An Introduction to Stiff ODEs in Python

July 24, 2021

1 An Introduction to Stiff ODEs in Python

Notice: In my previous post on stiff ODEs, I demonstrated how different ODE solvers in Matlab perform with a few examples. The live script for the post is also provided for educational purpose. In this post, I will do the same in Python. You can find the `ipynb` file in the same repository.

The solver interfaces provided by Matlab and SciPy are not exactly the same (SciPy uses `rtol*abs(y)+atol` while Matlab uses `max(rtol*abs(y),atol)`), so we will use different solvers and tolerances. . If you are interested, please refered to the SciPy document and the Matlab document

---

It's well-known that stiff ODEs are hard to solve. Many books are written on this topic, and SciPy even provides solvers specialized for stiff ODEs. It is easy to find resources, including the wikipedia entry, with technical and detailed explanations. For example, one of the common descriptions for stiff ODEs may read:

An ODE is stiff if absolute stability requirement is much more restrictive than accuracy requirement, and we need to be careful when we choose our ODE solver.

However, it's fairly abstract and hard to understand for people new to scientific computing. In this post, I hope to make the concept of stiffness in ODEs easier to understand by showing a few examples. Let's start with a simple (non-stiff) example, and compare it with some stiff examples later on.

1.1 Example 1

Let's consider a non-stiff ODE

where

In the first case we have . The solution is

,

meaning we have a exponential decaying function.

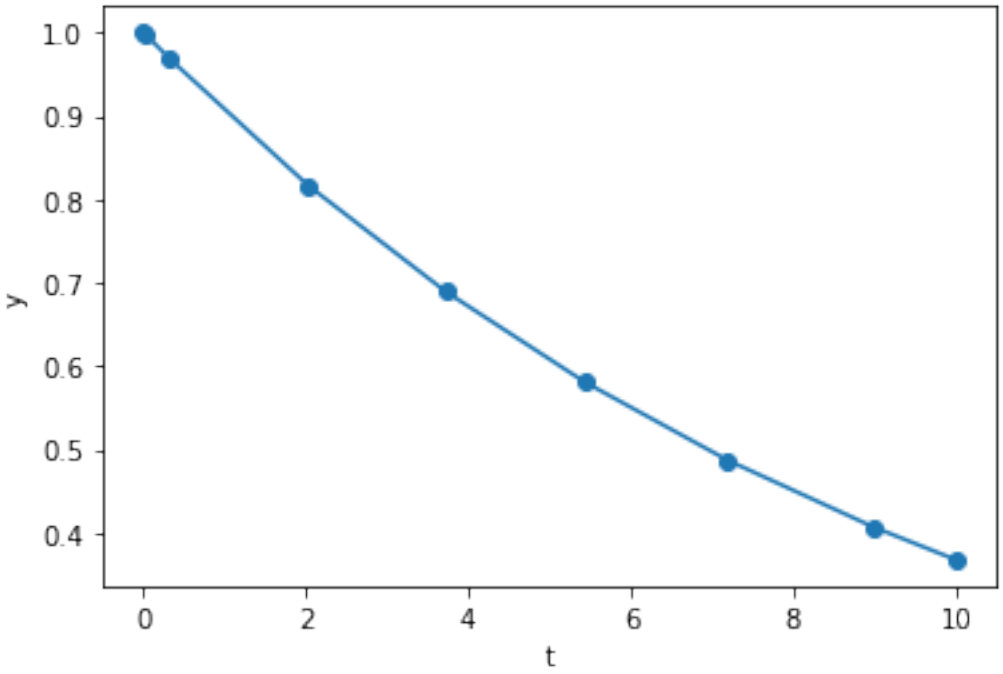
[1]:
```
import numpy as np
from scipy.integrate import solve_ivp, RK45
import matplotlib.pyplot as plt

mlambda = -1e-1
```

```
A = np.matrix([mlambda])

F = lambda t,u: A.dot(u.flatten())

# initial condition
u0 = np.ones(A.shape[0])

# time points
t = [0,10]
```

1.1.1 RK45

We can look at the solution from `RK45`:

[2]:
```
# solve ODE
sol = solve_ivp(F,t,u0,'RK45',rtol=1e-7,atol=1e-7)

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



As we can see, it `RK45` gives us a decaying function. In this interval, `RK45` used

[3]:
```
sol.t.size
```

[3]: 9

steps to achieve the specified tolerance.

1.2 Example 2

Let's consider the same equation

but now

In the first case we have . This means we have two decoupled equations. The solution is

,

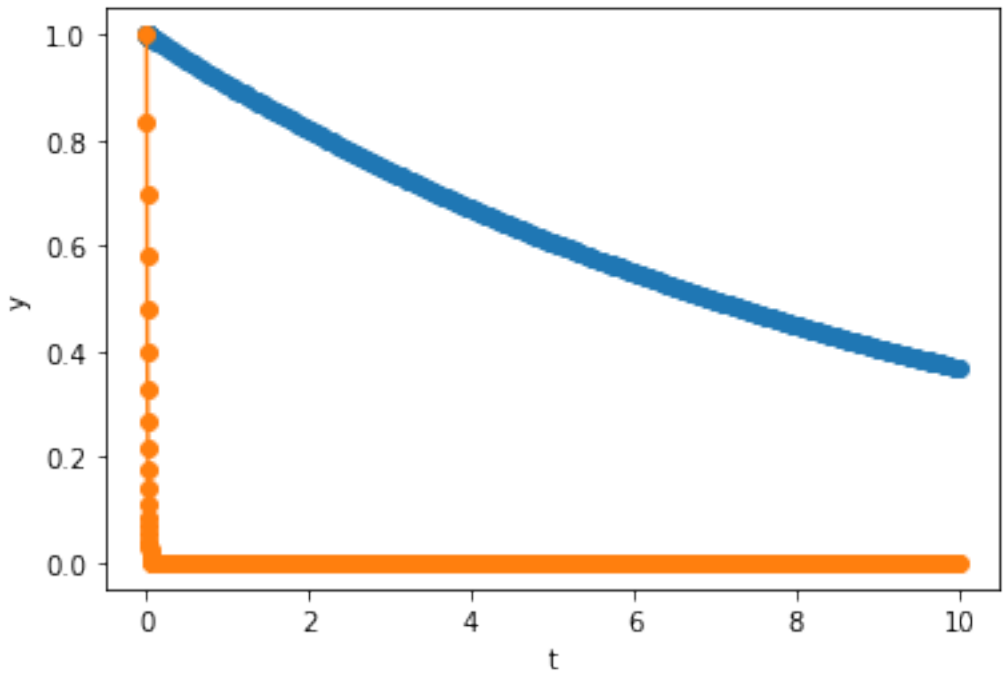
meaning we have two exponential decaying functions.

[4]:
```
mlambda1 = -1e-1
mlambda2 = 1e3*mlambda1
A = np.diag([mlambda1, mlambda2])

F = lambda t,u: A.dot(u)


# initial condition
u0 = np.ones(A.shape[0])

# time points
t = [0,10]
```

1.2.1 RK45

We can use `RK45` to solve it in the same fashion:

[5]:
```
# solve ODE
sol = solve_ivp(F,t,u0,'RK45',rtol=1e-7,atol=1e-7)

# # plot results
plt.plot(sol.t,sol.y[0],'o-',sol.t,sol.y[1],'o-')
plt.xlabel('t')
plt.ylabel('y')
plt.show()
```



This time we get 2 decaying functions, and decays much faster then . In this same interval, `RK45` used

[6]:
```
sol.t.size
```

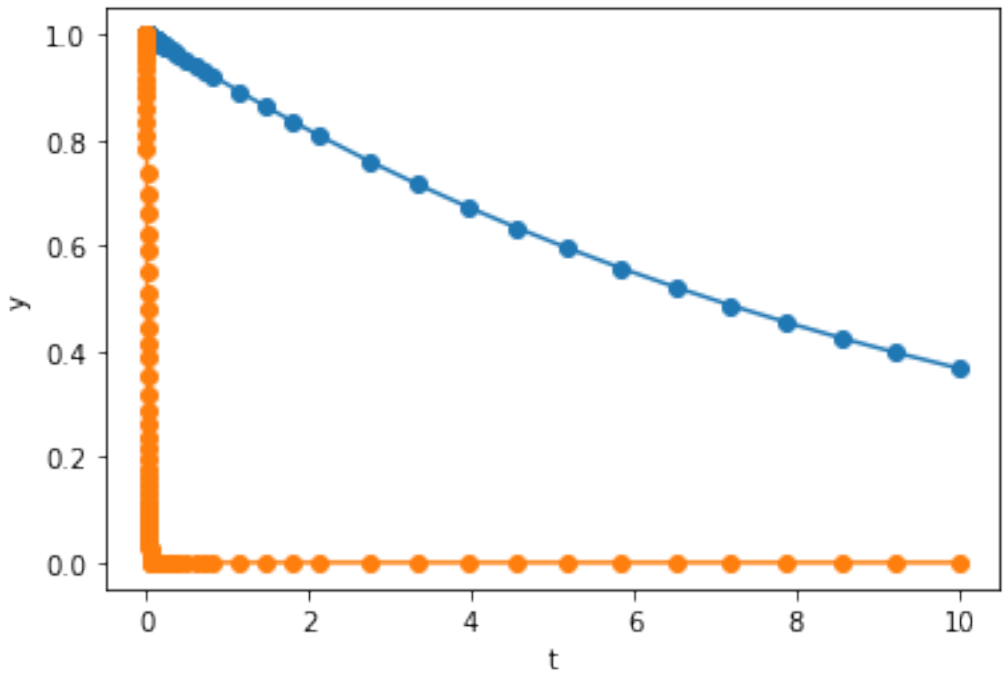
[6]: 333

steps to achieve the desired error tolerance. In this example, is exactly the same as the solution in Example 1, but it take much longer to calculate. One may think the step size of `RK45` is limited by the *accuracy requirement* due to the addition of . However, this is clearly not the case since is almost identically on the entire interval. What is happening here is that, the step size of `RK45` is limited by the *stability requirement* of , and we call the ODE in Example 2 ***stiff***.

SciPy provides specialized ODE solvers for stiff ODEs. Let's look at `BDF` and `Radau`

1.2.2 BDF

[7]:
```
# solve ODE
sol = solve_ivp(F,t,u0,'BDF',rtol=1e-7,atol=1e-7)

