In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate
import time
%matplotlib notebook
# For plotting. Don't include this if you submit
                     # a Jupyter Notebook to Gradescope.
```

Stability and stiff ODEs

First, finish Activity 5.

Recall, we had just coded up three functions for solving the IVP $y'(t) = f(t, y)$, $y(0) = y_0$:

- one that solves the IVP using Forward Euler;
- one that solves the IVP using Backward Euler; and
- one that solves the IVP using the midpoint method.

Those functions are given below.

```python
In [2]: f forward_euler(odefun, tspan, y0):
          # Forward Euler method
          # Solves the differential equation y' = f(t,y) at the times
          # specified by the vector tspan and with initial condition y0.
          # - odefun is an anonymous function of the form odefun = lambda t,v: ...
          # - tspan is a 1D array
          # - y0 is a number

          dt = tspan[1] - tspan[0]  # calculate dt from t values
          y = np.zeros(len(tspan))  # Create array of same length as tspan
          y[0] = y0  # Set initial condition
          for k in range(len(y) - 1):
              y[k + 1] = y[k] + dt * odefun(tspan[k], y[k]) # Forward Euler step

          return tspan, y # return two things

        f backward_euler(odefun, tspan, y0):
          # Backward Euler method
          # Solves the differential equation y' = f(t,y) at the times
          # specified by the vector tspan and with initial condition y0.
          # - odefun is an anonymous function of the form odefun = lambda t,v: ...
          # - tspan is a 1D array
          # - y0 is a number

          dt = tspan[1] - tspan[0]  # calculate dt from t values
          y = np.zeros(len(tspan))  # Create array of same length as tspan
          y[0] = y0  # Set initial condition

          for k in range(len(y) - 1):
              g = lambda z: z - y[k] - dt*odefun(tspan[k+1], z) # Implicit equation
              y[k+1] = scipy.optimize.fsolve(g, y[k]) # Defines y[k+1]

          return tspan, y

        f midpoint(odefun, tspan, y0):
          # Midpoint method
          # Solves the differential equation y' = f(t,y) at the times
          # specified by the vector tspan and with initial condition y0.
          # - odefun is an anonymous function of the form odefun = lambda t,v: ...
          # - tspan is a 1D array
          # - y0 is a number

          dt = tspan[1] - tspan[0]  # calculate dt from t values
          y = np.zeros(len(tspan))  # Create array of same length as tspan
          y[0] = y0  # Set initial condition
          for k in range(len(y) - 1):
              k1 = odefun(tspan[k], y[k])
              k2 = odefun(tspan[k] + dt/2, y[k] + dt/2*k1)
              y[k + 1] = y[k] + dt*k2 # Forward Euler step

          return tspan, y # return two things
```

We can test these three functions on the IVP

$$y'(t) = 0.3y + t,\ \ y(0) = 1$$

using the following code.

```
In [3]: dydt = lambda t, y: 0.3*y + t # Define the ODE
        y0 = 1  # Define the IC
        tspan = np.arange(0, 1+0.1, 0.1) # The t values at which we want the solutic

        # Solve using Forward Euler, print the answer at t=1
        tans, yans = forward_euler(dydt, tspan, y0)
        print('Forward Euler gives y(1) = ', yans[-1])

        # Solve using Backward Euler, print the answer at t=1
        tans, yans = backward_euler(dydt, tspan, y0) # Backward Euler
        print('Forward Euler gives y(1) = ', yans[-1])

        # Solve using midpoint, print the answer at t=1
        tans, yans = midpoint(dydt, tspan, y0) # Backward Euler
        print('The Midpoint method gives y(1) = ', yans[-1])
```

```
Forward Euler gives y(1) =  1.8318761498343654
Forward Euler gives y(1) =  1.9790907633163404
The Midpoint method gives y(1) =  1.9031262552691077
```

We can see that all are giving similar but slightly different numbers, as expected.

We now want to compare the results for two systems. There are two things we will be thinking about

- accuracy, and
- time it takes to solve.

With that in mind, let's think about

System 1

The first IVP we will solve is

$$y(t) = y^3 - y, \; 0 \leq t \leq 0.1$$
$$y(0) = 2,$$

using $\Delta t = 0.001$. We will compare what the different methods give at $t = 0.1$ using the exact solution at $t = 0.1$ (which I calculated at another time): $y(0.1) = 3.451397662017099$ (up to truncation.

```
In [4]: # Define the ODE
        dydt = lambda t, y: y**3 - y
        # Define the IC
        y0 = 2
        # Define tspan, the times at which we want the solution
        dt = 0.001
        tspan = np.arange(0, 0.1+dt, dt)
        # Define the "exact_sol"
        exact_sol = 3.451397662017099
```

Now we will solve the IVP using the four methods and record the error. The error is defined as

$$\text{Error} = |y(0.1) - \hat{y}_N|,$$

where $\hat{y}_N$ is the approximation from our given method at the end ($t = 0.1$).

In [5]:
```
## Solve using Forward Euler
t, FE_ans = forward_euler(dydt, tspan, y0)
# Calculate the error, using exact_sol
FE_err = np.abs(exact_sol - FE_ans[-1])
print("Forward-Euler error = ", FE_err)
# Run this first to check work, and then use other methods

## Solve using Backward Euler
t, BE_ans = backward_euler(dydt, tspan, y0)
# Calculate the error, using exact_sol
BE_err = np.abs(exact_sol - BE_ans[-1])
print("Backward-Euler error = ", BE_err)

## Solve using midpoint
t, mid_ans = midpoint(dydt, tspan, y0)
# Calculate the error, using exact_sol
mid_err = np.abs(exact_sol - mid_ans[-1])
print("Midpoint error = ", mid_err)

## Solve using scipy.integrate.solve_ivp
sol = scipy.integrate.solve_ivp(dydt,
                             [tspan[0], tspan[-1]],
                             [y0])
# Extract the answer from solve_ivp
rk45 = sol.y[0]
# Calculate the error, using exact_sol
rk45_err = np.abs(exact_sol - rk45[-1])
print("RK45 error = ", rk45_err)
```

