Evolutionary Dynamics
Chapter 10 - HIV Infection
Scenario 1: One virus strain, one immune response, no mutation
(c) 2009 Daniel Rosenbloom. Feel free to use/modify/redistribute with attribution. www.danielrosenbloom.com

In[1]:=

Set up the system (double-click margin at right to open up a section):

Clear out all variables from your current session:

```
In[2]:= Clear["`*"];
```

Import the package that allows display of legends in plots:

```
In[3]:= Needs["PlotLegends`"];
```

See Equations 10.1 on p. 171, using only the subscript i = 1

```
In[4]:= eqn1 = D[v1[t], t] == r * v1[t] - p * x1[t] * v1[t]

Out[4]= v1′[t] == r v1[t] - p v1[t] x1[t]

In[5]:= eqn2 = D[x1[t], t] == c * v1[t] - b * x1[t]

Out[5]= x1′[t] == c v1[t] - b x1[t]
```

Define initial conditions. Start with 0.1 virus, no immune response:

```
In[6]:= init1 = v1[0] == 0.1

Out[6]= v1[0] == 0.1

In[7]:= init2 = x1[0] == 0.0

Out[7]= x1[0] == 0.
```

Parameters (similar to bottom of p. 172). Note that this does not automatically set the parameters of the diff. eq -- rather, it defines a variable called "params" that can be applied to the diff. eq. using the syntax "/. params":

In[8]:=
```
params = {r → 2.5, p → 2, c → 0.2, b → 0.1}
```

Out[8]=
```
{r → 2.5, p → 2, c → 0.2, b → 0.1}
```

Define the system by evaluating our four equations, given the parameters:

In[9]:=
```
system1 = {eqn1, eqn2, init1, init2} /. params
```

Out[9]=
```
{v1′[t] == 2.5 v1[t] - 2 v1[t] x1[t], x1′[t] == 0.2 v1[t] - 0.1 x1[t], v1[0] == 0.1, x1[0] == 0.}
```

1) How to plot the ODE:

The function NDSolve numerically integrates the system of ODEs (ordinary differential equations). Syntax is NDSolve[list of equations, list of dependent variables, independent variable with range].

In[10]:=
```
solution1 = NDSolve[system1, {x1, v1}, {t, 0, 100}]
```

Out[10]=
```
{{x1 → InterpolatingFunction[{{0., 100.}}, <>], v1 → InterpolatingFunction[{{0., 100.}}, <>]}}
```

Look at the format of the "solution1" object that the NDSolve function outputted. It is a list containing two elements:

1) An element assigning x1 (the immune response) to an "Interpolating Function"

2) An element assigning v1 (the amount of virus) to an "Interpolating Function"

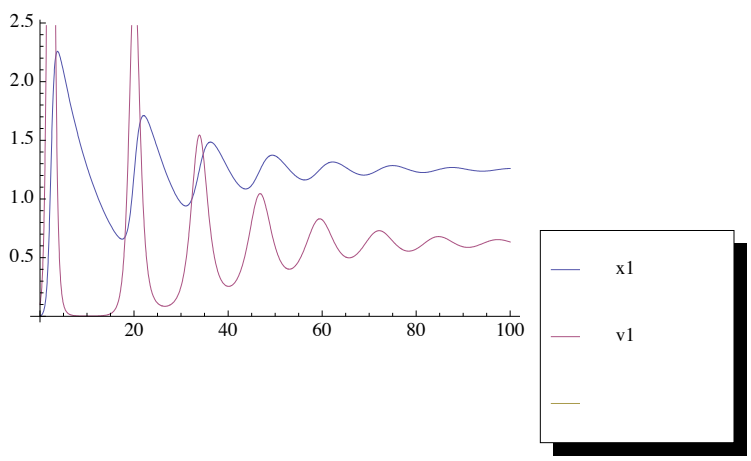
(It's not crucial to understand exactly what the InterpolatingFunction syntax is doing, but the help documentation explains it well.)

In order to plot the system, we need to evaluate the functions x1, v1 using solution1.

First, plot each variable versus time.

In[11]:=
```
Plot[
 Evaluate[{x1[t], v1[t]} /. solution1],
 (* Plot both variables, evaluated using solution1.  *)
 {t, 0, 100},   (* Time ranges from 0 to 100 *)
 PlotLegend → {"x1", "v1"}, LegendPosition → {1, -1} (* Displays legend in lower-right *)
]
```

Out[11]=



Solve for the equilibrium points by setting the RHS of our equations equal to zero. (Note that any equation is a subscriptable object: Subscript 1 gets you the LHS, and subscript 2 gets you the RHS.)

In[12]:=
```
equilibrium1 = Solve[{eqn1[[2]] == 0, eqn2[[2]] == 0}, {v1[t], x1[t]}]
```

Out[12]= $\left\{\left\{v1[t] \to \frac{b\,r}{c\,p}, x1[t] \to \frac{r}{p}\right\}, \{x1[t] \to 0, v1[t] \to 0\}\right\}$

(p. 172: "The system settles in damped oscillations to the equilibrium...")

We care about the first equilibrium point (stable equilibrium). Evaluate it at our parameters:

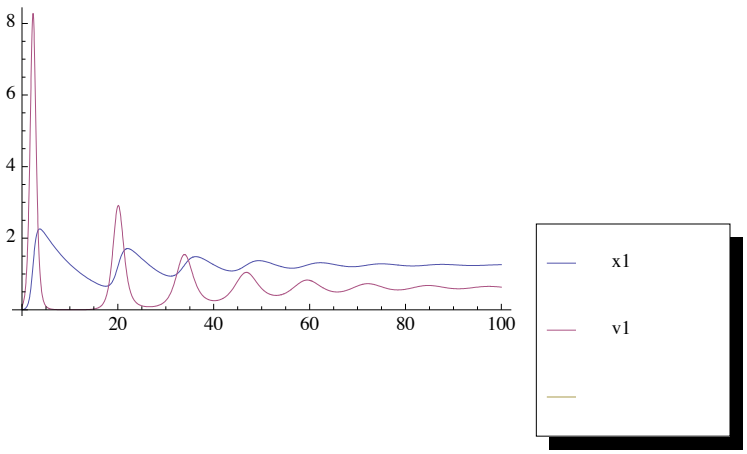
In[13]:=
```
stableEq1 = equilibrium1[[1]] /. params
```

