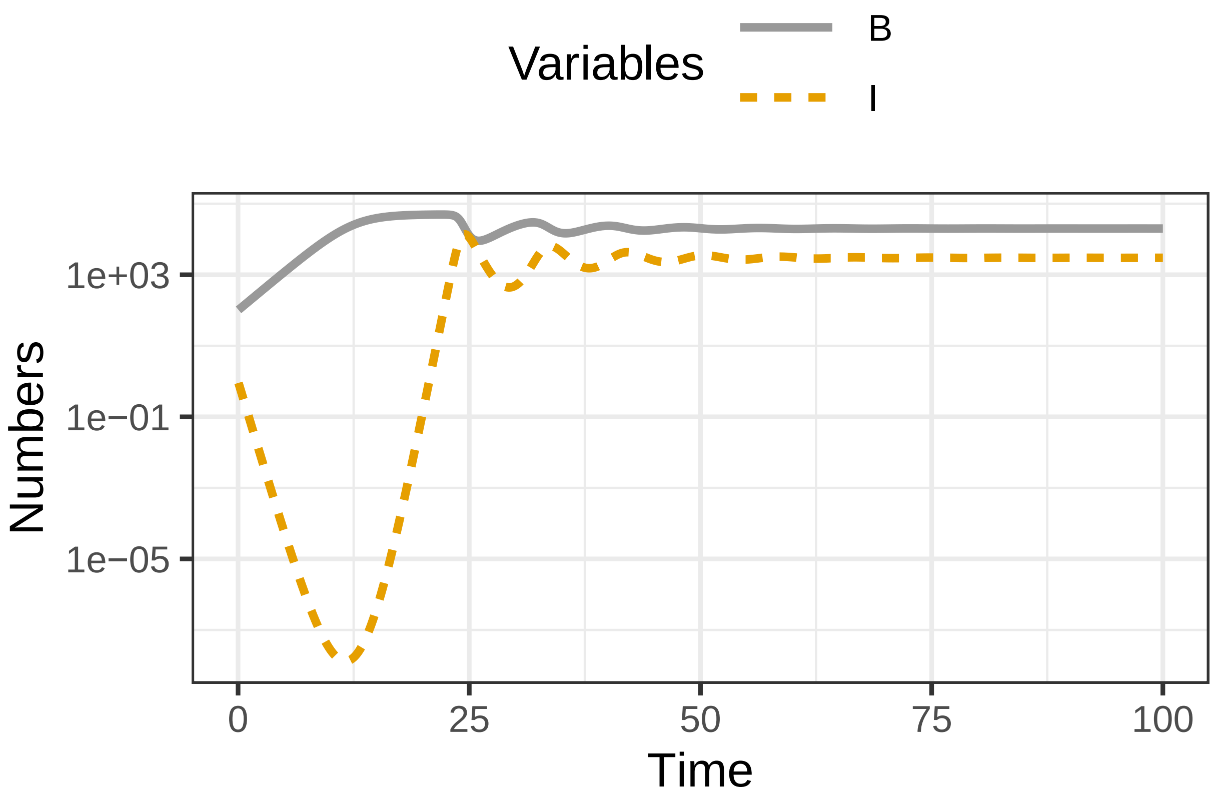
Record

- Day of second peak for the immune response

For this task, the change in plotting does not change the underlying results, so we we can simply retain `sim_result` and send it to the plot functions with different settings.