```
Forward-Euler error =  0.033306066218885544
Backward-Euler error =  0.036003231143017l8
Midpoint error =  0.00026819802845023943
RK45 error =  6.180182133874723e-05
```

We see that the error is ranked, from smallest to largest, to be: rk45<midpoint<forward<backward

This is as expected, based on the global error for each method (remember that RK45 has global error $\mathcal{O}(\Delta t^4)$, midpoint has global error $\mathcal{O}(\Delta t^2)$, and the Euler methods have global error $\mathcal{O}(\Delta t)$).

But what about speed? That's what we want to consider next. To measure the time it takes to complete the solve, we are going to use `time.perf_counter()` . For example, for Forward Euler we have:

In [6]:
```
time_0 = time.perf_counter()
t_sol, y_sol = forward_euler(dydt, tspan, y0)
time_FE = time.perf_counter() - time_0
print("The time to complete the Forward-Euler solve =", time_FE)
```

```
The time to complete the Forward-Euler solve = 0.0004119629999946007
```

We can do the same with Backward Euler, midpoint, and `scipy.integrate.solve_ivp` .

In [7]:
```python
## Time for Backward Euler
time_0 = time.perf_counter()
t_sol, y_sol = backward_euler(dydt, tspan, y0)
time_BE = time.perf_counter() - time_0
print("The time to complete the Backward-Euler solve =", time_BE)

## Time for midpoint
time_0 = time.perf_counter()
t_sol, y_sol = midpoint(dydt, tspan, y0)
time_mid = time.perf_counter() - time_0
print("The time to complete the midpoint solve =", time_mid)

## Time for solve_ivp
time_0 = time.perf_counter()
sol = scipy.integrate.solve_ivp(dydt,
                                [tspan[0], tspan[-1]],
                                [y0])
time_rk45 = time.perf_counter() - time_0
print("The time to complete the rk45 solve =", time_rk45)
```

```
The time to complete the Backward-Euler solve = 0.012664074999999997
The time to complete the midpoint solve = 0.00034115600000017565
The time to complete the rk45 solve = 0.0013615670000000008
```

We can now rank the times it takes to solve: Forward < Midpoint < RK45 < Backward. Note that this is almost exactly the opposite of the rank we had for accuracy (besides Backward Euler).

Q: Why is Backward-Euler so much slower?

A: The answer is that it's because it's implicit! It has to do the root-finding problem everytime in the for loop (which takes some amount of time). **The other methods are explicit, so all we have to do is plug in and add/subtract/multiply/divide.**

Q: Why is RK45 slower than midpoint which is slower than Forward Euler?

A: Midpoint is the same as Forward Euler, except twice. RK4 is the same as Forward Euler, except four times. Then RK45 does this variable timestep thing, so that takes even more time!

Q: Which method should we choose to solve this problem?

A: Let's start with what I would not choose. BE is slowest and has the worst error, so I definitely would not choose that. The error for RK45 is 3 orders of magnitude better than the next most accurate method, Midpoint. Since the error is so much lower for RK45, and since it only takes 1 order of magnitude more time, I would usually choose RK45 for this task. If I wanted something that prioritized speed, I would choose Midpoint because it also has low error and is very fast.

System 2

The next IVP we will solve is

$$y(t) = (5 \times 10^5) \times (-y + \sin(t)),\ 0 \leq t \leq 2\pi$$
$$y(0) = 0,$$

using 100 equally spaced points between $t = 0$ and $t = 2\pi$. We will compare what the different methods give at $t = 2\pi$ using the exact solution at $t = 2\pi$ (which I calculated at another time): $y(2\pi) = -1 \times 10^{-6}$.

In [8]:
```
# Define the ODE
dydt = lambda t, y: 5e5*(-y+np.sin(t))
# Define the IC
y0 = 0
# Define tspan, the times at which we want the solution
tspan = np.linspace(0, 2*np.pi, 100)
# Define the "exact_sol"
exact_sol = -1e-6
```

Now solve, time, and find the error for each method.

In [9]:
```python
### Solve using Forward Euler
# Start the timer
time_0 = time.perf_counter()
t, FE_ans = forward_euler(dydt, tspan, y0)
# Stop the timer and print how long it took to solve
time_FE = time.perf_counter() - time_0
print("The time to complete the Forward-Euler solve =", time_FE)
# Calculate the error, using exact_sol
FE_err = np.abs(exact_sol - FE_ans[-1])
print("Forward-Euler error = ", FE_err)
# Run this first to check work, and then use other methods

### Solve using Backward Euler
# Start the timer
time_0 = time.perf_counter()
t, BE_ans = backward_euler(dydt, tspan, y0)
# Stop the timer and print how long it took to solve
time_BE = time.perf_counter() - time_0
print("The time to complete the Backward-Euler solve =", time_BE)
# Calculate the error, using exact_sol
BE_err = np.abs(exact_sol - BE_ans[-1])
print("Backward-Euler error = ", BE_err)


### Solve using midpoint
# Start the timer
time_0 = time.perf_counter()
t, mid_ans = midpoint(dydt, tspan, y0)
# Stop the timer and print how long it took to solve
time_mid = time.perf_counter() - time_0
print("The time to complete the midpoint solve =", time_mid)
# Calculate the error, using exact_sol
mid_err = np.abs(exact_sol - mid_ans[-1])
print("Midpoint error = ", mid_err)

### Solve using scipy.integrate.solve_ivp
# Start the timer
time_0 = time.perf_counter()
sol = scipy.integrate.solve_ivp(dydt,
                            [tspan[0], tspan[-1]],
                            [y0])
# Stop the timer and print how long it took to solve
time_RK45 = time.perf_counter() - time_0
print("The time to complete the RK45 solve =", time_RK45)
# Extract the answer from solve_ivp
rk45 = sol.y[0]
# Calculate the error, using exact_sol
rk45_err = np.abs(exact_sol - rk45[-1])
print("RK45 error = ", rk45_err)
```

```
The time to complete the Forward-Euler solve = 0.0005860760000002685
Forward-Euler error =  nan
The time to complete the Backward-Euler solve = 0.010135344000000046
Backward-Euler error =  9.986573507003113e-07
The time to complete the midpoint solve = 0.0007135390000003738
Midpoint error =  nan
```

```
<ipython-input-8-e61549fefad9>:2: RuntimeWarning: overflow encountered in
double_scalars
  dydt = lambda t, y: 5e5*(-y+np.sin(t))
<ipython-input-2-9783c6424fe5>:13: RuntimeWarning: invalid value encounte
red in double_scalars
  y[k + 1] = y[k] + dt * odefun(tspan[k], y[k]) # Forward Euler step
<ipython-input-2-9783c6424fe5>:48: RuntimeWarning: invalid value encounte
red in double_scalars
  k2 = odefun(tspan[k] + dt/2, y[k] + dt/2*k1)

