3.3

Consider the following ODE:

$$a^2\ddot{y}(t) + y(t) = Ku(t) \tag{28}$$

with the following conditions:

$$u(t) = 2, \qquad a = 2, \qquad K = 1, \qquad y(0) = 0, \qquad \dot{y}(0) = 1 \tag{29}$$

Thus:

$$4\ddot{y}(t) + y(t) = 2 \tag{30}$$

3.3.A

To solve the ODE analytically, we first solve the Homogeneous equation:

$$4\ddot{y} + y = 0 \tag{31}$$

$$\ddot{y} = -\frac{y}{4} \tag{32}$$

Representing in the form:

$$\dot{y} = Ay \tag{33}$$

We get:

$$\begin{bmatrix} \dot{y} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{1}{4} & 0 \end{bmatrix} \begin{bmatrix} y \\ \dot{y} \end{bmatrix} \tag{34}$$

The eigenvalues of A are the following complex conjugate roots:

$$\lambda_1 = \frac{i}{2}, \qquad \lambda_2 = -\frac{i}{2} \tag{35}$$

And so the general solution to our ODE is:

$$y = \alpha_1 cos\left(\frac{t}{2}\right) + \alpha_2 sin\left(\frac{t}{2}\right) \tag{36}$$

Using the method of undetermined coefficients and our initial conditions, we have:

$$y = C \tag{37}$$

$$4\ddot{y} + y = 2 \tag{38}$$

$$4\ddot{y} + C = 2 \tag{39}$$

$$C = 2 \tag{40}$$

Plugging in our coefficient C and substituting our initial conditions (29) into (36), we get the following particular solution:

$$y = -2cos\left(\frac{t}{2}\right) + 2sin\left(\frac{t}{2}\right) + 2 \tag{41}$$

Plotting y as a function of t, we get:

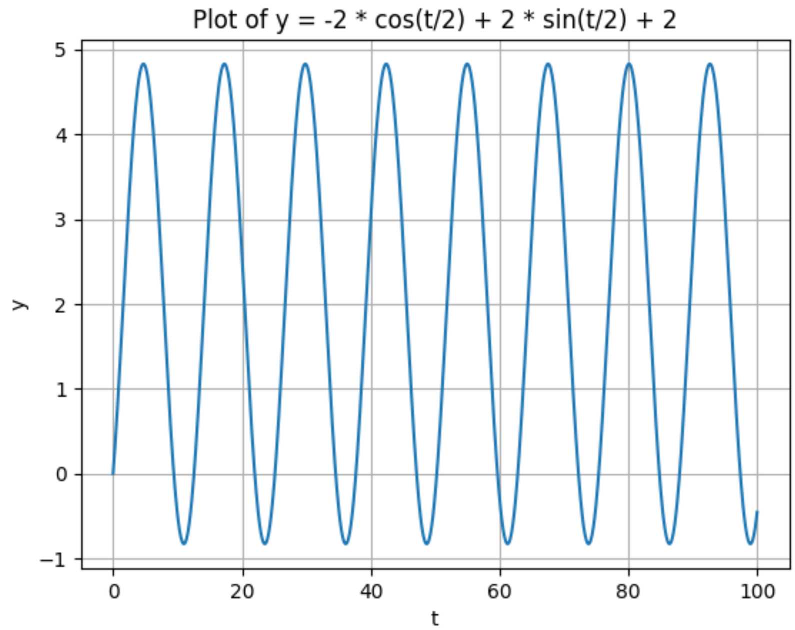

Figure 1:

3.3.B

We rearrange our ODE (28) into the following form:

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{42}$$

$$y(t) = Cx(t) + Du(t) \tag{43}$$

Where:

$$x(t) = \begin{bmatrix} y \\ \dot{y} \end{bmatrix}, \qquad u(t) = u, \qquad y(t) = y \tag{44}$$

And find A, B, C, D by analytic inspection:

$$\begin{bmatrix} \dot{y} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{1}{a^2} & 0 \end{bmatrix} \begin{bmatrix} y \\ \dot{y} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{K}{a^2} \end{bmatrix} u \tag{45}$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} y \\ \dot{y} \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} u \tag{46}$$

