1 Intro to IVP and BVP

Initial Value Problems

An initial value problem is a differential equation with a set of constraints at the initial point. An IVP may look something like this

$$\begin{aligned} y'' + y' + y &= f(t) \\ y(a) &= \alpha \\ y'(a) &= \beta \\ t &\in [a, b]. \end{aligned}$$

This problem gives a differential equation with initial conditions for y and y'.

Formulating and solving initial value problems is an important tool when solving many types of problems. One simple example of an IVP would be a differential equation modeling the path of a ball thrown in the air where the initial position ($y(a)$) and velocity ($y'(a)$) are known. These problems can be tricky to solve by hand. Luckily, SciPy has great tools that help us solve initial value problems for most systems of first order ODEs. We will be using `solve_ivp` from `scipy.integrate`.

Consider the following example

$$y'' + 3y = sin(t), \quad y(0) = -\pi/2, \quad y'(0) = \pi, \quad t \in [0, 5]$$

We begin by changing this second order ODE into a first order ODE system.

$$\text{Let } y_1 = y \text{ and } y_2 = y' \text{ so that,}$$
$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}' = \begin{bmatrix} y_2 \\ \sin t - 3y_1 \end{bmatrix}.$$

This formulation allows us to use `solve_ivp`. We need three code elements in order to implement `solve_ivp`:

1. the ODE function:
   This function defines the ODE system by returning an array containing our first order system of ODES.

2. the time domain:
   This is a tuple giving the interval of integration.

3. the initial conditions:
   This is an array containing the initial conditions starting with the "zeroeth" derivative constraint, followed by the first derivative constraint, and so on if there are other constraints of higher order derivatives.

The following code is for the example given above.

```
from scipy.integrate import solve_ivp
import numpy as np

# element 1: the ODE function
def ode(t, y):
    '''defines the ode system'''
    return np.array([y[1],np.sin(t)-3*y[0]])

# element 2: the time domain
t_span = (0,5)

# element 3: the initial conditions
y0 = np.array([-np.pi /2, np.pi])

# solve the system
# max_step is an optional parameter that controls maximum step size and
# a smaller value will result in a smoother graph
sol = solve_iv
```

Then we can plot the solution with the following code

```
from matplotlib import pyplot as plt

plt.plot(sol.t,sol.y[0])
plt.xlabel('t')
plt.ylabel('y(t)')
plt.show()
```