Out[13]= {v1[t] → 0.625, x1[t] → 1.25}

By default, the above plot doesn't show the entire range. Use PlotRange -> Full to see it all:

In[14]:=
```
Plot[
 Evaluate[{x1[t], v1[t]} /. solution1],
 (* Plot both variables, evaluated using solution1 *)
 {t, 0, 100},   (* Time ranges from 0 to 100 *)
 PlotRange → Full, (* Shows everything *)
 PlotLegend → {"x1", "v1"}, LegendPosition → {1, -1} (* Displays legend in lower-right *)
]
```

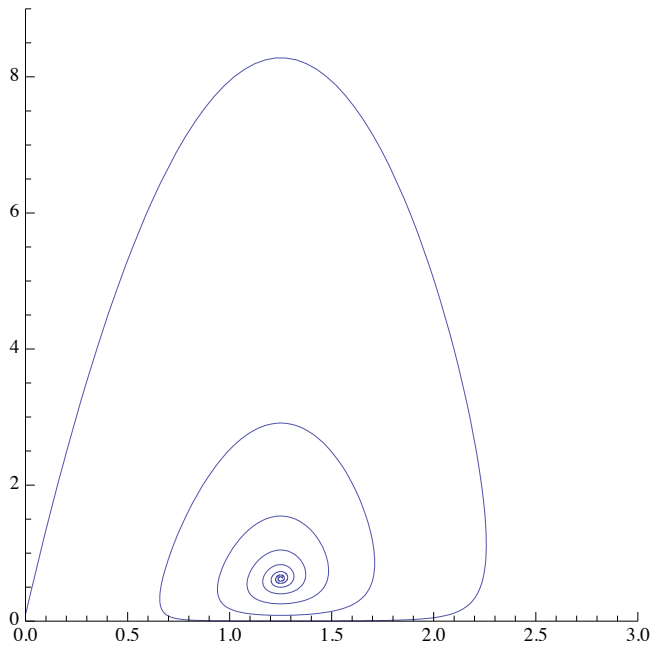
Out[14]=



Plot both variables parametrically (x on horizontal axis, v on vertical axis):

In[15]:=
```
ParametricPlot[
 Evaluate[{x1[t], v1[t]} /. solution1],
 (* Plot both variables, evaluated using solution1 *)
 {t, 0, 100},  (* Time ranges from 0 to 100 *)
 PlotRange → {{0, 3}, {0, 9}},  (* Horizontal axis from 0 to 3, Vertical from 0 to 9 *)
 AspectRatio → 1  (* Ensure that the plot is square *)
]
```

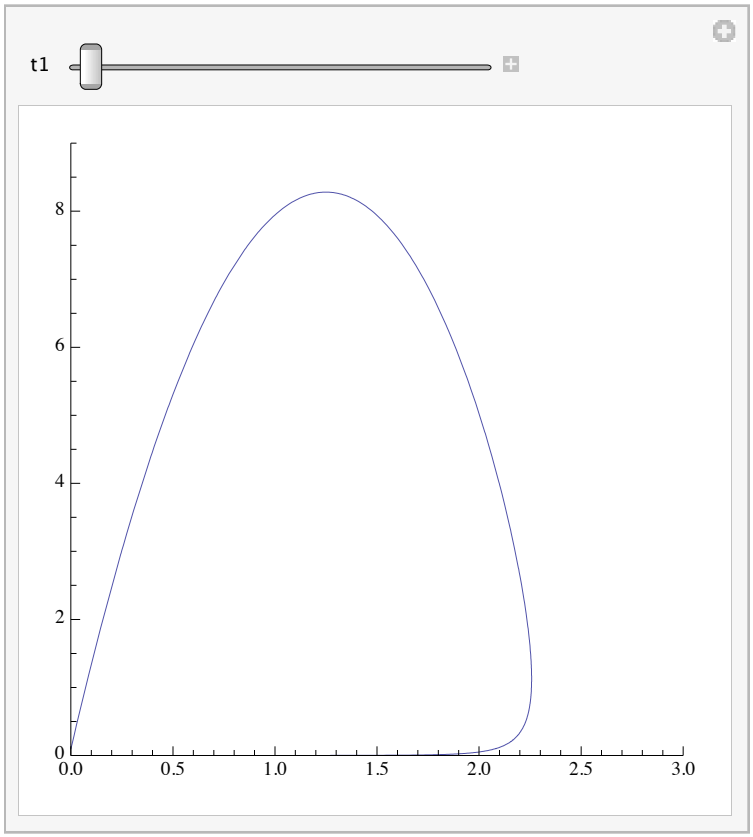
Out[15]=



Let's animate this plot! Use the Manipulate function:

In[16]:=
```
Manipulate[
 ParametricPlot[
  Evaluate[{x1[t], v1[t]} /. solution1],
  (* Plot both variables, evaluated using solution1 *)
  {t, 0, t1},   (* Time ranges from 0 to t1 ... and t1
   is a variable that the user can manipulate *)
  PlotRange → {{0, 3}, {0, 9}},   (* Horizontal axis from 0 to 3, Vertical from 0 to 18 *)
  AspectRatio → 1  (* Ensure that the plot is square *)
 ], {t1, 10, 100}]  (* Lets user manipulate the value of t1, from 10 to 100 *)
```

Out[16]=



2) How to measure the period of oscillations, graphically:

Say that we want to explore different parts of this plot, and display them side-by-side. We can display different plots together in a Table. Define four different plot ranges, successively zooming in on the equilibrium:

In[17]:=
```
pr1 = {{0, 3}, {0, 9}}
pr2 = {{0.5, 2}, {0, 4.5}}
pr3 = {{1.1, 1.4}, {0.2, 1.1}}
pr4 = {{1.22, 1.28}, {0.53, 0.72}}
```

Out[17]= {{0, 3}, {0, 9}}

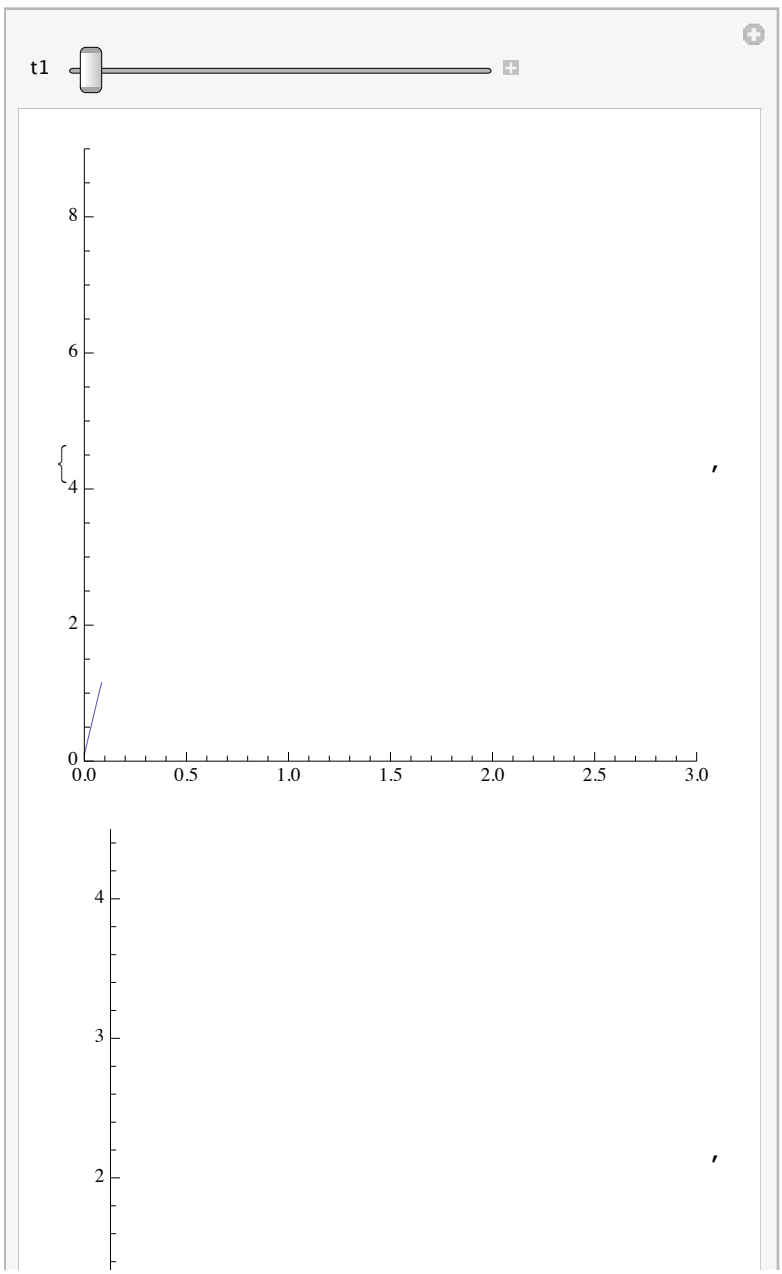
Out[18]= {{0.5, 2}, {0, 4.5}}

Out[19]= {{1.1, 1.4}, {0.2, 1.1}}

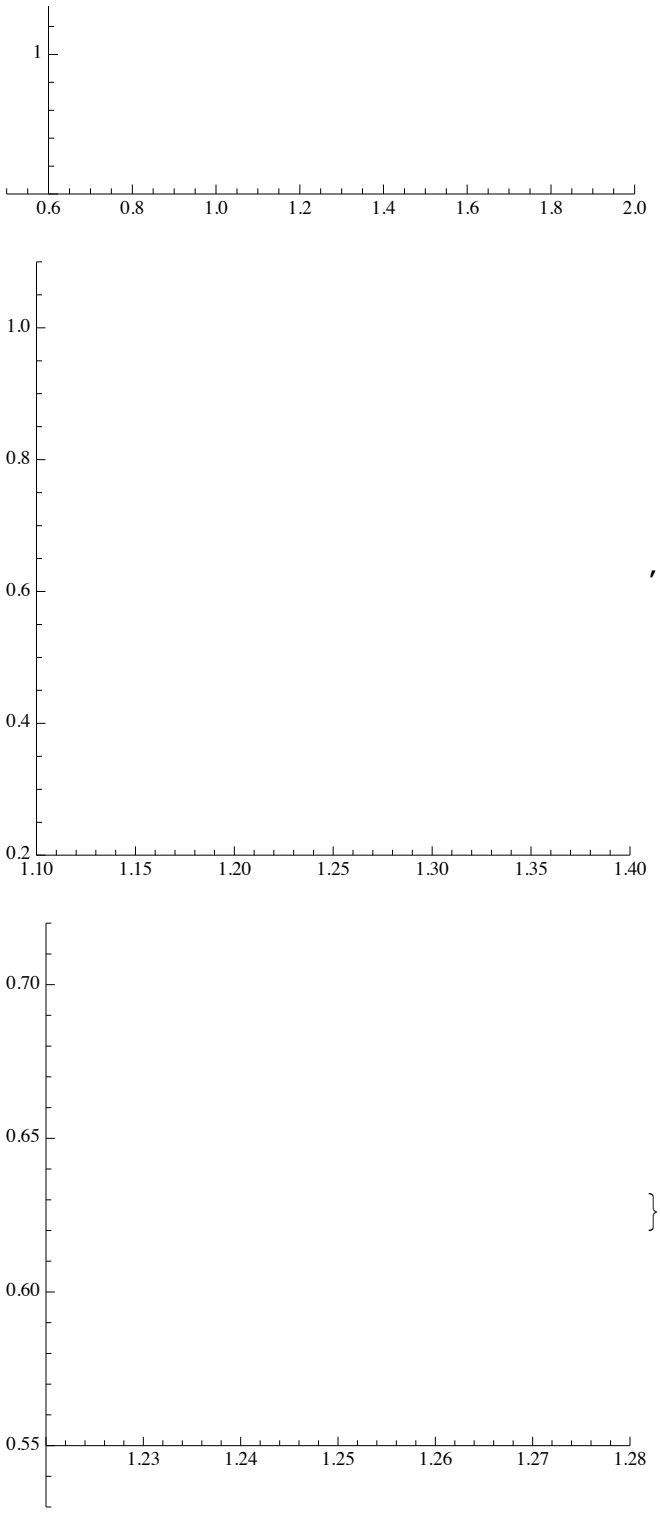
Out[20]= {{1.22, 1.28}, {0.53, 0.72}}

We use the same Manipulate function as above, but now the object being "manipulated" is a table of 4 plots, instead of just a single plot object:

In[21]:=
```
Manipulate[
 Table[
  ParametricPlot[
   Evaluate[{x1[t], v1[t]} /. solution1],
   (* Plot both variables, evaluated using solution1 *)
   {t, 0, t1},   (* Time ranges from 0 to t1 ... and t1 is
    a variable that the user can manipulate *)
   PlotRange → pr,   (* The range plotted is called "pr" -- which
     can take the values defined two lines below... *)
   AspectRatio → 1  (* Ensure that plots are square *)
  ], {pr, {pr1, pr2, pr3, pr4}}  (* Says that pr can take values pr1 through pr4 *)
 ], {t1, 1, 100}]  (* Lets user manipulate the value of t1, from 10 to 100 *)
```



Out[21]=



Using the above output, we can determine the period of oscillations around the stable equilibrium.
Expand the "t1" slider (click on the little gray "+") -- this lets you type in values for time.

Note that the system reaches the "apex" of its oscillations at the following (approximate) times:

-- 2.3
-- 20
-- 34
-- 47
-- 59.5
-- 72
-- 84.5
-- 97.5

It looks like the period of oscillation settles down to something between 12.5 and 13:

In[22]:=
```
Differences[{2.3, 20, 34, 47, 59.5, 72, 84.5, 97.5}]
```

Out[22]=
```
{17.7, 14, 13, 12.5, 12.5, 12.5, 13.}
```

3) Compute period of oscillations analytically:

We can verify this analytically by examining the Jacobian matrix at the stable equilibrium:

Define rhs = List of ODE right-hand sides:

In[23]:=
```
rhs = {eqn1[[2]], eqn2[[2]]}
```

Out[23]=
```
{r v1[t] - p v1[t] x1[t], c v1[t] - b x1[t]}
```

Define vars = List of variables (in same order as the rhs -- v1 first, then x1)

In[24]:=
```
vars = {v1[t], x1[t]}
```

Out[24]=
```
{v1[t], x1[t]}
```

Compute the Jacobian matrix using the Table function:

In[25]:=
```
jacMatrix = Table[
  D[rhs[[i]], vars[[j]]],
  (* Differentiate one of the right-hand sides with respect to one of the variables *)