Thus,

$$A = \begin{bmatrix} 0 & 1 \\ -\frac{1}{a^2} & 0 \end{bmatrix}, \qquad B = \begin{bmatrix} 0 \\ \frac{K}{a^2} \end{bmatrix}, \qquad C = \begin{bmatrix} 1 & 0 \end{bmatrix}, \qquad D = \begin{bmatrix} 0 \end{bmatrix} \tag{47}$$

3.3.C

We would like to numerically approximate the ODE in (28) with the following numerical ODE solvers:

```
    # ODE Solver Function Definitions

def ode1_vectorized(f, y0, t0, t_end, dt):
  t = np.arange(t0, t_end, dt)
  y = np.zeros((len(t), len(y0)))
  y[0] = y0
  for i in range(1, len(t)):
    y [ i ] = y [ i - 1] + f ( t [ i - 1] , y [ i - 1]) * dt
  return t, y

def ode2_vectorized(f, y0, t0, t_end, dt):
  t = np.arange(t0, t_end, dt)
  y = np.zeros((len(t), len(y0)))
  y[0] = y0
  for i in range(1, len(t)):
    k1 = f ( t [ i - 1] , y [ i - 1])
    k2 = f ( t [ i - 1] + dt , y [ i - 1] + k1 * dt )
    y [ i ] = y [ i - 1] + ( k1 + k2 ) * dt / 2
  return t, y

def ode4_vectorized(f, y0, t0, t_end, dt):
  t = np.arange(t0, t_end, dt)
  y = np.zeros((len(t), len(y0)))
  y[0] = y0
  for i in range(1, len(t)):
    k1 = f ( t [ i - 1] , y [ i - 1])
    k2 = f ( t [ i - 1] + dt / 2 , y [ i - 1] + k1 * dt / 2)
    k3 = f ( t [ i - 1] + dt / 2 , y [ i - 1] + k2 * dt / 2)
    k4 = f ( t [ i - 1] + dt , y [ i - 1] + k3 * dt )
    y [ i ] = y [ i - 1] + ( k1 + 2 * k2 + 2 * k3 + k4 ) * dt / 6
  return t, y
```

We begin by representing our ODE as follows:

```
    # 4y" + y = 2
def f(t, y):
  y1, y2 = y
  dy1dt = y2
  dy2dt = (2 - y1) / 4
  return np.array([dy1dt, dy2dt])

# Initial conditions
y0 = [0, 1]
t0 = 0
t_end = 100
```

The following is a plot of the error generated by `ODE1()` plotted against time for different step sizes T. Included, also, is the code used to generate the approximate solution vectors, error vectors, and plots:

```
    # ODE 1:
t_1, y_1 = ode1_vectorized(f, y0, t0, t_end, 0.05)
t_2, y_2 = ode1_vectorized(f, y0, t0, t_end, 0.1)
t_3, y_3 = ode1_vectorized(f, y0, t0, t_end, 0.15)

y_e_1 = -2 * np.cos(t_1/2) + 2 * np.sin(t_1/2) + 2
y_e_2 = -2 * np.cos(t_2/2) + 2 * np.sin(t_2/2) + 2
y_e_3 = -2 * np.cos(t_3/2) + 2 * np.sin(t_3/2) + 2

err1 = np.array(y_e_1 - y_1[:, 0])
err2 = np.array(y_e_2 - y_2[:, 0])
err3 = np.array(y_e_3 - y_3[:, 0])

plt.plot(t_1, err1, t_2, err2, t_3, err3)
plt.legend(['T = 0.05', 'T = 0.1', 'T = 0.15'])
plt.grid(True)
plt.xlabel('t')
plt.ylabel('y')
plt.title('Solution of 4y" + y = 2 with ODE1')
plt.show()
```