The time to complete the RK45 solve = 128.32015411199998
RK45 error =  9.158258369179522e-07
```

There's a few things to note here.

- First off, `scipy.integrate.solve_ivp` is still going! This may take a while, your computer may get hot. If you need to stop the calculation, go to Kernel -> Interrupt
- Note that we are getting a few warnings here. Some things aren't working as expected. Look at the error for Forward-Euler and midpoint: it says `nan`. This means "Not a Number." This means that the number got too big for python to handle. This means that the method will not work for this problem.
- Notice that **the only method that's given us an answer so far is Backward-Euler,** and the error is tiny! Even if RK45 does find the solution, we know it's going to be **a lot** slower. So I think we can be pretty confident that *Backward-Euler is the best method to use for this problem,* even if the error for RK45 is a little smaller.

The forward Euler and midpoint methods will not "blow up" if we use enough points in our time interval. We are going to try to incleane the number of points and find the magic number for them to "converge."

```
In [10]: ## First redefine tspan using 1000=1e3 equally spaced points
         tspan = np.linspace(0, 2*np.pi, 10000000)

         ### Solve using Forward Euler
         # Start the timer
         time_0 = time.perf_counter()
         t, FE_ans = forward_euler(dydt, tspan, y0)
         # Stop the timer and print how long it took to solve
         time_FE = time.perf_counter() - time_0
         print("The time to complete the Forward-Euler solve =", time_FE)
         # Calculate the error, using exact_sol
         FE_err = np.abs(exact_sol - FE_ans[-1])
         print("Forward-Euler error = ", FE_err)
         # Run this first to check work, and then use other methods


         ### Solve using midpoint
         # Start the timer
         time_0 = time.perf_counter()
         t, mid_ans = midpoint(dydt, tspan, y0)
         # Stop the timer and print how long it took to solve
         time_mid = time.perf_counter() - time_0
         print("The time to complete the midpoint solve =", time_mid)
         # Calculate the error, using exact_sol
         mid_err = np.abs(exact_sol - mid_ans[-1])
         print("Midpoint error = ", mid_err)
```

```
The time to complete the Forward-Euler solve = 23.695188158000008
Forward-Euler error =  1.000000000384528e-06
The time to complete the midpoint solve = 52.907626773000004
Midpoint error =  1.000000000369255e-06
```

The time it takes to solve increases with the number of points we have, because the for loop has to go more times. In the end, it looks like we need about 1e7 = 10 million points in order to get a solution. This obviously takes quite a while, and the error is still not smaller than the Backward-Euler method!

I think at this point it's pretty clear how we would rank which method we'd use for this problem, at least what the #1 method is.

I would definitely choose backward Euler. It is two orders of magnitude faster than RK45 and has the same accuracy as RK45.

BE was the worst choice for solving system 1 (it was the slowest and had the worst error) but it is the best choice for system 2. The reason backward Euler is good in this case is because the explicit methods needed many points (or a very small dt) in order to be *stable*. Differential equations for which is is true are called **stiff problems** or **stiff ODEs**. You generally want *implicit methods* for solving *stiff ODEs*.

Before explaining more about stability, I want to show you how to solve stiff ODEs using a built-in method. The Backward-Euler method is *implicit* which is what makes it good at solving stiff ODEs, but it's also not very accurate. There are more accurate implicit methods. We will do so using `scipy.integrate.solve_ivp(..., method='BDF')` which means "Backward Differentiation Formula." The syntax is exactly the same as we have used for `solve_ivp` before. It uses a variable-stepsize method as well.

In [11]:
```
### Solve using scipy.integrate.solve_ivp, using BDF
# Start the timer
time_0 = time.perf_counter()
sol = scipy.integrate.solve_ivp(dydt,
                                [tspan[0], tspan[-1]],
                                [y0], method='BDF')
# Stop the timer and print how long it took to solve
time_BDF = time.perf_counter() - time_0
print("The time to complete the RK45 solve =", time_BDF)
# Extract the answer from solve_ivp
BDF = sol.y[0]
# Calculate the error, using exact_sol
BDF_err = np.abs(exact_sol - BDF[-1])
print("BDF error = ", BDF_err)
```

```
The time to complete the RK45 solve = 0.03224144600000045
BDF error =  1.0001580814357555e-06
```

Notice that this is extremely fast and has good error! It's actually still worse than Backward Euler in this case, but it could be made more accurate using `rtol`.

So how do we explain what we are seeing? In order to explain it we need to discuss *stability.*

Stability

The definition and study of *stability* is a complicated one. Unfortunately, this means that I am also using somewhat non-standard definitions of words like "stable" and "unstable". If you are interested in the standard definitions, a good place to start is by looking up "A-stability".

We are not going to go into the details. Very roughly, we will say that a differential equation solver is *unstable* if the numerical solution *blows up* or goess off to infinity as $t \to \infty$. If this does not happen, we call the ODE solver *stable*.

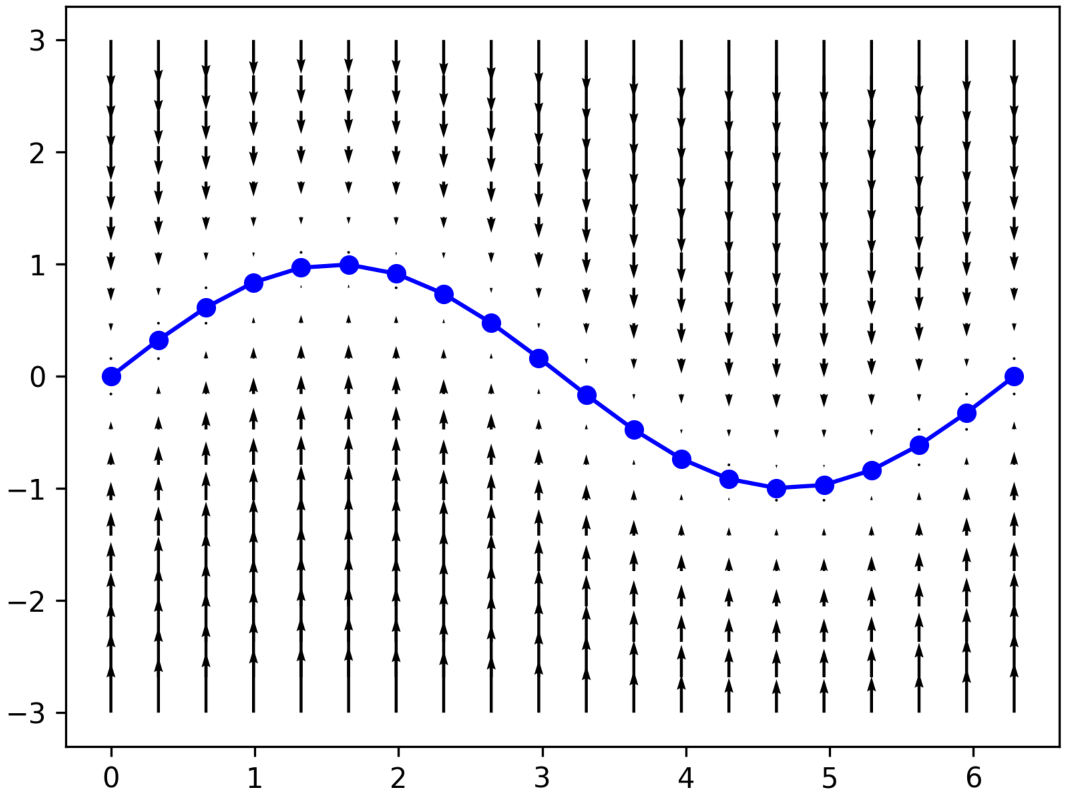
Let's think about the IVP we were trying to solve above,

$$y(t) = (5 \times 10^5) \times (-y + \sin(t)),\ 0 \leq t \leq 2\pi$$
$$y(0) = 0.$$

We are going to plot the vector field for this differential equation, using `quiver`.