  {i, 1, Length[rhs]}, (* Subscript i ranges from 1 to the
   length of "rhs" -- In this case, the length is 2. *)
  {j, 1, Length[vars]} (* Subscript j ranges from 1 to the length
   of "vars" -- In this case, the length is 2. *)
 ]
```

Out[25]=
```
{{r - p x1[t], -p v1[t]}, {c, -b}}
```

Display the Jacobian matrix in a form more pleasing to the eye:

In[26]:=
```
jacMatrix // MatrixForm
```

Out[26]//MatrixForm=

$$\begin{pmatrix} r - p\,x1[t] & -p\,v1[t] \\ c & -b \end{pmatrix}$$

Evaluate the Jacobian matrix using the parameters and the stable equilibrium, and then compute the eigenvalues / eigenvectors:

In[27]:=
```
jac1 = jacMatrix /. equilibrium1[[1]];
(* This is the jacobian matrix at the stable equilibrium, for any parameter values *)
jac1 // MatrixForm
```

Out[28]//MatrixForm=

$$\begin{pmatrix} 0 & -\frac{b\,r}{c} \\ c & -b \end{pmatrix}$$

In[29]:=
```
eigvals1 = Eigenvalues[jac1]
```

Out[29]= $\left\{\frac{1}{2}\left(-b-\sqrt{b}\,\sqrt{b-4\,r}\right),\ \frac{1}{2}\left(-b+\sqrt{b}\,\sqrt{b-4\,r}\right)\right\}$

In[30]:=
```
eigvals1a = eigvals1 /. params
```

Out[30]= {-0.05 - 0.497494 i, -0.05 + 0.497494 i}

The eigenvalues are complex conjugate, with negative real part (as they must be, since we have damped oscillations). The magnitude of the imaginary part is the frequency of oscillation.

Period = 2π / (Magnitude of Imag. part)

In[31]:=
```
frequency = Abs[Im[eigvals1a[[1]]]]
```

Out[31]= 0.497494

In[32]:=
```
period = 2 * Pi / frequency
```

Out[32]= 12.6297

This period of oscillation agrees with our graphical estimate above.

To determine the parameter values that are required to ensure that we have damped oscillations to a stable equilibrium, we need to ensure that the eigenvalues are complex conjugate with negative real part. Examine the expression "eigvals1"...

In[33]:=
```
Expand[eigvals1[[1]]]
```

Out[33]= $-\frac{b}{2}-\frac{1}{2}\sqrt{b}\,\sqrt{b-4\,r}$

So, we need:

$b > 0$ (that is, immune responses decay in the absence of stimulation)

$b < 4r$ (that is, viral growth rate must be more than 25% of the immune decay rate)

Intuitively, if the immune response decay rate (b) is very fast (greater than $4r$), then the immune system reacts quickly to a decrease in virus. This fast reaction would prevent the system from "over-shooting" the equilibrium, and we would not see oscillations.

