# 3 Assignment 3

## 3.1

Consider the following ODE:

$$\dot{x} = -x \tag{1}$$

The general form for the Backward Euler Approximation Method is:

$$\hat{x}_{k+1} = \hat{x}_k + Tf(\hat{x}_{k+1}) \tag{2}$$

Plugging in $f(x) = -x$ and substituting $x_k$ for $\hat{x}_k$ (assuming we are given an exact $x_k$), we get:

$$\hat{x}_{k+1} = x_k - T\hat{x}_{k+1} \tag{3}$$

as our implicit update function for finding $\hat{x}_{k+1}$ given $x_k$.

## 3.2

Consider an ODE defined by the following equation:

$$\dot{x}(t) = f(x(t)) \tag{4}$$

Consider, now, numerically approximating the solution to (4) with the following algorithm:

$$\hat{x}_{k+1} = \hat{x}_k + T(\theta f(\hat{x}_k) + (1+\theta) f(\hat{x}_{k+1})) \tag{5}$$

### 3.2.A Order of Local Error at $\theta = 0, 0.5, 1$

When $\theta = 0$, we have the Backwards Euler Method:

$$\hat{x}_{k+1} = \hat{x}_k + Tf(\hat{x}_{k+1}) \tag{6}$$

To determine the Order of Local Error of the Backwards Euler Method, we will begin by representing the exact update algorithm with a Taylor Series Approximation:

$$x_{k+1} = x_k + Tf(x_k) + \frac{T^2}{2} f'(x_k) + O(T^3) \tag{7}$$

Now we will represent our Backwards Euler update function in the same form:

$$\hat{x}_{k+1} = \hat{x}_k + Tf(\hat{x}_{k+1}) \tag{8}$$

$$\hat{x}_{k+1} = \hat{x}_k + T(f(\hat{x}_k) + Tf'(\hat{x}_k) + \frac{T^2}{2} f''(\hat{x}_k) + O(T^3)) \tag{9}$$

$$\hat{x}_{k+1} = \hat{x}_k + Tf(\hat{x}_k) + T^2 f'(\hat{x}_k) + \frac{T^3}{2} f''(\hat{x}_k) + TO(T^3)) \tag{10}$$

Now, we subtract our exact $x_{k+1}$ from our approximated $\hat{x}_{k+1}$ to get:

$$\hat{x}_{k+1} - x_{k+1} = T^2 f'(\hat{x}_k) - \frac{T^2}{2} f'(x_k) \tag{11}$$

Thus, we can see that our error occurs in the term of order 2, and so the order of local error in the Backwards Euler Method is 2.

When $\theta = 0.5$, we have the Trapezoidal Method:

$$\hat{x}_{k+1} = \hat{x}_k + \frac{T}{2}(f(\hat{x}_k) + f(\hat{x}_{k+1})) \tag{12}$$

To determine the Order of Local Error of the the Trapezoidal Method, we will begin once again by representing our exact update function in the form of a Taylor Series:

$$x_{k+1} = x_k + Tf(x_k) + \frac{T^2}{2} f'(x_k) + \frac{T^3}{3} f''(x_k) + O(T^4) \tag{13}$$

Next, we will manipulate our Trapezoidal Approximation update function into a similar form:

$$\hat{x}_{k+1} = \hat{x}_k + \frac{T}{2}(f(\hat{x}_k) + f(\hat{x}_{k+1})) \tag{14}$$

$$\hat{x}_{k+1} = \hat{x}_k + \frac{T}{2}(f(\hat{x}_k) + f(\hat{x}_k) + Tf'(\hat{x}_k) + \frac{T^2}{2}f''(\hat{x}_k) + O(T^3)) \tag{15}$$

Simplifying gives us:

$$\hat{x}_{k+1} = \hat{x}_k + Tf(\hat{x}_k) + \frac{T^2}{2}f'(\hat{x}_k) + \frac{T^3}{4}f''(\hat{x}_k) + ... \tag{16}$$

Comparing with our exact representation (13), we can see the first discrepancy occurs in the 3rd order term, thus the order of local error of the Trapezoidal Method is 3.

When $\theta = 1$, we have the Forwards Euler Method:

$$\hat{x}_{k+1} = \hat{x}_k + Tf(\hat{x}_k) \tag{17}$$

If we recall the Taylor approximation of the exact equation (equation (7)) and subtract our Forwards Euler update equation from it, we will be left with a 2nd order term:

$$(7) - (17) = \frac{T^2}{2}f'(x_k) \tag{18}$$

Thus, the order of error of the Forwards Euler Method is 2.