```
In [12]: # Setup figure
         ig, ax = plt.subplots()
         # Create tspan and yspan for the vector field plot. Use 20 equally spaced po
         # tspan should go from 0 to 2*pi, yspan should go from -3 to 3
         span = np.linspace(0, 2*np.pi, 20)
         span = np.linspace(-3, 3, 20)
         # Create the meshgrid
         , Y = np.meshgrid(tspan, yspan)
         # Plot using quiver
         x.quiver(T, Y, np.ones(T.shape), 5e5*(-Y + np.sin(T)))
```

Figure 1



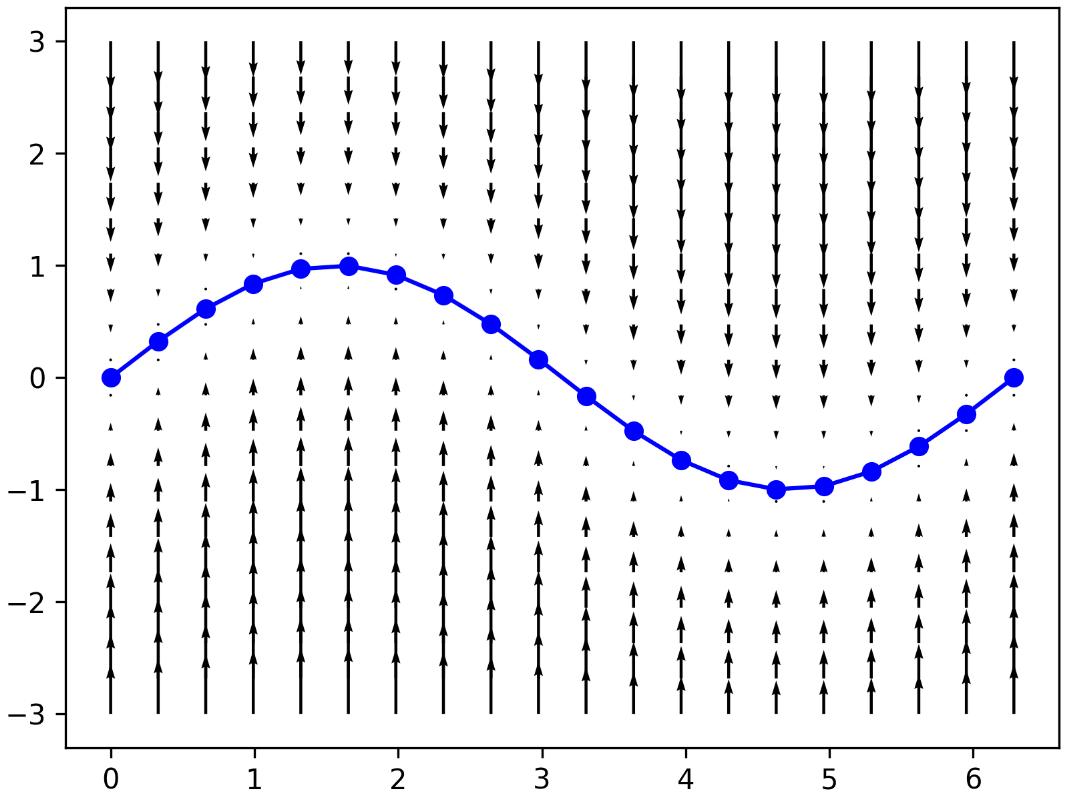
```
Out[12]: <matplotlib.quiver.Quiver at 0x7fc406aa7a30>
```

First let's see what happens when we solve this with the Backward-Euler method because we know it works well.

In [13]:
```
# Solve with Backward Euler
t, BE_ans = backward_euler(dydt, tspan, y0)
# Then plot the solution in blue. Dots with lines in between
ax.plot(t, BE_ans, 'bo-')
# Show the figure
fig
```