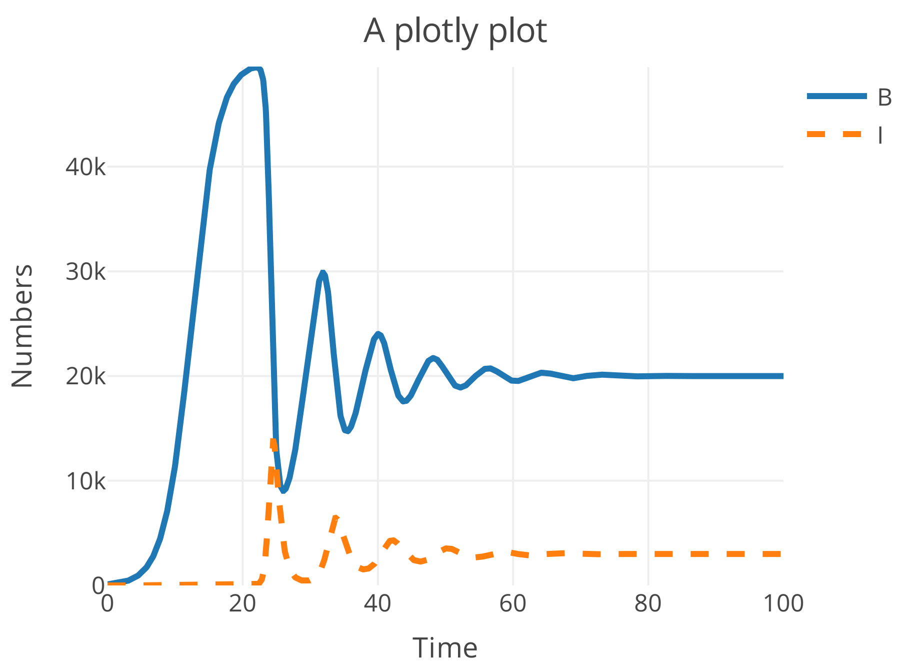
Here is an example plotting with a log scale for the y-axis. We need to modify the input structure a bit. The help file for `generate_ggplot` explains how.

```
plot_list = list(list(ts = sim_result$ts, yscale = "log10"))
generate_ggplot(plot_list)
```



There is also an equivalent helper function to generate `plotly` plots, which the following code does

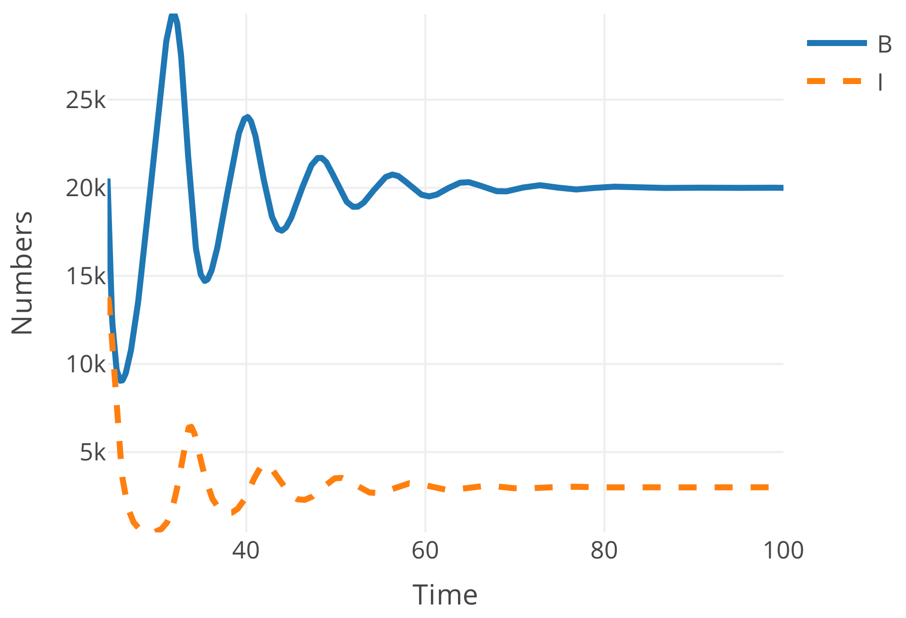
```
plot_list = list(list(ts = sim_result$ts, title = "A plotly plot"))
generate_plotly(plot_list)
```



From the plot, you can read off the time of the second peak for the immune response.