In[34]:=

4) Visualize the effect of changing parameter *b* with the Manipulate function:

In[35]:=
```
params2 = {r → 2.5, p → 2, c → 0.2} (* Parameters excluding b *)
```

Out[35]=
```
{r → 2.5, p → 2, c → 0.2}
```

In[36]:=
```
{eqn1, eqn2, init1, init2} /. params2
```

Out[36]=
```
{v1′[t] == 2.5 v1[t] - 2 v1[t] x1[t], x1′[t] == 0.2 v1[t] - b x1[t], v1[0] == 0.1, x1[0] == 0.}
```

The following syntax defines system2 as a function of *b*. The ":=" operator tells Mathematica to re-evaluate the right-hand side anew, each time "system2" is evaluated (in other words, "system2" behaves like a function, rather than as a static value). The "_" tells Mathematica that "b" is just a dummy variable, and that any variable can be used as an argument for the function.

In[37]:=
```
system2[b_] :=
  {v1′[t] == 2.5 v1[t] - 2 v1[t] x1[t], x1′[t] == 0.2 v1[t] - b x1[t], v1[0] == 0.1, x1[0] == 0.}
```

In[38]:=
```
solution2[b_] := NDSolve[system2[b], {x1, v1}, {t, 0, 100}]
```

By evaluating solution2 at a particular value of *b*, we get the appropriate Interpolating Function:

In[39]:=
```
solution2[0.1]
```

Out[39]=
```
{{x1 → InterpolatingFunction[{{0., 100.}}, <>], v1 → InterpolatingFunction[{{0., 100.}}, <>]}}
```

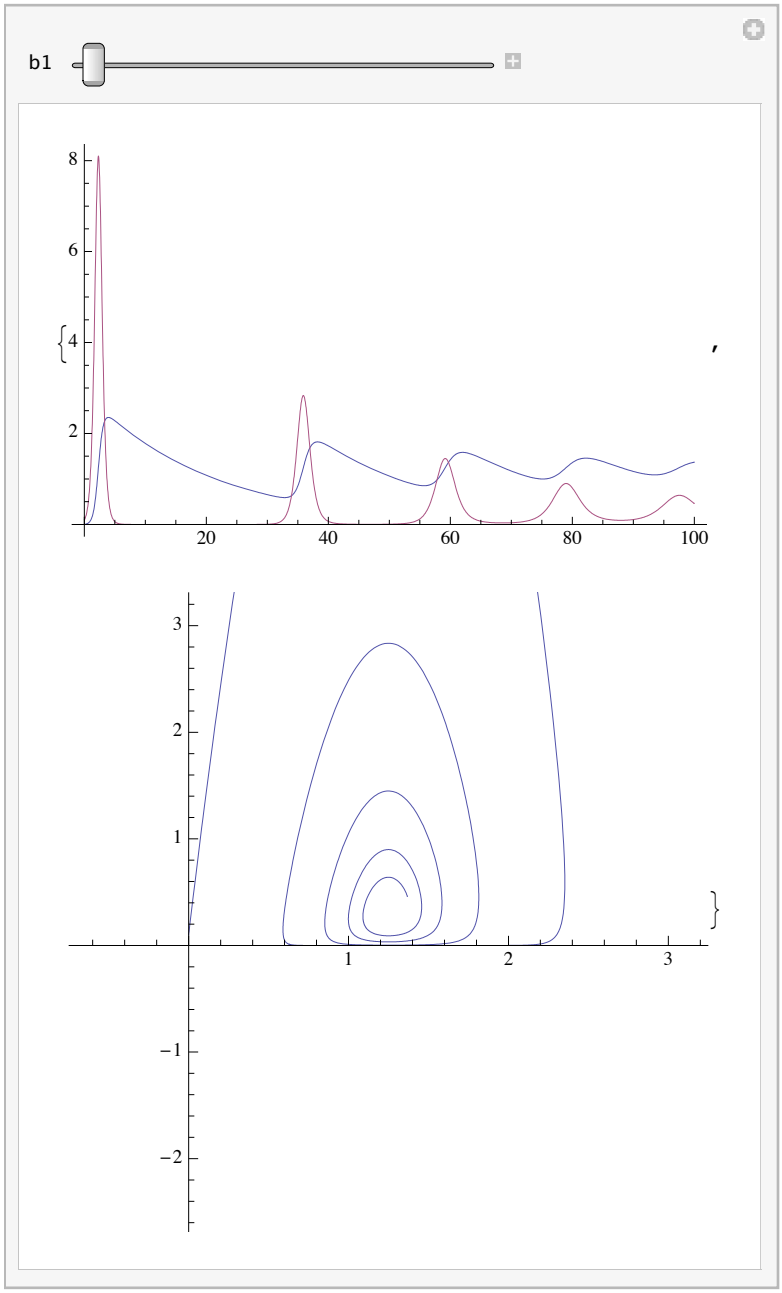
Define vEq = Equilibrium value of virus load, given *b* as input.
Define xEq = Equilibrium value of immune response (doesn't depend upon *b*)
Define pr = Desired plot range, given *b* as input (range will be 4 x 12 window around the equilibrium)

In[40]:=
```
vEq[b_] := (b * r) / (c * p)  /. params2
xEq = (r / p) /. params2;
pr[b_] := {{xEq - 2, xEq + 2}, {vEq[b] - 3, vEq[b] + 3}}
```

Output plots for a range of *b*. As *b* approaches 4r (= 10), the oscillations become looser (higher period), and at *b* = 10, the system approaches the equilibrium at a straight shot, without overshooting.