# # plot results
plt.plot(sol.t,sol.y[0],'o-',sol.t,sol.y[1],'o-')
plt.xlabel('t')
plt.ylabel('y')
plt.show()
```



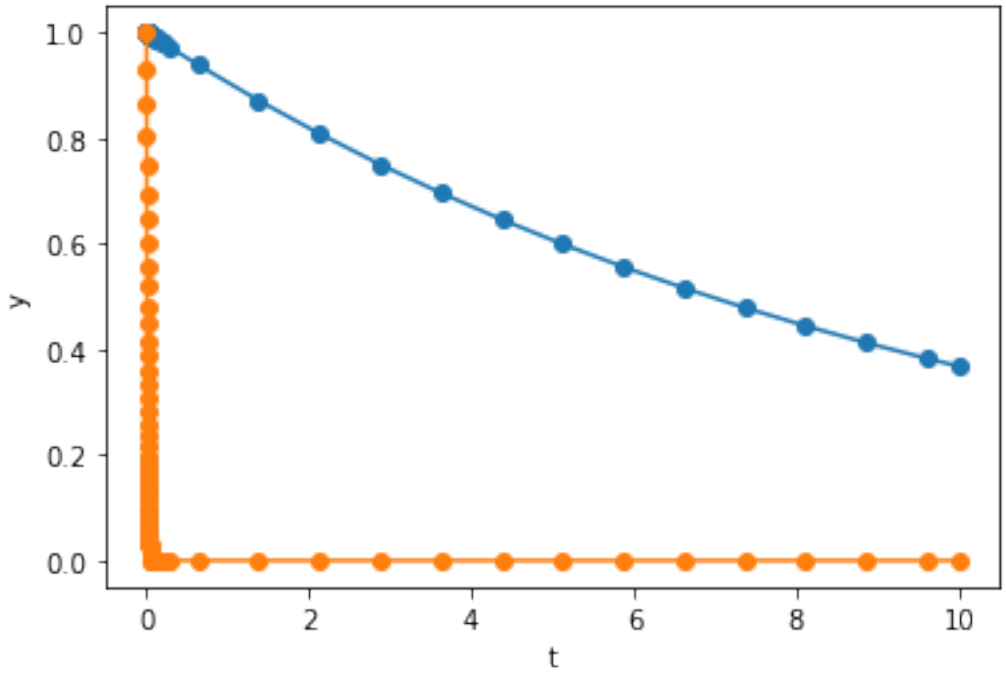
BDF takes

[8]:
```
sol.t.size
```

[8]: 134

1.2.3 Radau

[9]:
```
# solve ODE
sol = solve_ivp(F,t,u0,'Radau',rtol=1e-7,atol=1e-7)