```
/Users/jeremyupsal/opt/anaconda3/lib/python3.8/site-packages/scipy/optimi
ze/minpack.py:175: RuntimeWarning: The iteration is not making good progr
ess, as measured by the
  improvement from the last ten iterations.
  warnings.warn(msg, RuntimeWarning)
```

Figure 1



This should look strange to you. All of the slopes on the slope field look vertical. But, the solution is not vertical. It is a lot like a sine wave! But, we know that the solution always follows the slope field... so what's up? To see what's going on better, let's change the differential equation a little bit. Let's look at

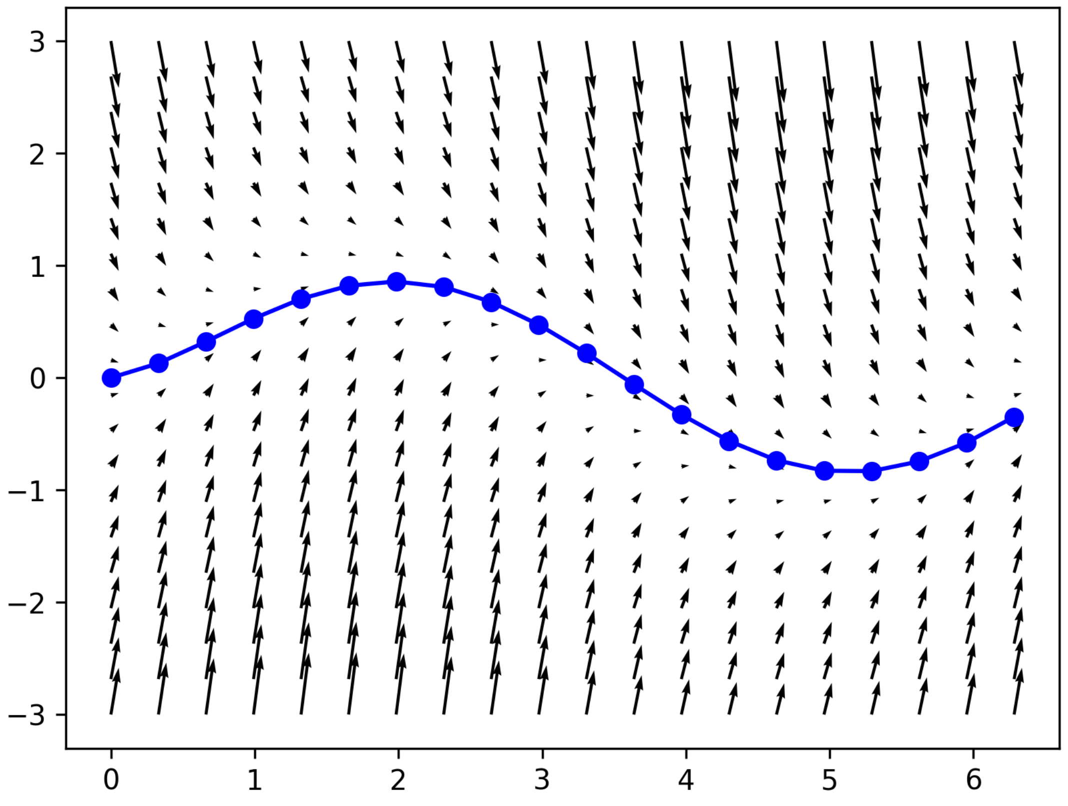
$$y' = 2(-y + \sin(t)),\ \ y(0) = 0.$$

We have changed the large constant out front. Doing so changes the figure.

In [14]:
```
# Setup new figure
fig2, ax2 = plt.subplots()
# Create tspan and yspan for the vector field plot. Use 20 equally spaced p
# tspan should go from 0 to 2*pi, yspan should go from -3 to 3
tspan = np.linspace(0, 2*np.pi, 20)
yspan = np.linspace(-3, 3, 20)
# Create the meshgrid
T, Y = np.meshgrid(tspan, yspan)
# Define the new ODE, dydt
dydt = lambda t, y: 2*(-y+np.sin(t))
# Plot using quiver
ax2.quiver(T, Y, np.ones(T.shape), dydt(T, Y))

# Calculate the solution using Backward Euler
t, BE_ans = backward_euler(dydt, tspan, y0)
# Then plot the solution in blue. Dots with lines in between
ax2.plot(t, BE_ans, 'bo-')
```

Figure 2



Out[14]:
```
[<matplotlib.lines.Line2D at 0x7fc406afcca0>]
```

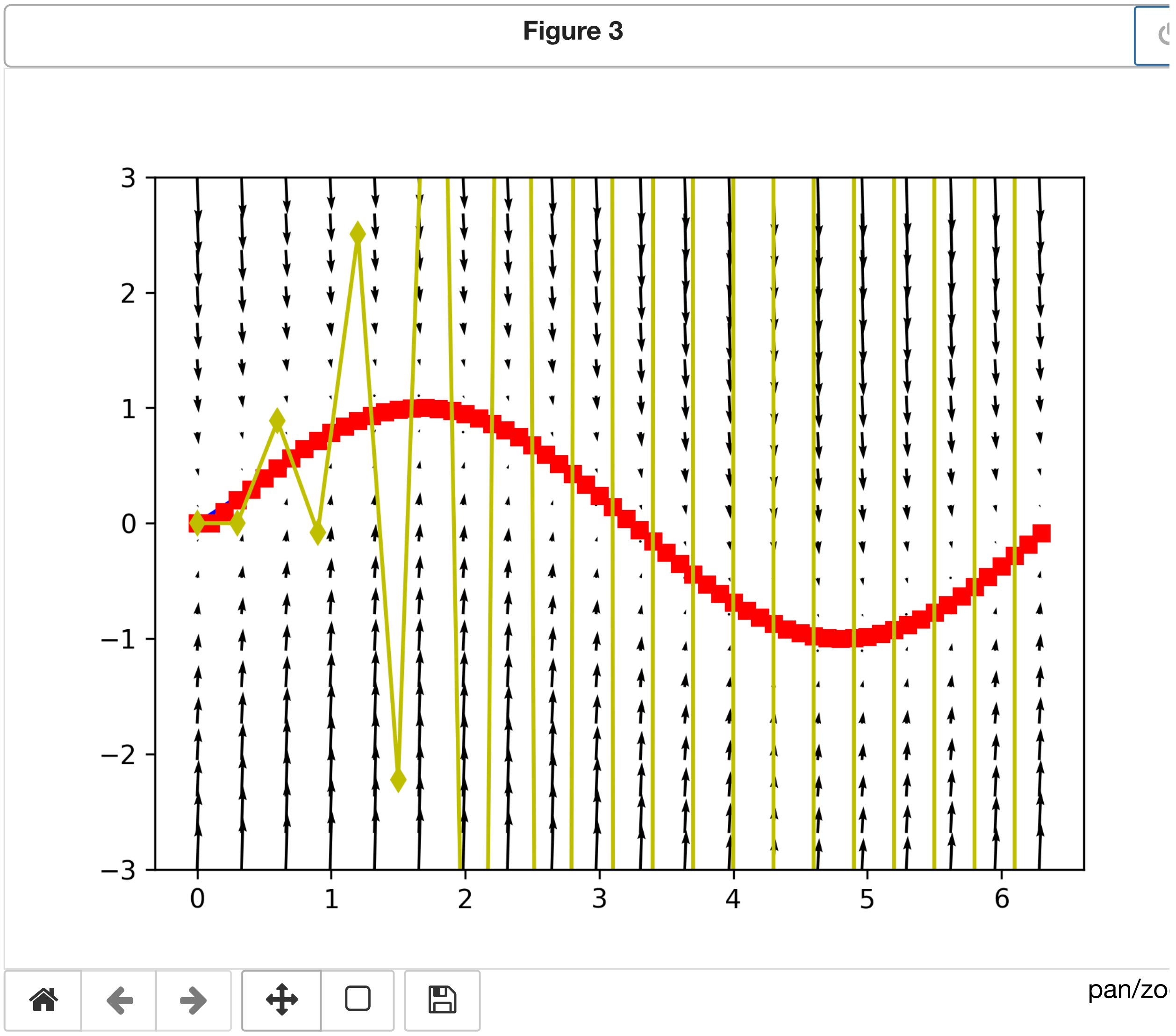
Notice that far away from the exact solution, the arrows of the slope field are closer to vertical but they curve in as we get close to the solution curve. If we increase the constant to 10, i.e.

$$y'(t) = 10(-y + \sin(t)),$$

we get the following figure.