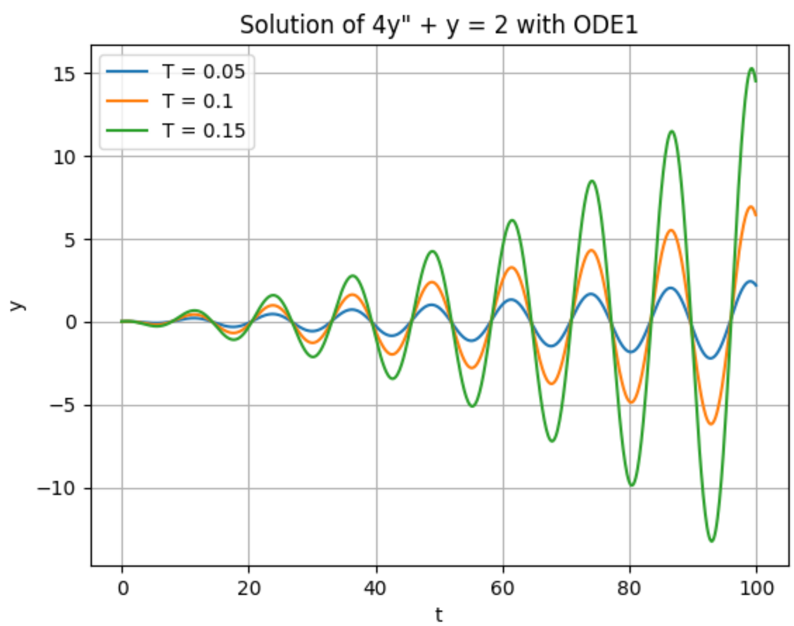

Figure 2:

We can see how the magnitude of error generated by the `ODE1()` approximation decreases as the step size T is decreased. We can also see how as time goes on and more approximation steps are taken, the error increases as the error at each step propagates into the next.

The following is a plot of the error generated by `ODE2()` plotted against time for different step sizes T. Included, also, is the code used to generate the approximate solution vectors, error vectors, and plots:

```
    # ODE 2:
t_1, y_1 = ode2_vectorized(f, y0, t0, t_end, 0.3)
t_2, y_2 = ode2_vectorized(f, y0, t0, t_end, 0.4)
t_3, y_3 = ode2_vectorized(f, y0, t0, t_end, 0.5)

y_e_1 = -2 * np.cos(t_1/2) + 2 * np.sin(t_1/2) + 2
y_e_2 = -2 * np.cos(t_2/2) + 2 * np.sin(t_2/2) + 2
y_e_3 = -2 * np.cos(t_3/2) + 2 * np.sin(t_3/2) + 2

err1 = np.array(y_e_1 - y_1[:, 0])
err2 = np.array(y_e_2 - y_2[:, 0])
err3 = np.array(y_e_3 - y_3[:, 0])

plt.plot(t_1, err1, t_2, err2, t_3, err3)
plt.legend(['T = 0.3', 'T = 0.4', 'T = 0.5'])
plt.grid(True)
plt.xlabel('t')
plt.ylabel('y')
plt.title('Solution of 4y" + y = 2 with ODE2')
plt.show()
```

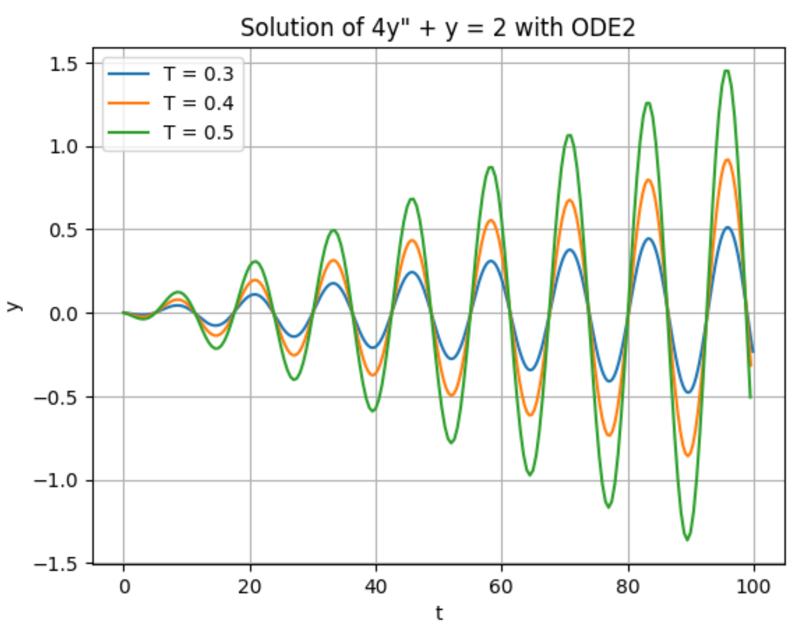

Figure 3:

We can see how the magnitude of error generated by the `ODE2()` approximation decreases as the step size T is decreased. We can also see how as time goes on and more approximation steps are taken, the error increases as the error at each step propagates into the next. Additionally, we notice that the max error of the `ODE2()` method is nearly one tenth of the max error of the `ODE1()` method.