# # plot results
plt.plot(sol.t,sol.y[0],'o-',sol.t,sol.y[1],'o-')
plt.xlabel('t')
plt.ylabel('y')
plt.show()
```



and `Radau` takes

[10]:
```
sol.t.size
```

[10]: 85

steps. Apparently, `BDF` and `Radau` is significantly more efficient than `RK45` for this example. From the figure above, we can also see that `BDF` and `Radau` stratigically used shorter step size when is decaying fast, and larger step size when flattens out.

At this point you may think that if you don't know whether an ODE is stiff or not, it is always better to use `BDF` and `Radau`. However, this is not the case, as we will show in the next example.

2 Ocsillatory ODE

2.1 Example 3

Let's look at an oscillatory ODE

and

The eigenpairs of are

The solution is oscillatory because the eigenvalues are imaginary.

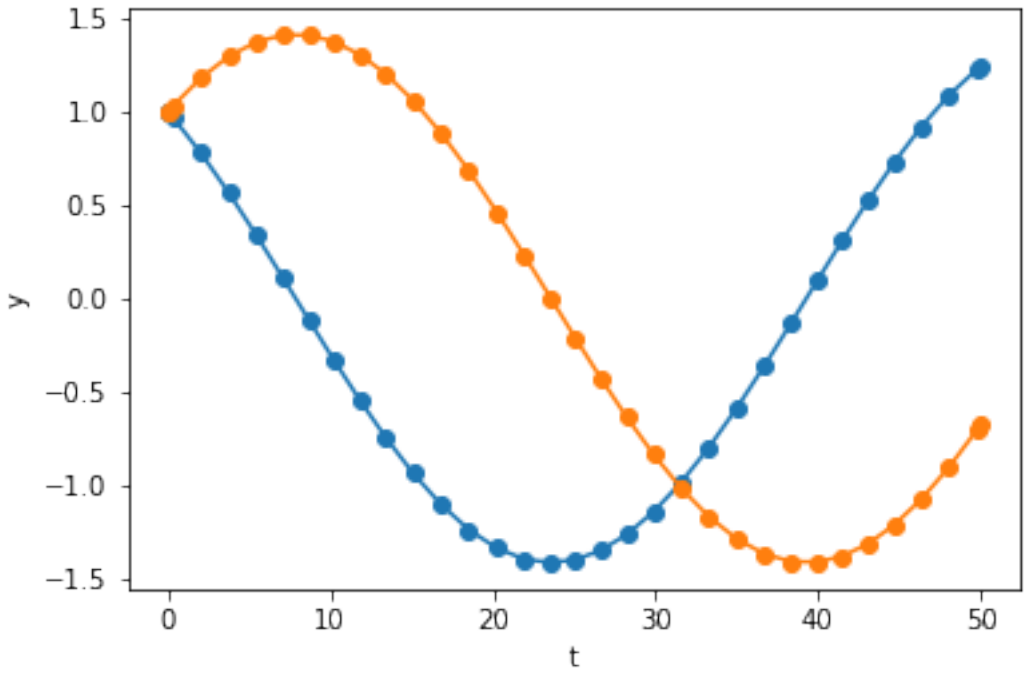
[11]:
```
mlambda = -1e-1
L = np.matrix([[ 0, mlambda],[ -mlambda, 0]])
```

```
F = lambda t,u: L.dot(u.flatten())


# initial condition
u0 = np.ones(L.shape[0])

# time points
t = [0,50]
```

2.1.1 RK45

Let's look at the solution from `RK45`

[12]:
```
# solve ODE
sol = solve_ivp(F,t,u0,'RK45',rtol=1e-7,atol=1e-7)