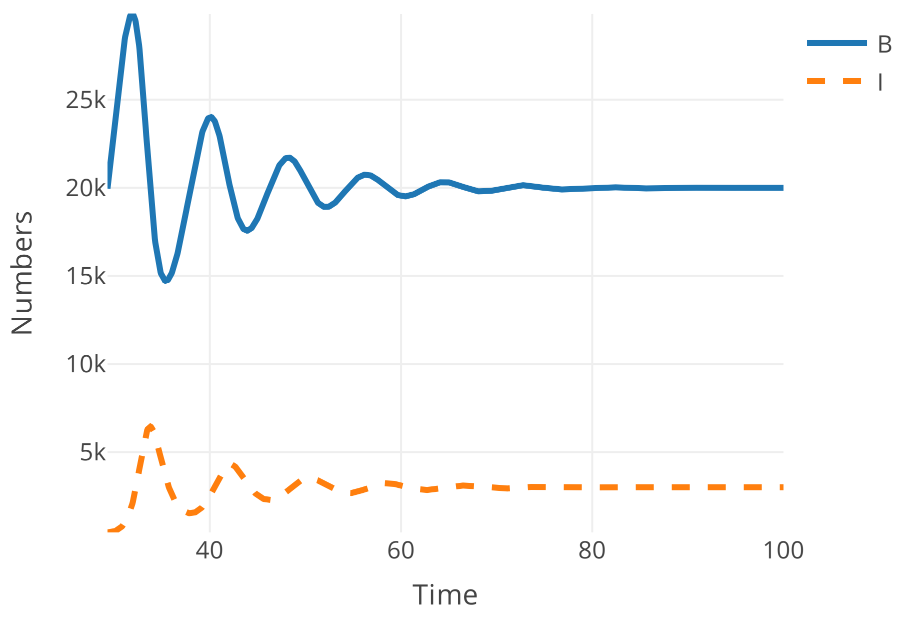
But we can also do it with code. This requires a bit of thinking of how to pull out the second peak. There are probably many ways, here is one way to do it. First we figure out when the first, highest peak happens, which is the maximum. Using the `which.max()` function we can identify the row in the time series dataframe corresponding to the maximum or peak of the immune response. Knowing which row the peak occurs, we can then subset the dataframe to exclude records from *before* the peak (using the `[these_rows, these_columns]` operator to index the dataframe).

```
# index at which peak 1 happens.
peak1 = which.max(sim_result$ts[,"I"])
# cut the time-series at the first I peak
ts_new = sim_result$ts[peak1:nrow(sim_result$ts),]
#just to check that it worked
generate_plotly(list(list(ts=ts_new)))
```



Let's do that again, now we find the minimum and chop / subset the time series dataframe again.

```
# minimum of I.
min1 = which.min(ts_new[,"I"])
# cut the time-series again
ts_new = ts_new[min1:nrow(ts_new),]
#just to check that it worked
generate_plotly(list(list(ts=ts_new)))
```



Ok so now the maximum of I is the second peak, which we want. So let's find it, and then determine at what time it happens.

```
# second maximum of I.
peak2 = which.max(ts_new[,"I"])
# find the time corresponding to the peak
t_peak2 = round(ts_new[peak2,"time"],0)
print(t_peak2)
```

```
## [1] 34
```

Ok, that worked and it agrees with our reading of the graph. Note that this was a bit cumbersome, and it only works if the peaks (and valleys) are getting smaller. Thus this is not a very robust method. I'm sure there are R packages available that can easily extract all maxima or minima of a time-series in a much more robust way. I wanted to show it here using a very basic approach, but if you need this feature often, it's probably best to find a good package that deals with time-series.

Answer for Task 2

Day of second peak for the immune response: **34**

Task 3

Go back to both linear scales for plotting. Keep all variable and parameter settings as before. Set *Models to run* to *both*. This runs and shows both the continuous-time and discrete-time models. Start with a time step of 0.01. Run the simulation. You should see the results from the 2 models essentially on top of each other and barely distinguishable. You should see that the maximum number of bacteria is 49,513 for the ODE model, and very close for the discrete-time model.