The following is a plot of the error generated by `ODE4()` plotted against time for different step sizes T. Included, also, is the code used to generate the approximate solution vectors, error vectors, and plots:

```
    # ODE 4:
t_1, y_1 = ode4_vectorized(f, y0, t0, t_end, 0.1)
t_2, y_2 = ode4_vectorized(f, y0, t0, t_end, 0.2)
t_3, y_3 = ode4_vectorized(f, y0, t0, t_end, 0.3)
t_4, y_4 = ode4_vectorized(f, y0, t0, t_end, 0.4)

y_e_1 = -2 * np.cos(t_1/2) + 2 * np.sin(t_1/2) + 2
y_e_2 = -2 * np.cos(t_2/2) + 2 * np.sin(t_2/2) + 2
y_e_3 = -2 * np.cos(t_3/2) + 2 * np.sin(t_3/2) + 2
y_e_4 = -2 * np.cos(t_4/2) + 2 * np.sin(t_4/2) + 2

err1 = np.array(y_e_1 - y_1[:, 0])
err2 = np.array(y_e_2 - y_2[:, 0])
err3 = np.array(y_e_3 - y_3[:, 0])
err4 = np.array(y_e_4 - y_4[:, 0])

plt.plot(t_1, err1, t_2, err2, t_3, err3, t_4, err4)
plt.legend(['T = 0.1', 'T = 0.4', 'T = 0.3', 'T = 0.4'])
plt.grid(True)
plt.xlabel('t')
plt.ylabel('y')
plt.title('Solution of 4y" + y = 2 with ODE4')
plt.show()
```

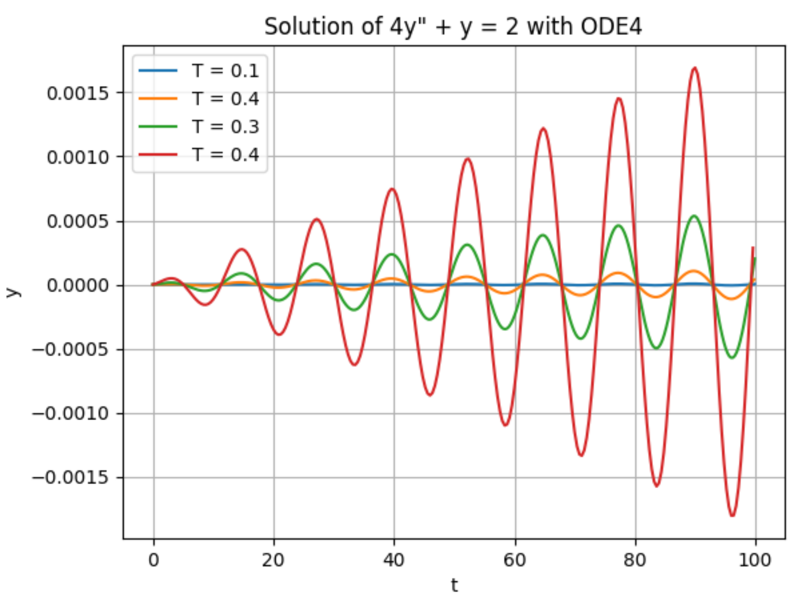

Figure 4:

We can see how the magnitude of error generated by the `ODE4()` approximation decreases as the step size T is decreased. We can also see how as time goes on and more approximation steps are taken, the error increases as the error at each step propagates into the next. Additionally, one observation we can make is that, at step size $T = 0.4$, the max error of the `ODE2()` method is nearly 3 orders of magnitude less than the max error of the `ODE1()` method, a testament to the precision of the `ODE4()` method. When $T = 0.1$, we see the error is a flat line at zero at the scale of this plot, which means the error is decreasing

much faster with respect to step size T compared to any of the previous methods.
Something we can also observe from the above plots is that the higher order approximation methods can tolerate much higher step sizes. In the `ODE1()` method, even with a step size of $T = 0.05$, the magnitude of error was reaching ± 15, while a step size of $T = 0.4$ was producing a magnitude of error near ± 0.0015 with the `ODE4()` method.