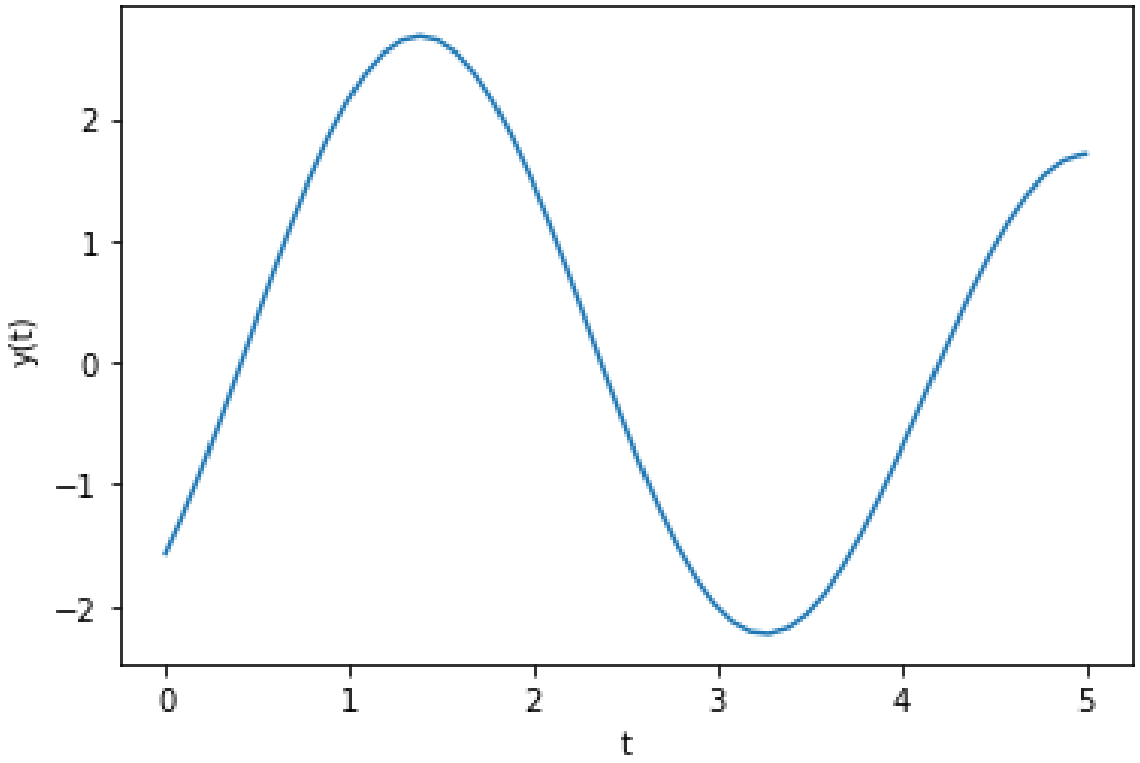

Figure 1.1: The solution to the above example

Problem 1. Use `solve_ivp` to solve for y in the equation $y'' - y = sin(t)$ with initial conditions $y(0) = -\frac{1}{2}$, $y'(0) = 0$ and plot your solution on the interval $[0, 5]$. Compare this to the analytic solution $y = -\frac{1}{2}(e^{-t} + sin(t))$.

Note: Using `max_step` $= 0.1$ with give you the smoother graph seen in figure 1.2.

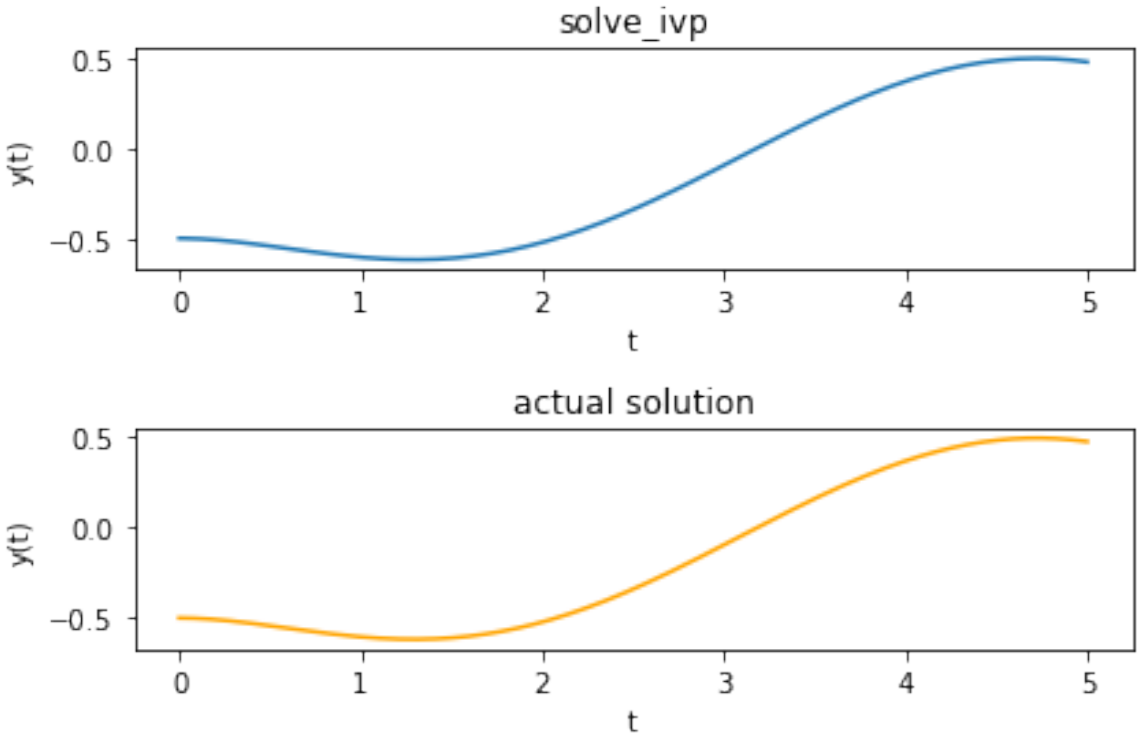

Figure 1.2: The solution to problem 1

Boundary Value Problems

A boundary value problem is a differential equation with a set of constraints. It is similar to initial value problems, but may give end constraints as well as initial constraints. A boundary value problem may look something like this

$$\begin{aligned} y'' + y' + y &= f(t) \\ y(a) &= \alpha \\ y(b) &= \beta \\ t &\in [a, b], \end{aligned}$$

where we have both right and left hand boundary conditions on y. One simple example of an IVP would be a differential equation modeling the path of a ball thrown in the air where the initial position ($y(a)$) and final position ($y(b)$).