```
In [15]: # Do all of the same steps as above, except with the constant = 10
         # Setup new figure
         fig3, ax3 = plt.subplots()
         # Create tspan and yspan for the vector field plot. Use 20 equally spaced po
         # tspan should go from 0 to 2*pi, yspan should go from -3 to 3
         tspan = np.linspace(0, 2*np.pi, 20)
         yspan = np.linspace(-3, 3, 20)
         # Create the meshgrid
         T, Y = np.meshgrid(tspan, yspan)
         # Define the new ODE, dydt
         dydt = lambda t, y: 10*(-y+np.sin(t))
         # Plot using quiver
         ax3.quiver(T, Y, np.ones(T.shape), dydt(T, Y))

         # Calculate the solution using Backward Euler
         t, BE_ans = backward_euler(dydt, tspan, y0)
         # Then plot the solution in blue.
         ax3.plot(t, BE_ans, 'b-')
```



```
Out[15]: [<matplotlib.lines.Line2D at 0x7fc406b44c10>]
```

Away from the solution, the slopes on the slope field are even more close to being vertical. The region around the exact solution where the arrows curve in toward the direction of the solution is thinner. So there is a region (near the solution) where the slope field changes very rapidly. **This is what makes a problem stiff.** If you go back to the case with the constant 5e5, the region around the exact solution where the slopes curve in is so small that we can't even see it. That's what makes this such a stiff problem.

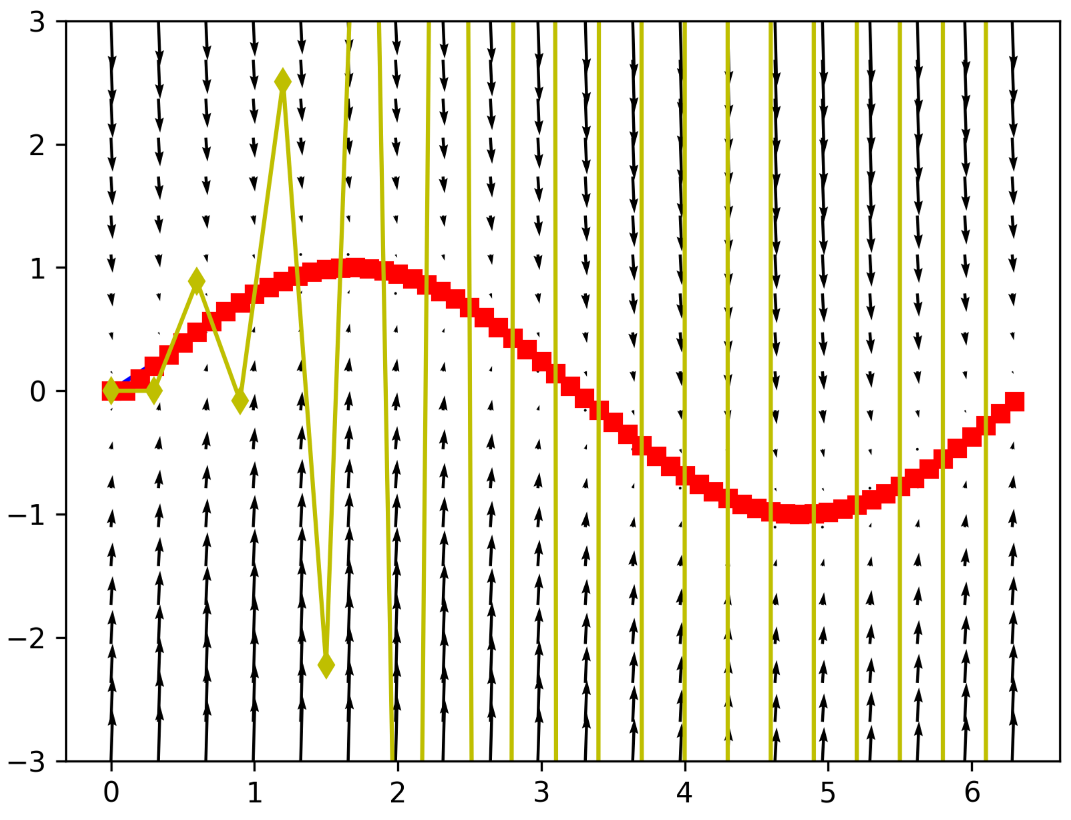
Let's continue to work with this ODE, $y'(t) = 10(-y + \sin(t))$, to see what happens with Forward Euler. We saw that for Δt small enough the solution converged (we got some small error). So let's choose a small Δt and see what happens with the solution.

We will choose $\Delta t = 0.1$ and solve below.

In [16]:
```
# Set dt
dt = 0.1
# Create the new tspan, from 0 to 2 pi with this dt
tspan = np.arange(0, 2*np.pi+dt, dt)
# Solve with Forward Euler
t, FE_ans = forward_euler(dydt, tspan, y0)
# Plot the solution using red squares
ax3.plot(t, FE_ans, 'rs-')