Record

- Maximum number of bacteria for the discrete-time model

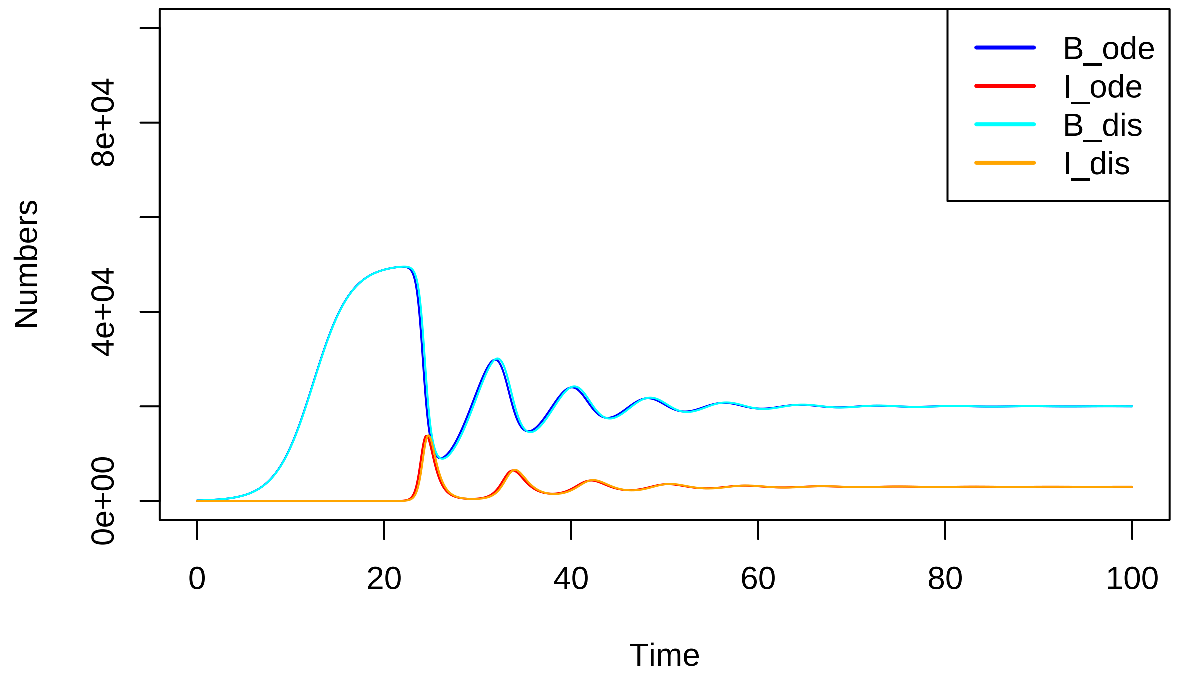
We now need to run both models. Here is the code to do so.

```
sim_result1 <- simulate_basicbacteria_ode(B = 100, I = 1, g = 1,
                                         Bmax = 1e+05, dB = 0.5,
                                         k = 1e-4, r = 1e-4, dI = 2,
                                         tstart = 0, tfinal = 100, dt = 0.01)

sim_result2 <- simulate_basicbacteria_discrete(B = 100, I = 1, g = 1,
                                         Bmax = 1e+05, dB = 0.5,
                                         k = 1e-4, r = 1e-4, dI = 2,
                                         tstart = 0, tfinal = 100, dt = 0.01)
```

If we wanted to use the generate_ functions, we would need to do some processing of the results obtained from the simulation runs to get the right list structure. This might be confusing, so we skip it here and instead just plot using base R plot commands.

```
#plot using the base R plotting functions; all further plotting will be done using the functions includ
plot(sim_result1$ts[,"time"], sim_result1$ts[,"B"], type = 'l', col='blue',
     xlab='Time', ylab='Numbers', ylim=c(0,100000))
lines(sim_result1$ts[,"time"], sim_result1$ts[,"I"], type = 'l', col='red')
lines(sim_result2$ts[,"time"], sim_result2$ts[,"B"], type = 'l', col='cyan')
lines(sim_result2$ts[,"time"], sim_result2$ts[,"I"], type = 'l', col='orange')
legend("topright", legend=c('B_ode','I_ode','B_dis','I_dis'), lwd=2, col=c('blue','red','cyan','orange'
```



We can get the maximum number of bacteria for the two models by using the max function. We also round again.

```
Bmax_ode = round(max(sim_result1$ts[,"B"]),0)
Bmax_dis = round(max(sim_result2$ts[,"B"]),0)
cat('Maximum of B for ODE and discrete models:', Bmax_ode, Bmax_dis)
```

```
## Maximum of B for ODE and discrete models: 49513 49550
```

Answer for Task 3

Maximum number of bacteria for the discrete-time model: **49550**

Task 4

Now try different values for the time step, *dt*. Leave all other settings as before. You should notice that as *dt* gets larger, the continuous-time model results remain the same, but the discrete-time models change and start moving away from those of the continous-time model. At a time step above 0.1, the results start to look very different. Somewhere above a time step of 0.5, it becomes so large that for these parameter settings, the simulation 'crashes' and you get an error message.