SciPy has great tools that help us solve boundary value problems. We will be using `solve_bvp` from `scipy.integrate`. Consider the following example:

$$y'' + 9y = cos(t), \quad y'(0) = 5, \quad y(\pi) = -\frac{5}{3}. \tag{1.1}$$

We begin by changing this second order ODE into a first order ODE system.

$$\text{Let } y_1 = y \text{ and } y_2 = y' \text{ so that,}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}' = \begin{bmatrix} y_2 \\ \cos t - 9y_1 \end{bmatrix}.$$

This formulation allows us to use `solve_bvp`. It is important to notice that there are several key differences between `solve_ivp` and `solve_bvp`. We need four code elements in order to implement `solve_bvp`:

1. the ODE function:
   This is the same function we used in `solve_ivp`.

2. the boundary condition function:
   Instead of just having a tuple containing our initial values, we now must use a function that returns an array of the residuals of the boundary conditions. We pass in 2 arrays: `ya`, representing the initial values, and `yb`, representing the final values. The ith entry of those arrays represents the boundary condition at the ith derivative. `ya[0]-x` would indicate that we know `y(a)=x`, `ya[1]-x` would indicate that we know `y`(a)=x`, `yb[0]-x` would indicate that we know `y(b)=x`, or `yb[1]-x` would indicate that we know `y`(b)=x`,

3. the time domain:
   Instead of a tuple giving the interval of integration, we now must pass in a linspace from the starting time to the ending time, containing the desired number of points (we now must choose the number).

4. the initial guess:
   As we no longer know all of the initial values, we now must make an educated guess. This is an array of shape (`n`,`t_steps`) where `n` is the shape of the output of the ODE function and `t_steps` is the chosen number of steps in our time domain linspace.

```
from scipy.integrate import solve_bvp
import numpy as np

# element 1: the ODE function
def ode(t,y):
    ''' define the ode system '''
    return np.array([y[1], np.cos(t) - 9*y[0]])
# element 2: the boundary condition function
def bc(ya,yb):
    ''' define the boundary conditions '''
    # ya are the initial values
    # yb are the final values
    # each entry of the return array will be set to zero
    return np.array([ya[1] - 5, yb[0] + 5/3])

# element 3: the time domain.
t_steps = 100
t = np.linspace(0,np.pi,t_steps)

# element 4: the initial guess.
y0 = np.ones((2,t_steps))

# Solve the system.
sol = solve_bvp(ode, bc, t, y0)
```

The syntax for plotting the function is also slightly different:

```
import matplotlib.pyplot as plt

# here we plot sol.x instead of sol.t
plt.plot(sol.x, sol.y[0])
plt.xlabel('t')
plt.ylabel('y(t)')
plt.show()
```