# Show the figure
fig3
```

Figure 3



We can see that the Forward-Euler solution matches up. What if we choose $\Delta t = 0.3$?

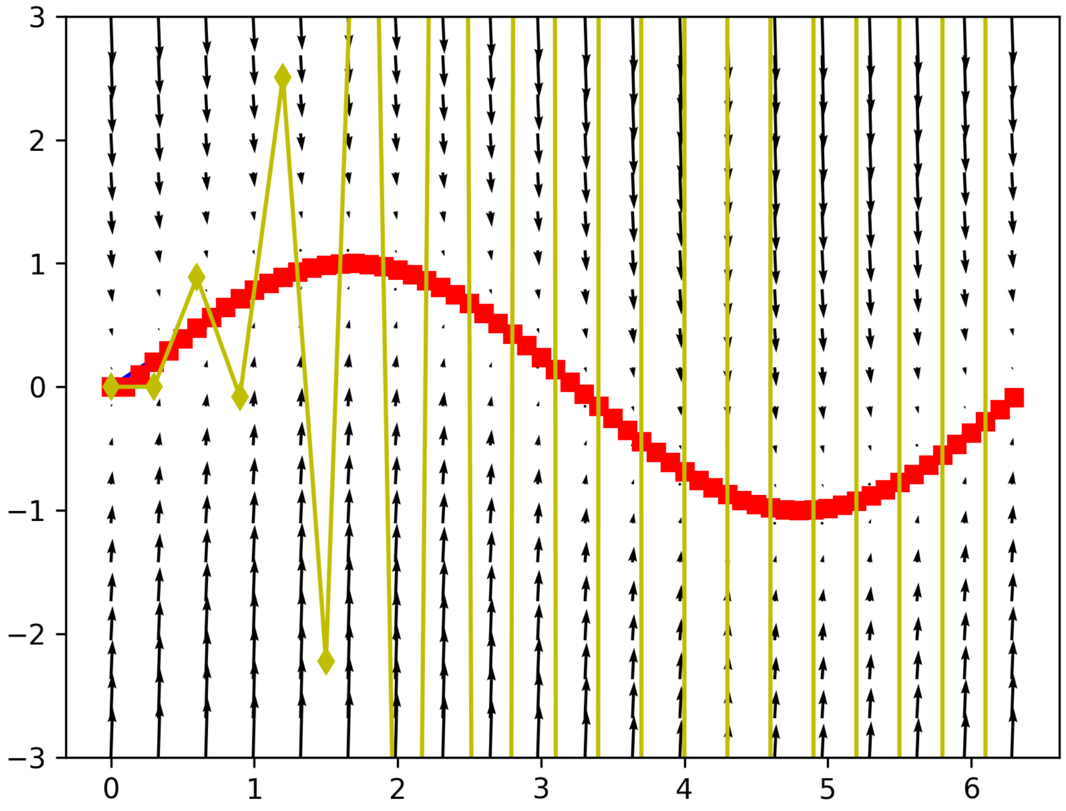
Let's solve with $\Delta t = 0.3$ and plot with yellow diamonds.

In [17]:
```python
# Set dt
dt = 0.3
# Create the new tspan, from 0 to 2 pi with this dt
tspan = np.arange(0, 2*np.pi+dt, dt)
# Solve with Forward Euler
t, FE_ans = forward_euler(dydt, tspan, y0)
# Plot the solution using yellow diamonds
ax3.plot(t, FE_ans, 'yd-')

ax3.set_ylim([-3, 3]) # We are going to fix the ylimit.