Record

- Final value for **B**, continous model, *dt*=0.1
- Final value for **B**, discrete-time model, *dt*=0.1

I'll just show the results for the $dt = 0.1$ time step, I'm sure you were able to obtain results for other time steps.

```
sim_result1 <- simulate_basicbacteria_ode(B = 100, I = 1, g = 1,
                                          Bmax = 1e+05, dB = 0.5,
                                          k = 1e-4, r = 1e-4, dI = 2,
                                          tstart = 0, tfinal = 100, dt = 0.1)

sim_result2 <- simulate_basicbacteria_discrete(B = 100, I = 1, g = 1,
                                          Bmax = 1e+05, dB = 0.5,
                                          k = 1e-4, r = 1e-4, dI = 2,
                                          tstart = 0, tfinal = 100, dt = 0.1)
```

You could use the same code as above to make another plot, I'll skip that and go straight to computing the quantities of interest.

```
Bfinal_ode = round(tail(sim_result1$ts[,"B"],1),0)
Bfinal_dis = round(tail(sim_result2$ts[,"B"],1),0)
cat('Final value of B for ODE and discrete models:', Bfinal_ode, Bfinal_dis)
```

```
## Final value of B for ODE and discrete models: 19996 20107
```

As expected, the continuous model is not affected since the time step is only relevant for the points that are returned, it doesn't impact the underlying simulation time-step. For the discrete-time model, the time-step does have an impact.

Answer for Task 4

Final value for **B**, continous model, *dt*=0.1: **19996**

Final value for **B**, discrete-time model, *dt*=0.1: **20107**

Task 5

Now we'll explore how model parameters impact outcomes. Set *dt* to 0.01, change the simulation time, *tfinal*, to 200 days. Also change the bacteria growth rate from 1 to 2 per day. Leave all other settings as before (you can keep running the discrete model, or switch back to ODE only). Run the simulation. Switch back and forth between growth rates of 1 and 2 and examine how bacteria and immune response dynamics change. Then, also try a growth rate of 3.

Record

- Final value for **B**, continous model, $g = 2$
- Final value for **B**, continous model, $g = 3$

```
sim_result1 <- simulate_basicbacteria_ode(B = 100, I = 1, g = 2,
                                          Bmax = 1e+05, dB = 0.5,
                                          k = 1e-4, r = 1e-4, dI = 2,
                                          tstart = 0, tfinal = 200, dt = 0.01)

sim_result2 <- simulate_basicbacteria_ode(B = 100, I = 1, g = 3,
                                          Bmax = 1e+05, dB = 0.5,
                                          k = 1e-4, r = 1e-4, dI = 2,
                                          tstart = 0, tfinal = 200, dt = 0.01)
```

```
Bfinal1 = round(tail(sim_result1$ts[,"B"],1),0)
Bfinal2 = round(tail(sim_result2$ts[,"B"],1),0)
cat('Final values of B:', Bfinal1, Bfinal2)
```

```
## Final values of B: 20000 20000
```

Maybe surprisingly, the rate of bacteria growth does not have an impact on the levels of bacteria at steady state. The next task explores this further.

Answer for Task 5

Final value for **B**, continous model, $g = 2$: **20000**

Final value for **B**, continous model, $g = 3$: **20000**

Task 6

In the previous task, you might have been surprised to find that the bacteria growth rate has no impact on the number of bacteria at the final, steady state. This is an indication that even a simple 2-variable model can lead to interesting, and maybe non-intuitive results.

For a model as simple as the one we have here, one can mathematically compute the steady state, i.e., the state at which bacteria and immune response don't change further. To do so, realize that *no change* means both left hand sides of the ODE model are zero. Equivalently, for the discrete time model, it means $B_{t+dt} = B_t$ and the same for **I**. We'll focus on the ODE model here. With the right side of the equations being zero, the model turns into just 2 algebraic equations, namely $0 = gB(1 - B/B_{max}) - d_B B - kBI$ and $0 = rBI - d_I I$. You can now solve this such that you end up with two equations of the form $B = XX$ and $I = YY$ where XX and YY are some combinations of model parameters. Let's do that for the second equation.

We rewrite the equation as $rBI = d_I I$, then divide by **I** and r to arrive at $B = d_I/r$. This shows that indeed, the number of bacteria at the steady state does not depend on the growth rate g. I'll let you do the same for the first equation to get **I** at steady state (note that at some poin in the solving process, you'll have to insert the steady state value of **B** we just found into the equation). You should end up with $I = (rB_{max}(g - d_B) - d_I g)/(krB_{max})$. Based on this, we expect that doubling d_I will lead to an increase (doubling) in B at steady state, and a decrease in **I**. Let's test that.