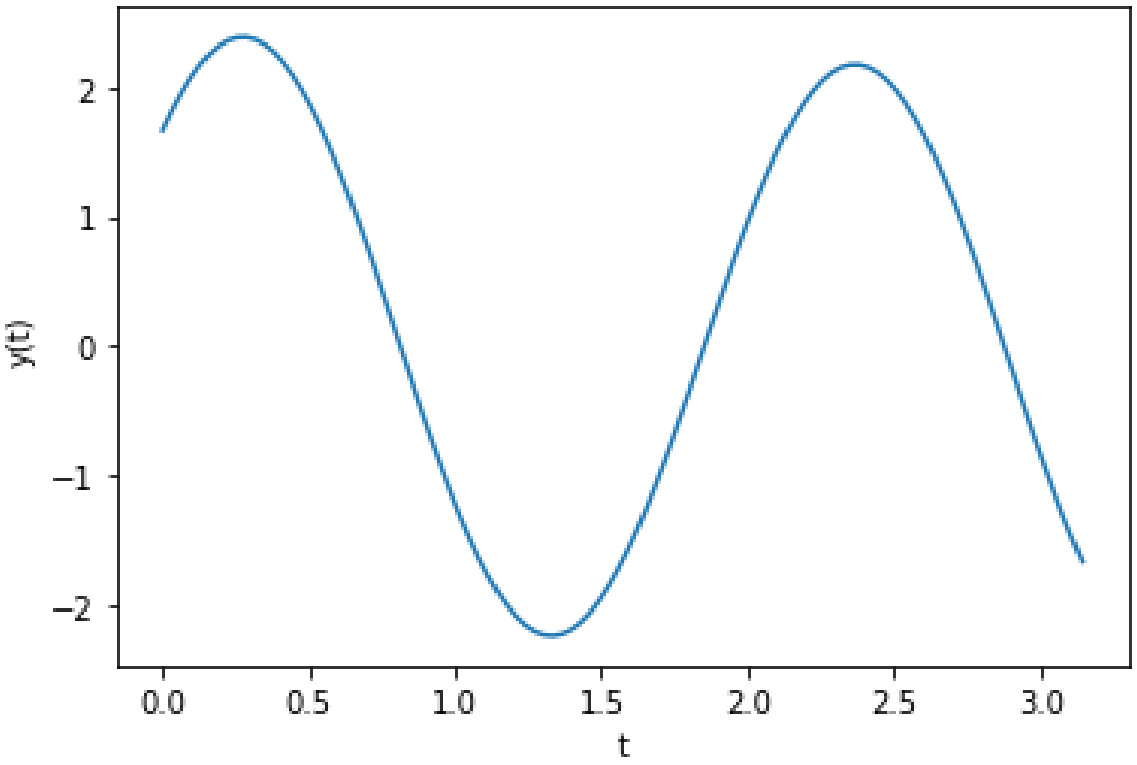

Figure 1.3: The solution to the above problem

Problem 2. Use `solve_bvp` to solve for y in the equation $y'' - y = sin(t)$ with initial conditions $y(0) = -\frac{1}{2}$, $y'(0) = 0$ and plot your solution on the interval $[0, 5]$. Compare this to the analytic solution $y = -\frac{1}{2}(e^{-t} + sin(t))$.

Note: Using `max_step` $= 0.1$ with give you the smoother graph seen in figure 1.4.

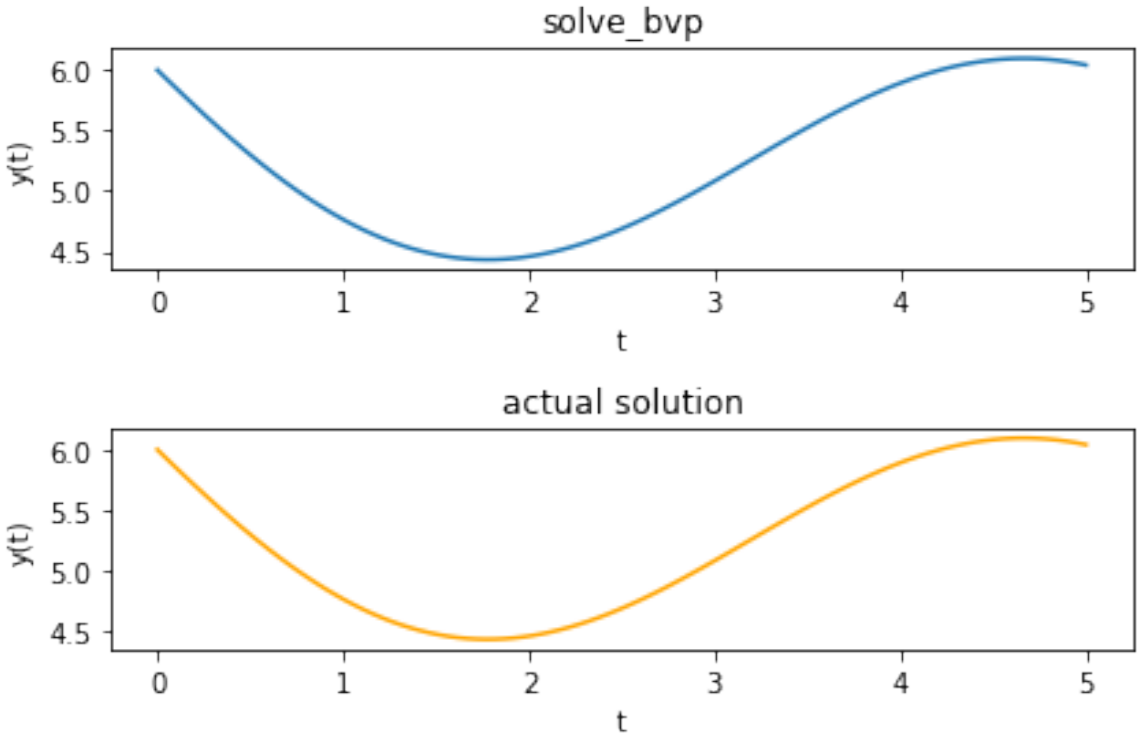

Figure 1.4: The solution to problem 2.