# Show the figure
fig3
```

Figure 3



pan/zo

We can see that the solution is blowing up! There's some error in the first step, then some more error, then more and more etc. **This is numerical instability.**

Forward Euler becomes unstable because there is some error after the first step. Because you are not right on the curve after the first step, the slope field is relatively steep, causing an overshoot. Then the successive overshoots take you to regions where the slope field is even steeper. It is easy to see why this would be an even bigger problem using the constant 5e5 because almost all of the slopes on the slope field are nearly vertical. Even a small error off of the exact solution takes you to a region with a very steep slope in the opposite direction.

Backward Euler avoids this problem because it traces backward along the slope field. It can never end up in a region with nearly vertical slopes because if you trace any of them backward, you don't end up near the exact solution. But *every explicit method will have this problem.*

When using an explicit method, **very small step sizes are needed to solve stiff problems.** The method *will* work with a very small step size, but sometimes it can become prohibitively small, meaning the method will just take too long to converge. That is why we saw `solve_ivp` taking so long to solve; because `solve_ivp` will decrease the step size automatically to make the error small, it has to choose a very very small Δt, meaning that it is taking *more steps in the for loop*, which takes more time!

So, how do you identify stiff problems? When would you know which method to use?

- Usually you would start with using `solve_ivp` because it's highly accurate. If it's taking a long time, your ODE is probably stiff. You may want to use `method = BDF`.
- You can also recognize stiff ODEs from the vector field itself. **If there is a small region where the vector field changes rapidly, then the problem is likely stiff.** (That's what we see here, the vector field goes from pointing straight up to horizontal in a narrow region).

Finally, just to reiterate:

- When you have a stiff problem, **use an implicit method.**
- Otherwise, use an **explicit method** because they are usually faster!

Material below is optional and we will not cover it in class.

Stability

Remember that stability has to do with how our solution behaves if we make the final time T very large. This turns out to be a very complicated concept, and it often depends on the details of our differential equation, so we will only look at one of the simplest versions of stability. In particular, we will only talk about stability in terms of a very simple family of differential equations:

$\dot{x} = \lambda x$ and $x(0) = x_0$,

where λ is a constant. (It turns out that when we try to solve systems of equations we will need to worry about complex values of λ, but for the moment we will just pretend that λ is real.) This is called a *test problem*.

Note: I am trying to avoid more technical definitions of stability and some of the more complicated concepts from differential equations. Unfortunately, this means that I am also using somewhat non-standard definitions of words like "stable" and "unstable". If you are interested in the standard definitions, a good place to start is by looking up "A-stability".

We already saw in the previous lecture that the true solution to the test problem is

$$x(t) = x_0 e^{\lambda t}.$$

We want to know what happens to this solution (and to our approximations, but let's start with the true solution) after a very long time. In this case, there are only two possibilities. If $\lambda > 0$, then x goes off to $\pm\infty$ as time goes on. We will call the true solution "unstable" in this case. If $\lambda < 0$, then x goes to zero as time goes on. We will call the true solution "stable" in this case. (Technically, there is a third possibility. If $\lambda = 0$, then $x(t)$ stays constant forever. However, this is a pretty degenerate case, so we won't worry about it here.)

Ideally, we want our numerical methods to capture this stable/unstable behavior. That is, if we solve this initial value problem using something like the forward or backward Euler method, we want our approximation to go to infinity when $\lambda > 0$ and to go to zero when $\lambda < 0$. Unfortunately, it turns out that this is not actually possible. We will always have some tradeoff where our approximation goes to infinity even though the true solution does not, or vice versa.

Stability of Forward Euler

Let's look at what happens when we apply the forward Euler method to the test problem. If we already know all of the x values up to x_k, then we can find the next x value using the equation

$$x_{k+1} = x_k + \Delta t f(t_k, x_k).$$

In this case, $f(t, x) = \lambda x$, so we have

$$x_{k+1} = x_k + \Delta t \lambda x_k = (1 + \Delta t \lambda) x_k.$$

If we use this formula with $k = 0$, then we find

$$x_1 = (1 + \Delta t \lambda) x_0.$$

Likewise, if we use $k = 1$ then we find

$$x_2 = (1 + \Delta t \lambda) x_1 = (1 + \Delta t \lambda)^2 x_0.$$

It is easy to check that if we repeat this process k times we will get the general formula

$$x_k = (1 + \Delta t \lambda)^k x_0.$$

From this equation we can see that our approximations x_k go to $\pm\infty$ if $|1 + \Delta t \lambda| > 1$. If this is the case then we say that forward Euler is "unstable". Likewise, if $|1 + \Delta t \lambda| < 1$ then our approximations x_k go to zero and we say that forward Euler is "stable".

It is very important to notice that this is not the same as the stability of the true solution. The true solution is stable whenever λ is negative, but it is easy to come up with combinations of Δt and λ where λ is negative, but forward Euler is unstable. For instance, if $\Delta t = 1$ and $\lambda = -10$, then $|1 + \Delta t \lambda| = 9 > 1$, so forward Euler is unstable even though the true solution is stable. However, it is easy to check that if λ is positive then $|1 + \Delta t \lambda| > 1$, so if the true solution is unstable then so is the forward Euler approximation.

We therefore know that the forward Euler solution is unstable whenever the true solution is unstable, but sometimes forward Euler is not stable even though the true solution is stable. In particular, if $-2 < \Delta t \lambda < 0$, then the forward Euler approximation will be stable, but if $\Delta t \lambda < -2$

then the forward Euler approximation will be unstable (even though the true solution is actually stable).

Notice that, for any fixed value of λ, if we choose Δt small enough then the stability of our approximation will always match the stability of the true solution, but if λ is negative then we might need a very small Δt to make sure that forward Euler is stable.

Stability of Backward Euler

Similarly, we can look at what happens when we apply the backward Euler method to the test problem. If we already know all of the x values up to x_k, then we can find the next x value using the equation

$$x_{k+1} = x_k + \Delta t f(t_{k+1}, x_{k+1}).$$

In this case, $f(t, x) = \lambda x$, so we have

$$x_{k+1} = x_k + \Delta t \lambda x_{k+1}.$$

This is an implicit equation, but it is very easy to solve for x_{k+1}. We get

$$x_{k+1} = \frac{1}{1-\Delta t\lambda} x_k.$$

If we use this formula with $k = 0$, we find that

$$x_1 = \frac{1}{1-\Delta t\lambda} x_0.$$

Likewise, if we use $k = 1$ then we find

$$x_2 = \frac{1}{1-\Delta t\lambda} x_1 = \left(\frac{1}{1-\Delta t\lambda}\right)^2 x_0.$$

It is easy to check that if we repeat this process k times we will get the general formula

$$x_k = \left(\frac{1}{1-\Delta t\lambda}\right)^k x_0.$$

From this equation we can see that our approximations x_k go to $\pm\infty$ if $|1/(1 - \Delta t\lambda)| > 1$. If this is the case then we say that backward Euler is "unstable". Likewise, if $|1/(1 - \Delta t\lambda)| < 1$ then our approximations x_k go to zero and we say that backward Euler is "stable".

Just like with forward Euler, it is very important to notice that this is not the same as the stability of the true solution (or as the rule for forward Euler). In particular, if λ is positive then the true solution is always unstable, but it is easy to come up with combinations of Δt and λ where λ is positive but backward Euler is stable. For example, if $\Delta t = 1$ and $\lambda = 10$, then $|1/(1 - \Delta t\lambda)| = 1/9 < 1$, so backward Euler is stable even though the true solution is unstable.

We therefore know that the backward Euler solution is stable whenever the true solution is stable, but sometimes backward Euler is still stable even though the true solution is unstable. In particular, the backward Euler approximation is only unstable when $0 < \Delta t\lambda < 2$. If $\Delta t\lambda > 2$ then backward Euler will be stable (even though the true solution is actually unstable).

For any fixed value of λ, if we choose Δt small enough then the stability of our approximation will always match the stability of the true solution, but if λ is positive then we might need a very small Δt to make sure that the behavior of backward Euler matches that of the true solution.

Overview

We only analyzed the stability of a very limited set of differential equations (the test problems) and we only looked at two methods (forward and backward Euler), but it turns out that this analysis applies to a wide variety of problems and methods. More complicated differential equations don't usually just go to zero as time goes on, but we are still interested in correctly capturing whatever long term behavior they have. We will say that the solution to a differential equation is "stable" if it does *not* go to infinity as time goes on. (This is not a very good definition, but we would have to spend several classes on differential equations theory in order to make a substantially better one.) It turns out that explicit approximation methods are prone to going to infinity even when the true solution is stable, while implicit methods are good at capturing stable behavior. This means that explicit methods often need a fairly small time step Δt in order to correctly capture long-term stable behavior. Implicit methods, on the other hand, can correctly capture long-term stable behavior even with a fairly large time step.

In real world applications, solutions rarely go to infinity. For example, if we are modeling the population of a species, there are physical limits (like space or resource requirements) that keep this population from becoming infinitely large. Because of this, we typically expect the true solution of our initial value problems to be stable. This means that implicit methods like backward Euler can usually correctly capture long-term behavior with a larger time step than explicit methods like forward Euler. We therefore say that implicit methods like backward Euler have "better stability properties" or are "more stable" than explicit methods.