In[43]:=
```
Manipulate[
 {
  Plot[
   Evaluate[{x1[t], v1[t]} /. solution2[b1]],
   (* Plot both variables, evaluated using solution2, setting b = b1 *)
   {t, 0, 100},  (* Time ranges from 0 to 100 *)
   PlotRange → All,  (* Display entire plot *)
   ImageSize → 300 (* Set image size to 300x300 pixels *)
  ],
  ParametricPlot[
   Evaluate[{x1[t], v1[t]} /. solution2[b1]],
   (* Plot both variables, evaluated using solution2, setting b = b1 *)
   {t, 0, 100},  (* Time ranges from 0 to 100 *)
   PlotRange → pr[b1],
   (* Display the plot range defined above, centered around the equilibrium *)
   AspectRatio → 1,  (* Ensure that the plot is square *)
   ImageSize → 300 (* Set image size to 300x300 pixels *)
  ]
 },
 {b1, 0.05, 10}]  (* Lets user manipulate the value of b1, from 0.1 to 10 *)
```

Out[43]=



5) How to use *Mathematica* in a "Run-it-and-forget-about-it" way (kick off a run, have it output to a file, come back later when it's done):

Say that, for a whole range of values *b*, you want to output the period of oscillations. Here, we set up a For loop that iterates through many values, calculates period using the eigenvalue, and outputs the period to a file.

Define function that calculates period in terms of b, given our other parameters:

In[44]:=
```
params2 = {r → 2.5, p → 2, c → 0.2} (* Parameters excluding b *)
```

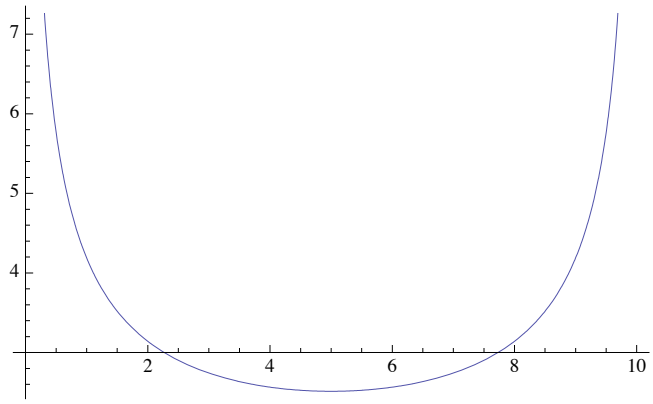
Out[44]=
```
{r → 2.5, p → 2, c → 0.2}
```

In[45]:=
```
periodfunction[b1_] := Evaluate[ 2 * Pi / Abs[Im[(eigvals1 /. params2)[[1]]]] /. {b → b1}]
```

Of course, we could just have *Mathematica* output a graph displaying how the period depends on *b* ....

In[46]:=
```
Plot[periodfunction[b], {b, 0, 10}]
```

Out[46]=



... But suppose that we wanted to output a table of values to a file, for later use. Use the following code:

In[47]:=
```
filename = ToFileName[Directory[], "math153_chap10_output1.txt"]
 (* Define a filename in the working directory. *)
```

Out[47]=
```
/Users/danielrosenbloom/math153_chap10_output1.txt
```

In[48]:=
```
str = OpenWrite[filename]; (* Creates a file with the above filename;
opens the file for writing.  If the file already exists, deletes it. *)

starttime = AbsoluteTime[]; (* starttime is set equal to the current time,
in seconds since January 1, 1900. *)

For[
 b = 0.001, (* Start at b = 0.001 *)
 b < 9.99999, (* Stop before b reaches 10.
  (Simply writing b < 10 leads to an odd floating-point error.  I'm not sure why.) *)
 b = b * 10 ^ (1 / 100 000), (* Increment b geometrically,
 going through 400,000 values in all *)
 WriteString[str, b, "\t", periodfunction[b], "\n"]
]

Print["Runtime is ", AbsoluteTime[] - starttime, " seconds."]

Close[str]; (* Closes the file stream *)
```

```
Runtime is 26.194677 seconds.
```

Optional - How to correct "weird" trajectories:

Sometimes -- especially if you are plotting a system for many, many timesteps -- you will encounter "weird" results: trajectories that cross, populations that become negative or shoot to infinity, and so on.

There are generally two types of problems:
1) The plotting function that you're using (Plot, ParametricPlot, etc.) is interpolating between too few points, failing to construct a smooth curve.
2) NDSolve is not plotting enough steps

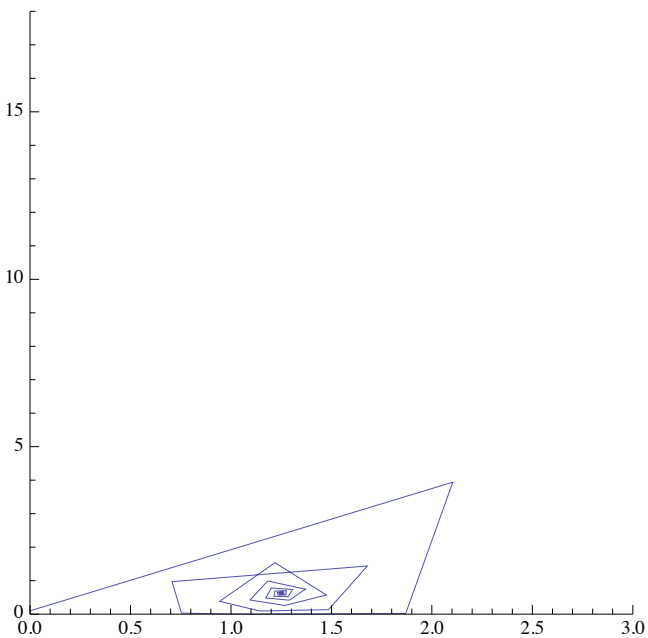
For an example of the first problem, here is a plot of the original ODE system, for 10,000 timesteps:

In[53]:=
```
solution1a = NDSolve[system1, {x1, v1}, {t, 0, 10 000}]
```

Out[53]=
```
{{x1 → InterpolatingFunction[{{0., 10 000.}}, <>],
  v1 → InterpolatingFunction[{{0., 10 000.}}, <>]}}
```