One other useful function of `solve_bvp`: `sol.sol` creates a callable function, which is the estimation of the boundary value problem. You can utilize it by plugging in any int or linspace (`sol.sol(np.linspace)`, `sol.sol(int)`, `sol.sol(list)`), like a normal lambda function.

The Pitfalls of `solve_bvp`

One of the common issues with `solve_bvp` is choosing a guess for the initial value. Often, small changes in the guess can cause large changes in the final approximation. The next problem demonstrates the huge difference that can be made between an initial guess of 10 and an initial guess of 9.99

Problem 3. Use `solve_bvp` to solve for y in the equation $y'' = (1 - y') * 10y$ with boundary conditions $y(0) = -1$ and $y(1) = \frac{3}{2}$ and plot your solution on the interval $[0, 1]$. Use an initial guess of 10. Compare this to the the same solution using an initial guess of 9.99.

The solution is found in figure 1.5.

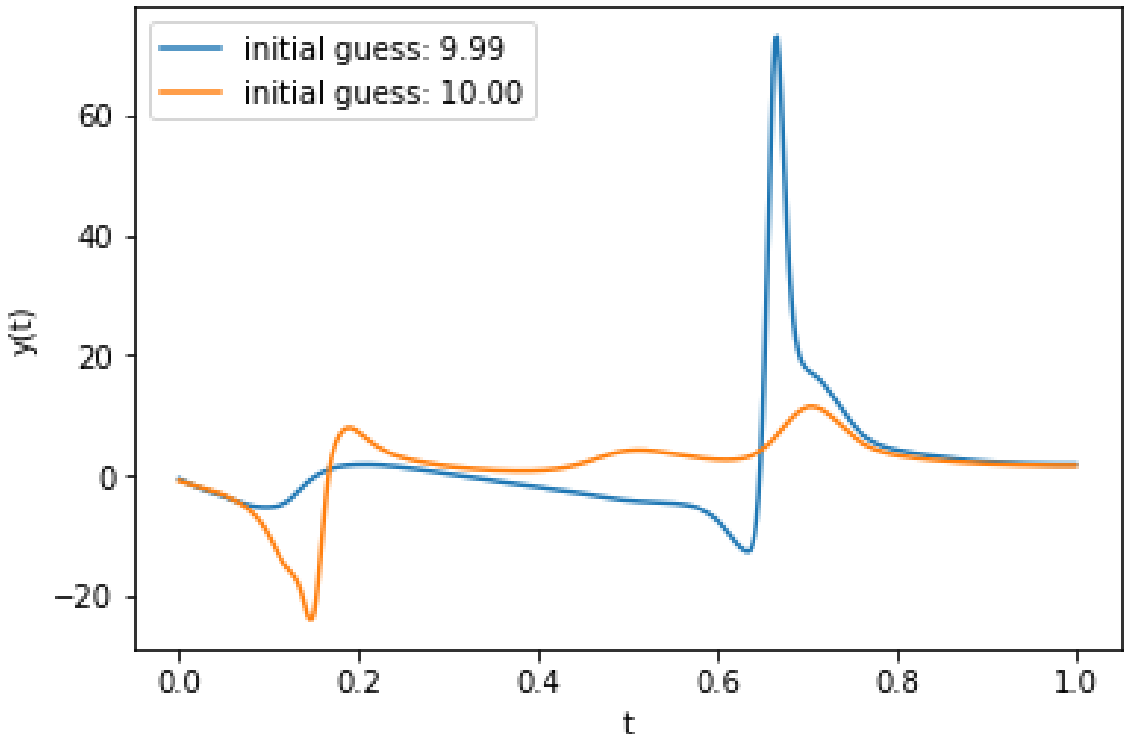

Figure 1.5: The solution to problem 3.