### 3.2.B

Consider the test equation:

$$\dot{x}(t) = \lambda x(t) \tag{19}$$

where:

$$\lambda = \sigma + \omega j \tag{20}$$

We will determine the stability of (5) with the above test equation (19). Using this specific test equation will allow us to generalize the stability of (5) for real and complex ODEs. Our goal is to model how any error in $\hat{x}_k$ will propagate to $\hat{x}_{k+1}$. We begin by rewriting the update formula as follows, substituting $\hat{x}_k$ for $x_k + \delta x_k$, where $\hat{x}_k$ is the exact solution value at $t = kT$, and $\delta x_k = \hat{x}_k - x_k$

$$x_{k+1} = x_k + T(\theta f(x_k + \delta x_k) + (1-\theta)f(x_{k+1})) \tag{21}$$

Substituting (19) and rearranging the equation, intentionally isolating $x_{k+1}$, we get:

$$x_{k+1} = x_k\frac{1 - T\lambda\theta}{1 - T\lambda(1-\theta)} + \delta x_k\frac{1 + T\lambda\theta}{1 - T\lambda(1-\theta)} \tag{22}$$

The second term on the left side of (22) is $\delta x_{k+1}$, the error in $x_{k+1}$ as a function of $\delta x_k$:

$$\delta x_{k+1} = \delta x_k\frac{1 + T\lambda\theta}{1 - T\lambda(1-\theta)} \tag{23}$$

As we learned in ROB310 Lecture, satisfying the following inequality will ensure stability:

$$\left|\frac{\delta x_{k+1}}{\delta x_k}\right| \leq 1 \tag{24}$$

So:

$$\left|\frac{\delta x_{k+1}}{\delta x_k}\right| = \left|\frac{1 + T\lambda\theta}{1 - T\lambda(1-\theta)}\right| \tag{25}$$

Simplifying with $k = T\lambda$, we get:

$$\left|\frac{\delta x_{k+1}}{\delta x_k}\right| = \left|\frac{1 + k\theta}{1 - k(1-\theta)}\right| \tag{26}$$

Thus, the stability is:

$$\left|\frac{1 + k\theta}{1 - k(1-\theta)}\right| \leq 1 \tag{27}$$

### 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:

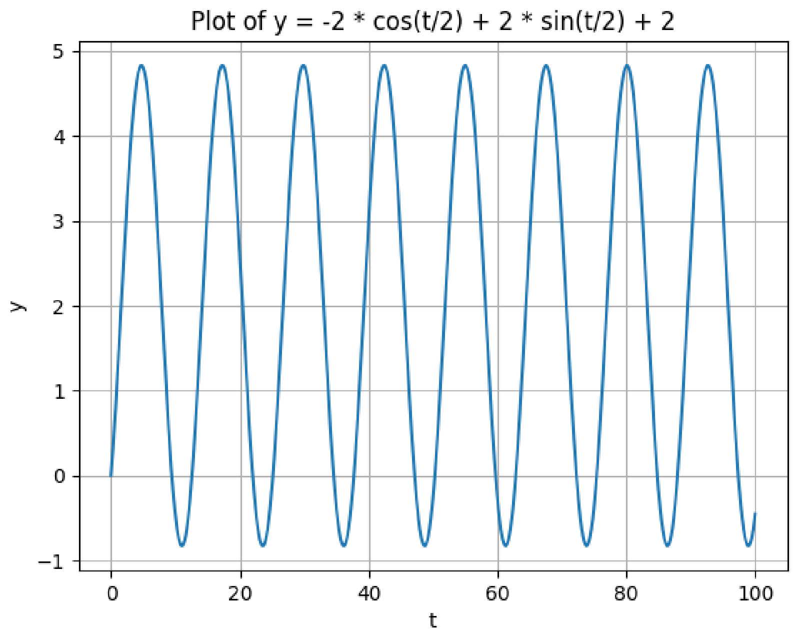

Figure 1:

### 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()
```

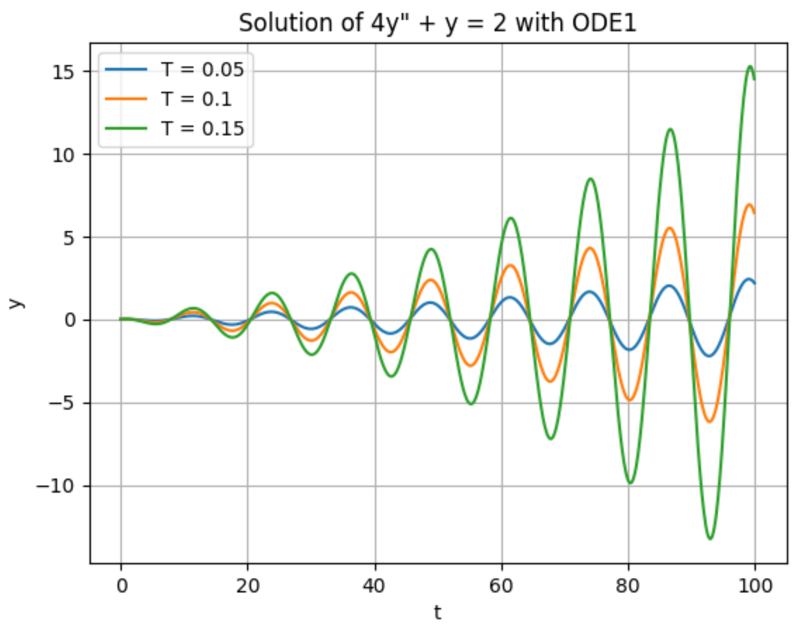

Figure 2:

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()
```

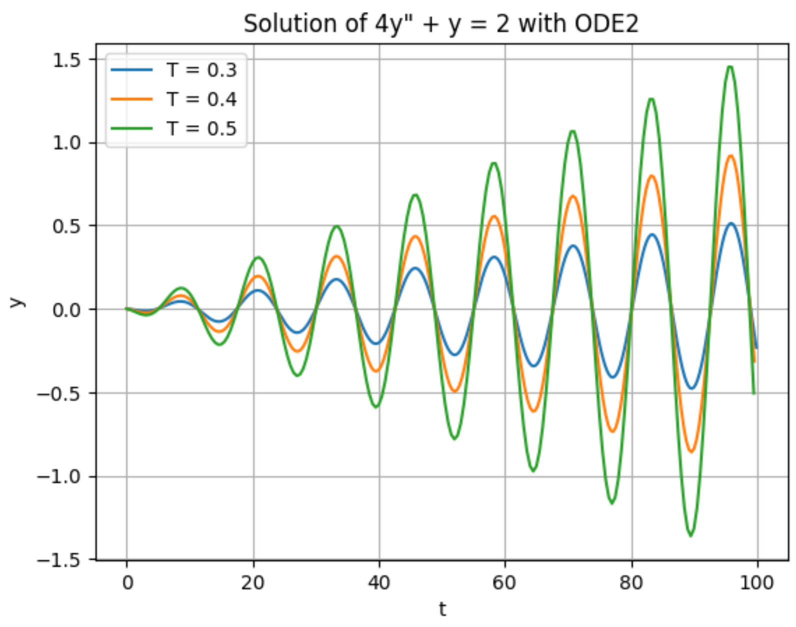

Figure 3:

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()
```

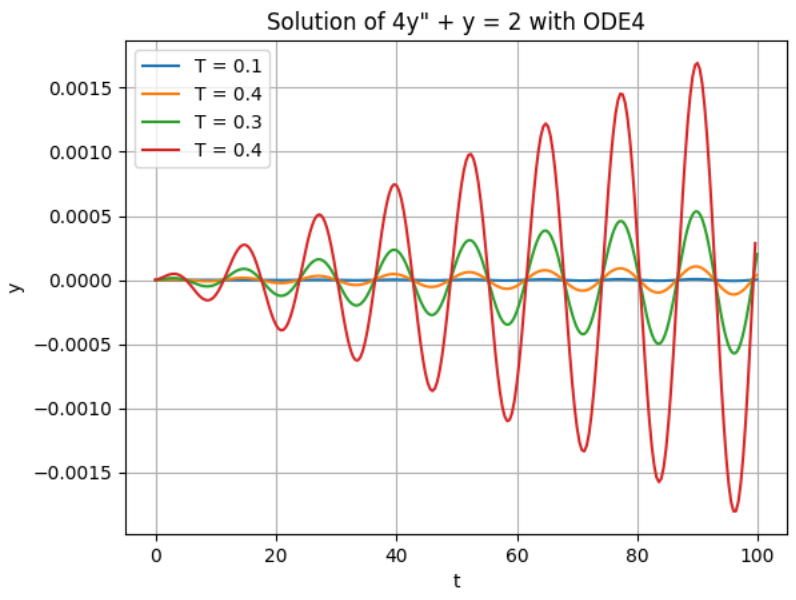

Figure 4:

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 $\pm 15$, while a step size of $T = 0.4$ was producing a magnitude of error near $\pm 0.0015$ with the `ODE4()` method.