In[54]:=
```
ParametricPlot[
 Evaluate[{x1[t], v1[t]} /. solution1a],
 (* Plot both variables, evaluated using solution1a *)
 {t, 0, 10 000},   (* Time ranges from 0 to 10000 *)
 PlotRange → {{0, 3}, {0, 18}},   (* Horizontal axis from 0 to 3, Vertical from 0 to 18 *)
 AspectRatio → 1  (* Ensure that the plot is square *)
]
```

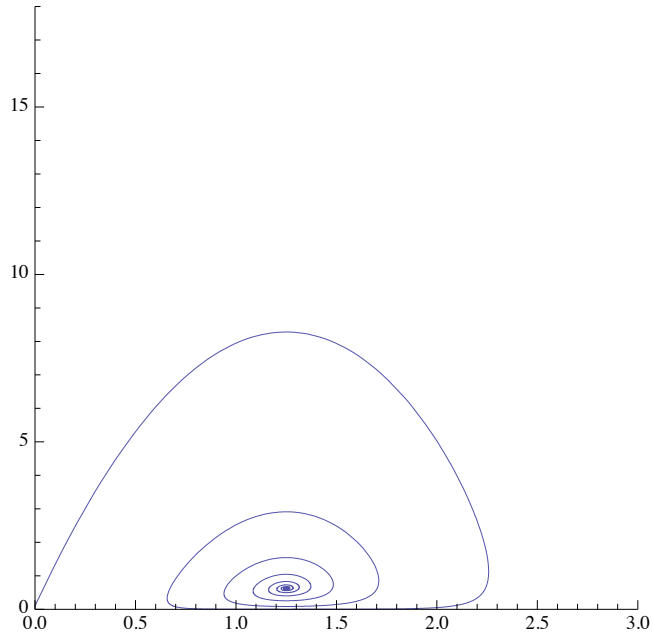
Out[54]=



The plot functions create curves by plotting a number of initial points, and then recursively calculating the values of intermediate points. If we increase the number of initial points using the "PlotPoints" setting, then we get a nice trajectory. Notice that it now takes slightly longer to render this plot:

In[55]:=
```
ParametricPlot[
 Evaluate[{x1[t], v1[t]} /. solution1a],
 (* Plot both variables, evaluated using solution1a *)
 {t, 0, 10 000},  (* Time ranges from 0 to 10000 *)
 PlotRange → {{0, 3}, {0, 18}},  (* Horizontal axis from 0 to 3, Vertical from 0 to 18 *)
 AspectRatio → 1,  (* Ensure that the plot is square *)
 PlotPoints → 10 000 (* Plot more points *)
]
```

Out[55]=



For an example of the second problem, consider the following ODE system, which has an elliptical trajectory:

In[56]:=
```
system3 = {
  D[x[t], t] == 2 * y[t],
  D[y[t], t] == -x[t],
  x[0] == 1.5,
  y[0] == 0
 }
```

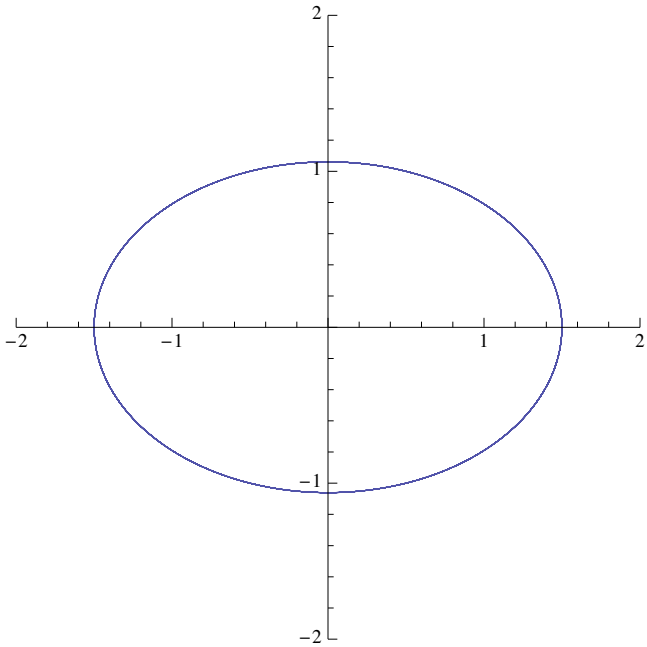
Out[56]= {x′[t] == 2 y[t], y′[t] == -x[t], x[0] == 1.5, y[0] == 0}

In[57]:=
```
solution3 = NDSolve[system3, {x, y}, {t, 0, 100}]
```

Out[57]= {{x → InterpolatingFunction[{{0., 100.}}, <>], y → InterpolatingFunction[{{0., 100.}}, <>]}}

In[58]:=
```
ParametricPlot[
 Evaluate[{x[t], y[t]} /. solution3],  (* Plot both variables, evaluated using solution3 *)
 {t, 0, 100},  (* Time ranges from 0 to 100 *)
 PlotRange → 2,  (* Both axes from -2 to 2 *)
 AspectRatio → 1  (* Ensure that the plot is square *)
]
```

Out[58]=



If we integrate the system for 1,000 timesteps, we run into a problem...