`solve_ivp` and Strange Attractors

In the growing field of dynamical systems, an attractor is a set of states toward which a system tends to evolve. Strange Attractors are special in that they showcase complex behavior in a simple set of equations. A minute change in the initial values can cause massive discrepancies in the outcome. The most famous of these is the Lorenz Attractor, introduced by Edward Lorenz in 1963. Later on, we dedicate a full lab to the study of the Lorenz Attractor, but today we focus on modeling the Four-Wing attractor. This is a system of first order ODEs defined by the following set of equations

$$\frac{dx}{dt} = ax + yz \tag{1.2}$$
$$\frac{dy}{dt} = bx + cy - xz \tag{1.3}$$
$$\frac{dz}{dt} = -z - xy \tag{1.4}$$

given some constants a, b, and c. As we mentioned earlier, `solve_ivp` and `solve_bvp` can be used to solve and model systems of first order ODEs. We will now use `solve_ivp` to model a Four-Wing attractor.

Problem 4. Use `solve_ivp` to solve the Four-Wing Attractor as described in equations (1.2), (1.3), and (1.4) where $a = 0.2$, $b = 0.01$, and $c = -0.4$. Try this with 3 different initial values

and plot (in three dimensions) the 3 corresponding graphs.

Possible solutions are seen in Figure 1.6.

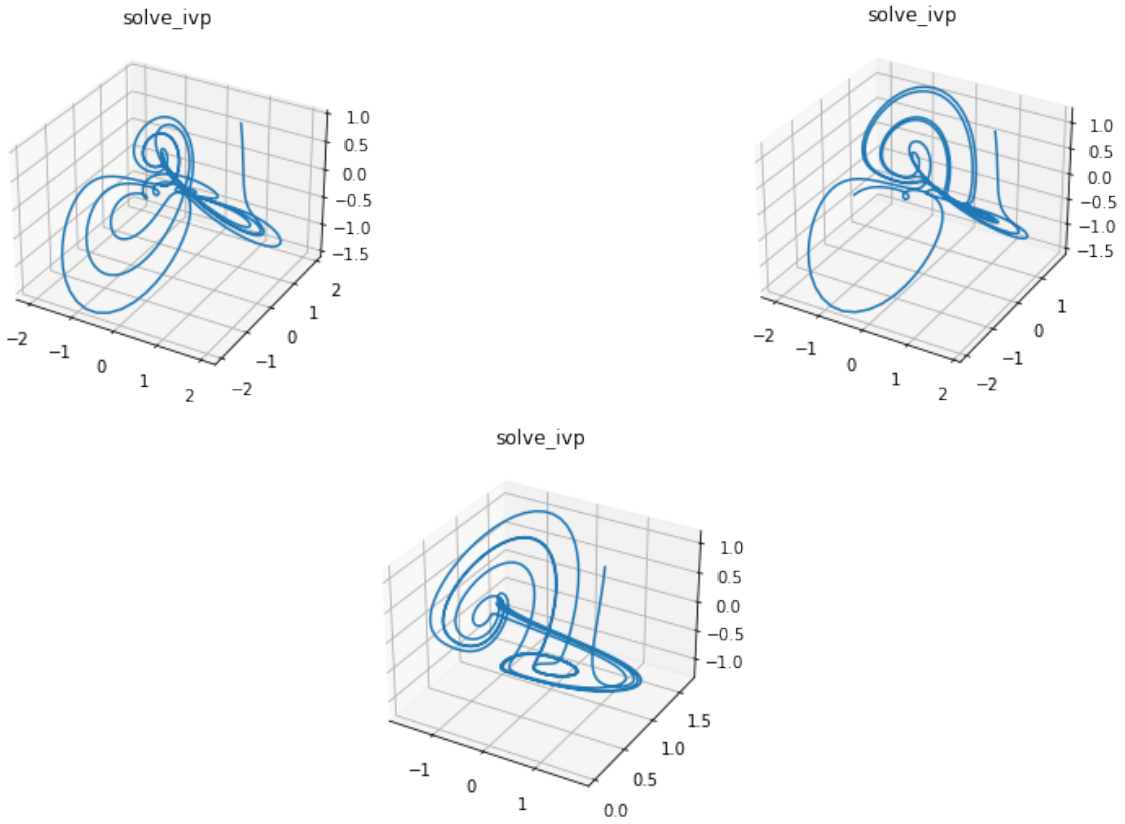

Figure 1.6: Possible solutions to problem 4.