Set everything back as in task 1 (you can use the *Reset Inputs* button), but increase the simulation time to 200 days (to ensure we reach steady state). Run the model. Record **B** and **I** at the end. Then set $d_I = 3$, run again, and again record values for **B** and **I**. Do it again for $d_I = 4$.

Record

- Final value for **B**, continous model, $d_I = 2$
- Final value for **B**, continous model, $d_I = 3$
- Final value for **B**, continous model, $d_I = 4$
- Final value for **I**, continous model, $d_I = 2$
- Final value for **I**, continous model, $d_I = 3$
- Final value for **I**, continous model, $d_I = 4$

Now we need to run the model again, now 3 times. It's reasonable to just copy the code 3 times. But I want to show you that you can also use loops. That approach becomes useful if you want to run the model multiple times, e.g., if you scan over a parameter (something we'll do in later apps).

```
all_results = list() #store all results in a big list
dI_vec = c(2,3,4)
for (n in 1:length(dI_vec))
{
all_results[[n]] <- simulate_basicbacteria_ode(B = 100, I = 1, g = 1,
                                     Bmax = 1e+05, dB = 0.5,
                                     k = 1e-4, r = 1e-4, dI = dI_vec[n],
                                     tstart = 0, tfinal = 200, dt = 0.01)
}
```

We can use the same loop approach to pull out the final values

```
Bfinal = rep(0,length(dI_vec))
Ifinal = rep(0,length(dI_vec))
for (n in 1:length(dI_vec))
{
  Bfinal[n] = round(tail(all_results[[n]]$ts[,"B"],1),0)
  Ifinal[n] = round(tail(all_results[[n]]$ts[,"I"],1),0)
}

cat('Final values of B:', Bfinal)
```

```
## Final values of B: 20000 30000 40000
```

```
cat('Final values of I:', Ifinal)
```

```
## Final values of I: 3000 2000 1000
```

Answer for Task 6

Final value for **B**, continous model, $d_I = 2$: **20000**

Final value for **B**, continous model, $d_I = 3$: **30000**

Final value for **B**, continous model, $d_I = 4$: **40000**

Final value for **I**, continous model, $d_I = 2$: **3000**

Final value for **I**, continous model, $d_I = 3$: **2000**

Final value for **I**, continous model, $d_I = 4$: **1000**

Task 7

Again, set everything back as in task 1 (you can use the *Reset Inputs* button). Set the y-axis to log for the plot. Change immune response decay rate, d_I, to 0.2. You should get a plot where both immune response and bacteria numbers drop below 1 and take on fractional values. Contemplate what that means. If not clear, re-read the bullet points at the start of this page. This is one of the draw-backs of ODE based models, and we'll revisit this topic in the stochastic apps.

Record

- Minimum value of **B** (report as X.YZ)
- Minimum value of **I** (report as X.YZ)

This runs the simulation with the altered value for d_I.

```
#call the simulator function with the specified settings
sim_result <- simulate_basicbacteria_ode(B = 100, I = 1, g = 1,
                                         Bmax = 1e+05, dB = 0.5,
                                         k = 1e-4, r = 1e-4, dI = 0.2,
                                         tstart = 0, tfinal = 100, dt = 0.01)
```

We can get the minimum using the `min` function.

```
Bmin = round(min(sim_result$ts[,"B"]),2)
Imin = round(min(sim_result$ts[,"I"]),2)
cat('Minimum for B and I:', Bmin, Imin)
```

```
## Minimum for B and I: 0.1 0.44
```

Both values are below 1. This might make sense for the immune response if we measure it in some arbitrary units. But if we assume that bacteria in the model correspond to actual bacteria (or immune response to actual immune cells), then values below 1 - and in fact any fractional value - doesn't make much sense. This is an inherent limitation of the ODE or discrete-time models. One way to resolve this issue is discussed in the stochastic model apps.

Answer for Task 7

Minimum value of **B** (report as X.YZ): **0.1**

Minimum value of **I** (report as X.YZ): **0.44**

Task 8

Go wild! Change any inputs you want to change, see how it affects the results. To start building intuition, it is best to change one quantity at a time. Before you run a simulation with new settings, contemplate what you think might happen. Then see if it does. Always go through an iterative cylce: Think about expectations -> run simulation -> update understanding (**Do Science/Research**).

Record

- Nothing

This is an open-ended exploration, with nothing to report.

Answer for Task 8

Nothing