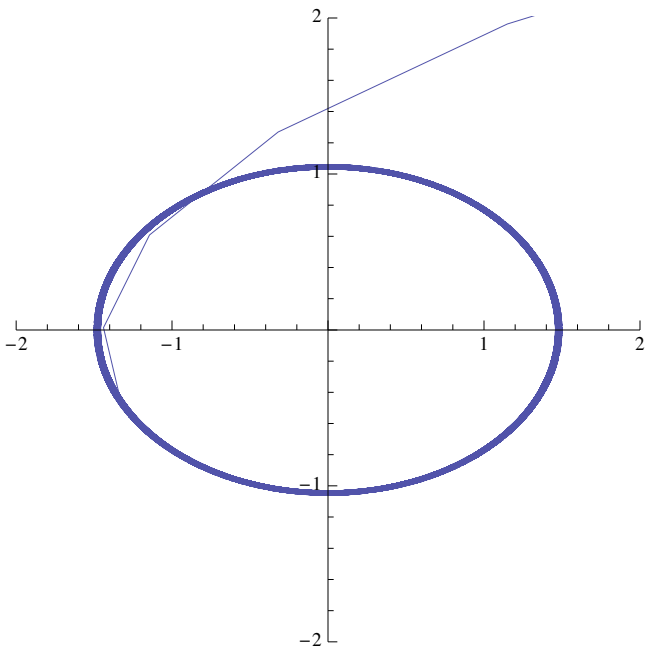
In[59]:=
```
solution3a = NDSolve[system3, {x, y}, {t, 0, 1000}]
```

NDSolve::mxst : Maximum number of 10000 steps reached at the point t == 907.7896262743865`. ≫

Out[59]=
```
{{x → InterpolatingFunction[{{0., 907.79}}, <>], y → InterpolatingFunction[{{0., 907.79}}, <>]}}
```

In[60]:=
```
ParametricPlot[
 Evaluate[{x[t], y[t]} /. solution3a],
 (* Plot both variables, evaluated using solution3a *)
 {t, 0, 1000},   (* Time ranges from 0 to 1000 *)
 PlotRange → 2,   (* Both axes from -2 to 2 *)
 AspectRatio → 1  (* Ensure that the plot is square *)
]
```

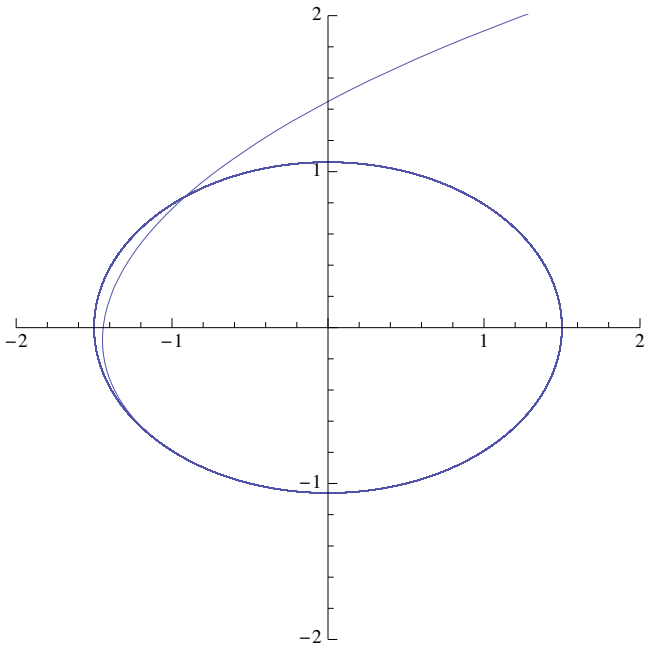
Out[60]=



... but this problem can't be solved merely by increasing the number of initial points plotted in ParametricPlot:

In[61]:=
```
ParametricPlot[
 Evaluate[{x[t], y[t]} /. solution3a],
 (* Plot both variables, evaluated using solution3a *)
 {t, 0, 1000},   (* Time ranges from 0 to 1000 *)
 PlotRange → 2,   (* Both axes from -2 to 2 *)
 AspectRatio → 1 ,  (* Ensure that the plot is square *)
 PlotPoints → 500
]
```

Out[61]=



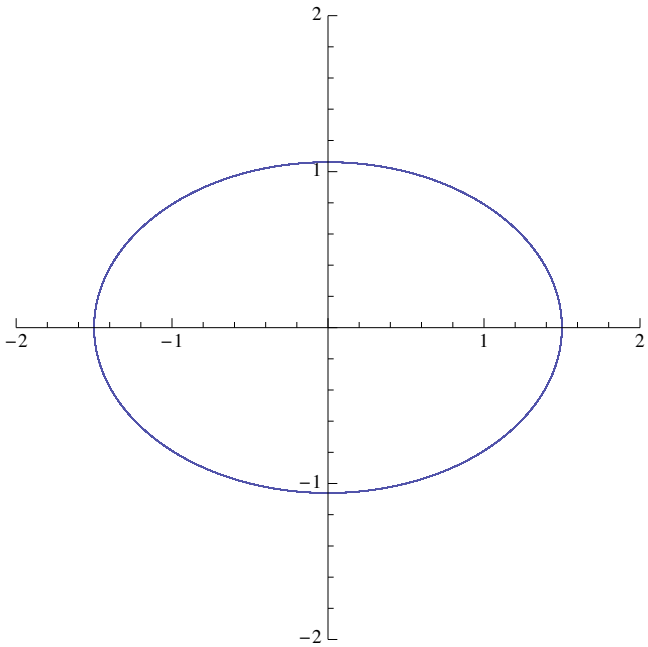
The NDSolve function stopped integrating at t = 907.7896. After that point in time, NDSolve just extrapolates naively, sending our trajectory out to infinity (though we only see the range from -2 to +2 in the plot above).

We need to increase the total number of steps computed, using the MaxSteps setting (default = 10,000, below I increase it to 100,000):

In[62]:=
```
solution3b = NDSolve[system3, {x, y}, {t, 0, 1000}, MaxSteps → 100 000]
```

Out[62]=
```
{{x → InterpolatingFunction[{{0., 1000.}}, <>], y → InterpolatingFunction[{{0., 1000.}}, <>]}}
```

And it works!

In[63]:=
```
ParametricPlot[
 Evaluate[{x[t], y[t]} /. solution3b],
 (* Plot both variables, evaluated using solution3b *)
 {t, 0, 1000},  (* Time ranges from 0 to 1000 *)
 PlotRange → 2,  (* Horizontal axis from 0 to 3, Vertical from 0 to 18 *)
 AspectRatio → 1 ,  (* Ensure that the plot is square *)
 PlotPoints → 500
]
```

Out[63]=