# # plot results
plt.plot(sol.t,sol.y[0],'o-',sol.t,sol.y[1],'o-')
plt.xlabel('t')
plt.ylabel('y')
plt.show()
```



[13]:
```
sol.t.size
```

[13]: 34

As expected, we see two slow oscillatory functions.

2.2 Example 4

Now let's look at a stiff oscillatory ODE

and

The eigenpairs of are

, and

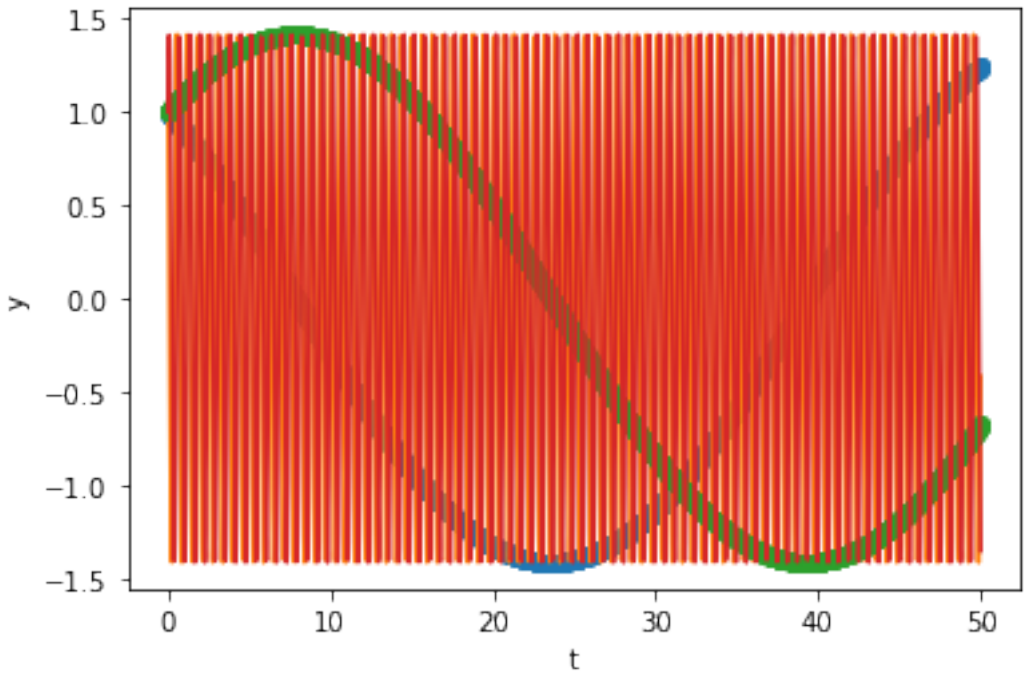
Similar to before, we set . Now we have both fast and slow oscillatory functions in our solution.

[14]:
```
mlambda1 = -1e-1
mlambda2 = 1e2*mlambda1
A = np.diag([mlambda1, mlambda2])
L = np.block([[np.zeros([2,2]),A],[-A,np.zeros([2,2])]])

F = lambda t,u: L.dot(u.flatten())


# initial condition
u0 = np.ones(L.shape[0])


# time points
t = [0,50]
```

2.2.1 RK45

[15]:
```
# solve ODE
sol = solve_ivp(F,t,u0,'RK45',rtol = 1e-6, atol = 1e-6)

# # plot results
plt.plot(sol.t,sol.y[0],'o-',sol.t,sol.y[1],'-',sol.t,sol.y[2],'o-',sol.t,sol.
 ↪y[3],'-')
plt.xlabel('t')
plt.ylabel('y')
```

[15]:
```
Text(0, 0.5, 'y')
```



In the plots we can see both slow and highly oscillatory parts. Again, similar to the decaying case, now `RK45` is taking shorter step sizes because of the the fast oscillating part, even though and could have taken much shorter time steps like the example above. In this case,

[16]:
```
sol.t.size
```

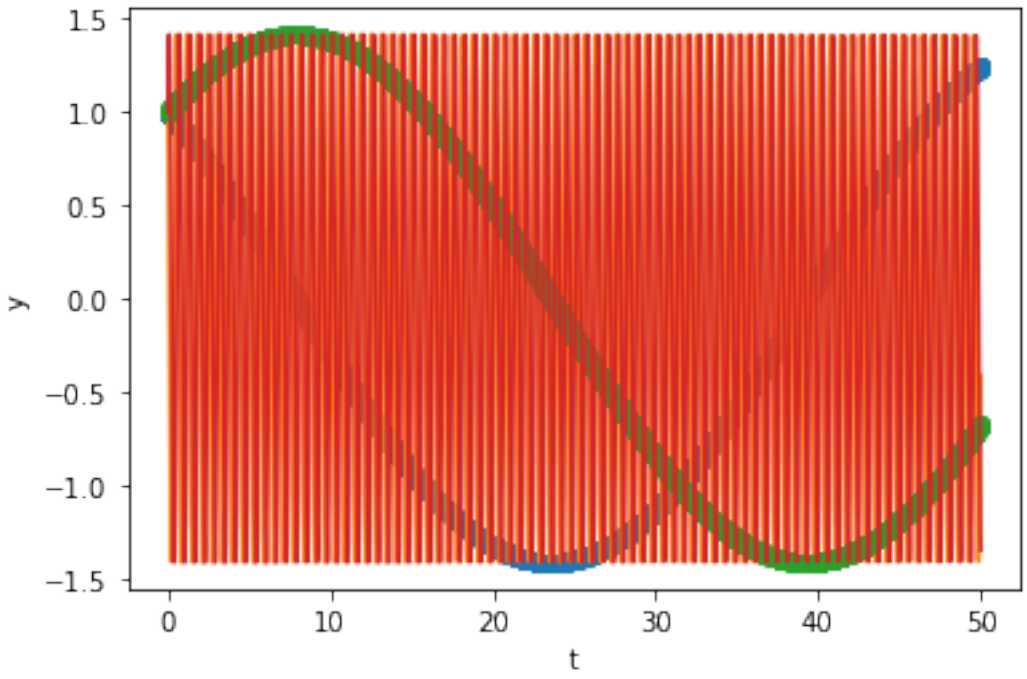
[16]:
```
1772
```

This time `BDF` and `Radau` are not that efficient either.

2.2.2 BDF

[17]:
```
# solve ODE
sol = solve_ivp(F,t,u0,'BDF',rtol = 1e-6, atol = 1e-6)

# # plot results
plt.plot(sol.t,sol.y[0],'o-',sol.t,sol.y[1],'-',sol.t,sol.y[2],'o-',sol.t,sol.
 ↪y[3],'-')
plt.xlabel('t')
plt.ylabel('y')
```

[17]:
```
Text(0, 0.5, 'y')
```

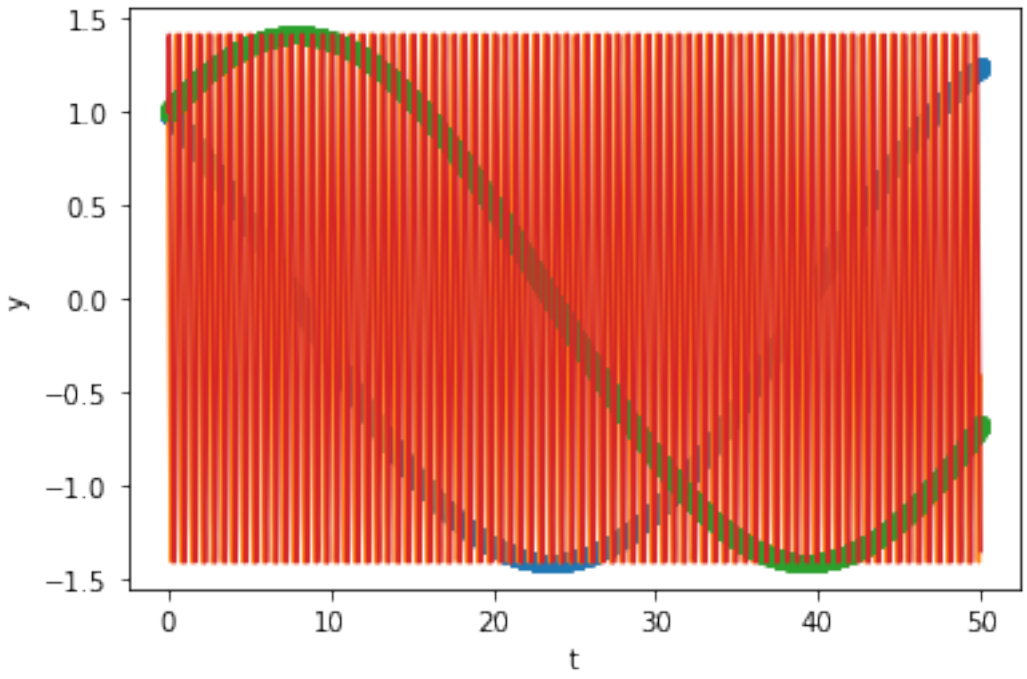


[18]:
```
sol.t.size
```

[18]: 3853

2.2.3 Radau

[19]:
```
# solve ODE
sol = solve_ivp(F,t,u0,'Radau',rtol = 1e-6, atol = 1e-6)

# # plot results
plt.plot(sol.t,sol.y[0],'o-',sol.t,sol.y[1],'-',sol.t,sol.y[2],'o-',sol.t,sol.
 ↪y[3],'-')
plt.xlabel('t')
plt.ylabel('y')
plt.show()
```



[20]:
```
sol.t.size
```

[20]: 3788

Notice highly oscillatory and stiff ODEs are generally hard to solve. All the solvers, `RK45`, `BDF`, and `Radau` take very short steps and become very expensive.

This blog post is published at https://edwinchenyj.github.io. The pdf version and the source code are available at https://github.com/edwinchenyj/scientific-computing-notes.